Layer Zero

Solidity examples

by Ackee Blockchain

14.11.2022

Contents

1. Document Revisions

1.0	Audit Preview	November 21, 2022

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Medium	-
	Low	Medium	Medium	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Miroslav Škrabal	Lead Auditor
Lukáš Böhm	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Solidity-examples repository is used as a template or example for creating Omnichain Fungible (OFT) and Non-Fungible Tokens (ONFT) based on LayerZero's interoperability protocol.

Revision 1.0

Layer Zero engaged Ackee Blockchain to perform a security review of the Solidity-examples repository with a total time donation of 10 engineering days in a period between November 1 and November 14, 2022 and the lead auditor was Miroslav Škrabal. The audit has been performed on the commit `aff7d54` with the scope consisting of the following files:

- DistributeONFT721,
- BitLib.sol,
- ONFT721A.sol,
- WrappedOFT.sol.

We began our review by using static analysis tools, namely Slither and Woke. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- exploitability of the NFT reentrancy,
- the correctness of the distribution logic in DistributeONFT721,
- the correctness of the bit operations,
- ensuring that no duplicate NFT ids can be created,
- the correctness of the logic for sending and receiving tokens in cross-chain transactions,

- ensuring access controls are not too relaxed or too strict,
- ensuring that the logic of the contracts correspond to the specification,
- looking for common issues such as data validation.

Additionally, we implemented fuzz tests in Woke framework to simulate the behavior of the contracts in various scenarios that better reflect the real-world usage. The tests and their description can be found in the Appendix C.

Our review resulted in 12 findings, ranging from Info to Medium severity. The most severe ones are the possibility to create duplicate NFT ids in the DistributeONFT721 and the incorrectness of the crediting logic in the ONFT721A. Both of the issues however have a low likelihood.

Ackee Blockchain recommends Layer Zero to:

- address all the reported issues,
- be more careful when reusing code from different contracts as the M1 issue is a result of a copy-paste error.

See Revision 1.0 for the system overview of the codebase.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Solution*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: _creditTo can incorrectly mint	Medium	1.0	Reported
M2: Duplicating tokenIds	Medium	1.0	Reported
M3: Dangerous ownership transfer	Medium	1.0	Reported
M4: Data validation in constructor	Medium	1.0	Reported
M5: Data validation in constructor	Medium	1.0	Reported
W1: Usage of `solc` optimizer	Warning	1.0	Reported
W2: Unexpected side effect of function	Warning	1.0	Reported

	Severity	Reported	Status
W3: Accepting messages from untrusted remotes	Warning	1.0	Reported
I1: Constants are lowercase	Info	1.0	Reported
I2: Unnecessary function call	Info	1.0	Reported
I3: For-loop style	Info	1.0	Reported
I4: Unnecessary variables creation	Info	1.0	Reported

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

LzApp.sol

LzApp is a base contract for Layer Zero protocol applications. These applications send and receive messages through lzEndpoint, which is set in the constructor. All the discussed tokens inherit from LzApp.

NonblockingLzApp.sol

NonblockingLzApp inherits from LzApp and implements three additional functions for receive handling and message retry.

ONFT721ACore.sol

ONFT721ACore provides the interface to the core cross-chain message-sending functionality. It implements the sending and receiving functions which interact with the LzApp (and in turn with the lzEndpoint).

DistributeONFT721.sol

DistributeONFT721 is an omnichain NFT contract. Its main feature is that it allows distributing unused tokenIds to other chains. It uses an uint array as a bitmap that holds the tokenIds that can be minted. Each of the chains is initialized with a non-overlaping tokenId array which ensures non-colliding transfer across chains. Otherwise, the contract is based on the classical Layer Zero ONFT721 contract.

ONFT721A.sol

ONFT721A is an omnichain NFT contract. It is based on the ERC721A Azuki implementation of ERC721. ERC721A provides certain gas optimizations, mainly in the minting domain. The ERC721A mints the tokenIDs in a sequential manner, which imposes restrictions on it can be used in cross-chain scenarios. Because of this limitation, the ONFT721A has to be the first minter of the tokenIDs in the cross-chain scenarios.

BitLib.sol

BitLib is a library that is mainly utilized by the DistributeONFT721 contract. It provides functions for manipulating the bitmap array. It contains functions to find the most significant bit position and a function for counting the set bits.

WrappedOFT.sol

WrappedOFT inherits logic from OFTCore and is used to transfer an existing token from chain Y to chain X. If token A is not on chain X, this contract is deployed, and the token can be transferred from chain Y using ProxyOFT contract, which must be deployed on chain Y. After the transfer, the token is minted by this contract, and it is locked on a source chain ProxyOFT contract.

Actors

This part describes the actors of the system, their roles, and permissions.

Owner

The owner can perform multiple privileged operations:

- distribute tokenIds in DistributeONFT721 contract,
- set custom adapter params in the Core contracts,
- set various important parameters in the LzApp, mainly the trusted remotes.

Trust model

The owner can easily manipulate the trusted remotes and their addresses through the LzApp functions. That can affect the users' ability to perform cross-chain transactions. Additionally, in the case of the DistributeONFT721 contract, he can create duplicate ids.

M1: _creditTo can incorrectly mint

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	ONFT721A.sol	Type:	Contract logic

Listing 1. Excerpt from /contracts/token/onft721a/ONFT721A.sol#L26-L34[ONFT721A._creditTo]

```
    function _creditTo(uint16, address _toAddress, uint _tokenId) internal virtual override {
        require(!_exists(_tokenId) || (_exists(_tokenId) && ERC721A.ownerOf(_tokenId) == address(this)));

        if (!_exists(_tokenId)) {
            _safeMint(_toAddress, _tokenId);
        } else {
            safeTransferFrom(address(this), _toAddress, _tokenId);
        }
    }
```

Listing 2. Excerpt from /contracts/token/onft721a/ONFT721A.sol#L8-L10[ONFT721A.]

```
// DISCLAIMER: This contract can only be deployed on one chain when deployed and calling
// setTrustedRemotes with remote contracts. This is due to the sequential way 721A mints tokenIds.
// This contract must be the first minter of each token id!
```

Description

The `ONFT721A` contract contains a function `_creditTo` (see Listing 1) that is used to credit a user with an NFT token upon a cross-chain transfer. Because of the sequential way ERC721A mints tokenIds the `ONFT721A` has to be the first minter of each tokenId (see Listing 2).

However, the `_creditTo` function doesn't enforce this invariant. It contains the test: `!_exists(_tokenId)` which indicates the possibility that the `_tokenId` might not exist. But because the `ONFT721A` is the first minter of each tokenId, the `_tokenId` must upon receival exist (and the `address(this)` must be the owner).

Apart from that, the _creditTo function contains the following line: `_safeMint(_toAddress, _tokenId);`. But as can be seen from:

```
function _safeMint(address to, uint256 quantity) internal virtual {
        _safeMint(to, quantity, '');
    }
```

the second parameter is the number of tokens to mint, not the tokenId. If the call was actually made with the tokenId as an argument, the `_safeMint` function would mint a number of tokens equal to the tokenId. This, however, should not happen because of Listing 2 - the token should be already minted and owned by `address(this)`.

Exploit scenario

By a mistake, an NFT is minted on a different chain and it is sent to the `ONFT721A` contract. Because the `ONFT721A` contract wasn't the first minter of the token, the `_creditTo` function will mint a number of tokens equal to the tokenId.

Recommendation

Because of the sequential way ERC721A mints tokenIds it does not make sense for other chains to be the first minters. Therefore, the `_creditTo` function should be modified. It should not contain the test

`!_exists(_tokenId)` and it should not call `_safeMint` with the `_tokenId` as the second parameter.

Instead, it makes sense to require (or assert) that the token actually `exists`.

This error is most likely a result of a copy-paste error. It is strongly recommended to pay special attention when reusing code from different contracts that might have different semantics.

Go back to Findings Summary

M2: Duplicating tokenIds

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	DistributeONFT721.sol	Type:	Contract logic

Listing 3. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L179-L186[DistributeONFT721._verifyAmounts]

```
179     function _verifyAmounts(TokenDistribute[] memory _tokenDistribute) internal view returns (bool) {
180         uint _tokenDistributeLength = _tokenDistribute.length;
181         for(uint i = 0; i < _tokenDistributeLength; i++) {
182             uint tempTokenIds = tokenIds[_tokenDistribute[i].index];
183             uint result = tempTokenIds & _tokenDistribute[i].value;
184             if(result != _tokenDistribute[i].value) return false;
185         }
186         return true;
```

Listing 4. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L189-L194[DistributeONFT721._flipBits]

```
189     function _flipBits(TokenDistribute[] memory _tokenDistribute) internal {
190         uint _tokenDistributeLength = _tokenDistribute.length;
191         for(uint i = 0; i < _tokenDistributeLength; i++) {
192             tokenIds[_tokenDistribute[i].index] = tokenIds[_tokenDistribute[i].index] ^ _tokenDistribute[i].value;
193         }
194     }
```

Listing 5. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L169-L175[DistributeONFT721._nonblockingLzReceive]

```
        } else if(functionType == FUNCTION_TYPE_DISTRIBUTE) {
            (, TokenDistribute[] memory tokenDistribute) = abi.decode(_payload, (uint16, TokenDistribute[]));
            for(uint i = 0; i < tokenDistribute.length; i++) {
                uint temp = tokenIds[tokenDistribute[i].index];
                tokenIds[tokenDistribute[i].index] = temp | tokenDistribute[i].value;
            }
            emit ReceiveDistribute(_srcChainId, _srcAddress, tokenDistribute);
```

Description

It is possible to create duplicate tokenIds if one gains control of the owner account. The attack can be done through the `distributeTokens` function.

The `distributeToken` function accepts an array of `TokenDistribute` as an argument. If the mentioned array contains multiple elements that point to the same index, it can lead to the creation of new token ids.

Inside the function, there are the two following function calls:

- _verifyAmounts
- _flipBits

`_verifyAmounts` verifies that the amount in the tokenId that corresponds to the given `_tokenDistribute[i].index` is as least as big as the one in the TokenDistribute array (see Listing 3).

It can be seen that it does not perform a check whether a given index has already been used.

`_flipBits` is a function that flips the bits of the corresponding tokenId. It performs `xor` of a tokenId with the amount from the corresponding element from the `TokenDistribute` array (see Listing 4). Thus if the number of the repeated elements with the same index is even, the xor will be undone. (and thus, the tokenId won't be subtracted by the sending amount).

The other chain upon receiving the `TokenDistribute` array performs an `or` (see Listing 5) with its tokenIds and thus new tokenIds will be created.

Test that confirms this issue is provided:

```
def test_repeating_indexes(e1: LZEndpointMock, e2: LZEndpointMock, a:
Dict[str, Address]):
    user1 = Address("0x4516d3f5b2e5b6e1f062f18e3d2f5f6a77469285")
    num_of_ids = 30
    init_val = int(250*"1"+6*"0", 2)
    init_value_list = [init_val for i in range(num_of_ids)]
    d1 = DistributeONFT721Mock.deploy(1, Address(e1), [i for i in
range(num_of_ids)],
                                              init_value_list,
from_=a["owner"])
    d2 = DistributeONFT721Mock.deploy(2, Address(e2), [i for i in
range(num_of_ids, 2*num_of_ids)],
                                              init_value_list,
from_=a["owner"])
    e1.setDestLzEndpoint(Address(d2), Address(e2))
    e2.setDestLzEndpoint(Address(d1), Address(e1))
    d1.setTrustedRemoteAddress(e2.getChainId(), bytes.fromhex(
str(Address(d2))[2:]), from_=a["owner"])
    d2.setTrustedRemoteAddress(e1.getChainId(), bytes.fromhex(
str(Address(d1))[2:]), from_=a["owner"])

    assert d1.tokenIds(0) == init_val
    assert d2.tokenIds(0) == 0

    token_distribute = [(0, init_val), (0, init_val)]

    d1.distributeTokens(d2.chainId(), token_distribute, a["owner"],
a["owner"],  from_=a["owner"], value=Wei(10**18))
```

```
    assert d1.tokenIds(0) == init_val
    assert d2.tokenIds(0) == init_val
```

From the `asserts` it can be seen that new tokenIds are created.

Exploit scenario

The owner incorrectly constructs (or his private key is hacked and a malicious payload is created intentionally) the `TokenDistribute` array and sends it to the `distributeTokens` function. The incorrectness lies in repeating the same index multiple times. As a result of the properties of the or and xor functions, new tokenIds are created (following the processes described above).

Recommendation

Ensure that the `TokenDistribute` array does not contain multiple elements with the same index.

Go back to Findings Summary

M3: Dangerous ownership transfer

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	LZApp.sol	Type:	Access Control

Description

`LZApp.sol` inherits functionality to transfer the ownership from `Ownable` contract.

```
function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
```

However, the transfer function does not have a robust verification mechanism for the proposed owner address. If a wrong owner address is accidentally passed to it, the error cannot be recovered. Thus passing a wrong address can lead to irrecoverable mistakes.

Exploit scenario

The current owner Alice wants to transfer the ownership to Bob. Alice calls the `transferOwnership` function but supplies the wrong address by mistake. As a result, the ownership will be passed to the wrong and possibly dead address.

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

The current owner Alice wants to transfer the ownership to Bob. The two-step process would have the following steps:

- Alice proposes a new owner, namely Bob. This proposal is saved to a variable candidate.
- The candidate Bob calls the function for accepting the ownership, which verifies that the caller is the new proposed candidate.
- If the verification passes, the function sets the caller as the new owner.

If Alice proposes the wrong candidate, she can change it. However, it can happen, though with an extremely low probability, that the wrong candidate is malicious (most often, it would be a dead address) and is able to accept the ownership in the meantime.

An authentication mechanism can also be employed to prevent the malicious candidate from accepting the ownership, though such a mechanism is almost never used in practice.

Go back to Findings Summary

M4: Data validation in constructor

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	DistributeONFT721.sol	Type:	Data validation

Description

The DistributeONFT721 contract lacks robust data validation in the constructor. That can lead to accepting wrong data and initializing the contract into an invalid state. As a consequence, the contracts can behave unexpectedly. Specifically, the contract does not perform any data validation on the `_indexArray` and `_valueArray` arguments.

Exploit scenario

The contract is deployed with the incorrect values for the `_valueArray`. Specifically, the elements of this array do not follow the invariant that each `uint` in the array uses at most 250 bits of its allotted 256 bits to represent token ids and it uses more bits. As a result the equation for calculating the token ids:

```
tokenId = (255 - position) + (i * NUM_TOKENS_PER) + 1;
```

can produce duplicate token ids.

To prove this, imagine the following simplified scenario:

- uints have 4bits,
- NUM_TOKENS_PER = 3 (ie it is expected that the least significant bit is 0).

Now consider that the contract is initialized with the following uints: `1111` and

`1111`.

Let's calculate the token ids based on this `_valueArray` and let's start from the id 3:

- the uint has 4 bits, 3 were already used, so the current state is `0001`, `1111`,
- new id is calculated as: 3 - 0 + (0 * 3) + 1 = 4,
- the following id will be calculated from the next uint (`1111`) as: 3 - 3 + (1 * 3) + 1 = 4 and thus we have a duplicate id.

Recommendation

Add more stringent data validation for the `_valueArray`. Specifically, ensure that each uint in the array uses at most 250 bits of its allotted 256 bits to represent token ids.

Go back to Findings Summary

M5: Data validation in constructor

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	LzApp.sol	Type:	Data validation

Description

The LzApp contract lacks robust data validation in the constructor. That can lead to accepting wrong `_endpoint` address and initializing the contract into an invalid state. As a consequence, the contracts can behave in an unexpected way.

Exploit scenario

An incorrect value of `_endpoint` is passed to the LzApp constructor. Instead of reverting, the call succeeds. If such a mistake is not discovered quickly and the contracts are not redeployed, the protocol can behave in an undefined way.

Recommendation

Add more stringent data validation for `_endpoint`. At the very least, this would include a zero-address check. Ideally, it is recommended to define a getter in the `_endpoint` such as `contractId()` that would return a hash of an identifier unique to the (project, contract). An example would be `keccak256("Layer Zero Endpoint")`. Upon constructing the `LZApp` the identifier would be retrieved from the passed-in `_endpoint` address and compared to the expected value.

Go back to Findings Summary

W1: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

W2: Unexpected side effect of function

Impact:	Warning	Likelihood:	N/A
Target:	DistributeONFT721.sol	Type:	Unexpected code logic

Listing 6. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L227-L231[DistributeONFT721._getNextMintTokenId]

```
            uint position = BitLib.mostSignificantBitPosition(currentTokenId);
            uint temp = 1 << position;
            tokenIds[i] = tokenIds[i] ^ temp;
            tokenId = (255 - position) + (i * NUM_TOKENS_PER) + 1;
            break;
```

Description

The `_getNextMintTokenId()` function, apart from getting the nextMintTokenId, also has the side-effect of clearing the 1bit flag for the corresponding id. The clearing of the flag is not expected from the function name and is not documented in the function description and as such might be unexpected for the caller.

Vulnerability scenario

A developer of a LzApp that uses the DistributeONFT721 contract might not expect the clearing of the flag and might not be aware of it. This could lead to unexpected behavior of the LzApp because the developer could implement their own clearing logic.

Recommendation

Either rename the function to `_getNextMintTokenIdAndClearFlag()` or add a comment to the function description that the flag is cleared. Optionally, the clearing of the flag could be moved to a separate function.

Go back to Findings Summary

W3: Accepting messages from untrusted remotes

Impact:	Warning	Likelihood:	N/A
Target:	LzApp.sol, NonblockingLzApp.sol	Type:	Data validation

Description

The LzApp contract uses `trustedRemoteLookup` mapping to validate whether a given remote can be trusted or not. The values to the mapping are set by the contract owner in the function `setTrustedRemote`. If special conditions are met, a call from an untrusted remote can be executed. The `retryMessage` function enables this as it performs no check on whether the given remote is trusted or not. Additionally, it is enabled by the fact that a given remote can be set as untrusted after it is set as trusted.

Vulnerability scenario

1. Owner sets a new trusted remote `A` using `setTrustedRemote` function.
2. `A` gets compromised by a malicious actor.
3. A message from `A` is delivered. The execution fails inside the `_blockingLzReceive`, and the message is thus stored to `failedMessages` mapping.
4. Because `A` was compromised, the owner sets `A` as untrusted. He does so by setting the value for key `A` to the default value.
5. The failed message gets reexecuted by calling `retryMessage`, and this time the call succeeds. As a result, a message from an *untrusted* remote gets executed.

Recommendation

Inside the `retryMessage` function, implement an additional check whether or not a given remote is trusted. This check is needed because a remote can become untrusted while a message is stored in the `failedMessages` buffer.

Go back to Findings Summary

I1: Constants are lowercase

Impact:	Info	Likelihood:	N/A
Target:	Bitlib.sol	Type:	Convention vialoation

Listing 7. Excerpt from /contracts/util/BitLib.sol#L9-L16[BitLib.]

```
9     uint256 constant m1  =
  0x5555555555555555555555555555555555555555555555555555555555555555;
10    uint256 constant m2  =
  0x3333333333333333333333333333333333333333333333333333333333333333;
11    uint256 constant m4  =
  0x0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f;
12    uint256 constant m8  =
  0x00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff00ff;
13    uint256 constant m16 =
  0x0000ffff0000ffff0000ffff0000ffff0000ffff0000ffff0000ffff0000ffff;
14    uint256 constant m32 =
  0x00000000ffffffff00000000ffffffff00000000ffffffff00000000ffffffff;
15    uint256 constant m64 =
  0x0000000000000000ffffffffffffffff0000000000000000ffffffffffffffff;
16    uint256 constant m128 =
  0x00000000000000000000000000000000ffffffffffffffffffffffffffffffff;
```

Description

The `BitLib` library defines constants that are not in all caps (see Listing 7). This is a convention violation that impairs code readability.

Recommendation

Rename the mentioned variables to all caps.

Go back to Findings Summary

I2: Unnecessary function call

Impact:	Info	Likelihood:	N/A
Target:	DistributeONFT721.sol	Type:	Gas optimization

Listing 8. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L84-L88[DistributeONFT721.countAllSetBits]

```
84    function countAllSetBits() public view returns (uint count) {
85        uint tokenIdsLength = tokenIds.length;
86        for(uint i = 0; i < tokenIdsLength; i++) {
87            count += BitLib.countSetBits(tokenIds[i]);
88        }
```

Description

`countAllSetBits()` unnecessarily calls `countSetBits` for tokenIds equal to 0 (see Listing 8). If a simple zero check was performed the function would be more gas efficient.

Recommendation

Implement a simple zero check to avoid unnecessary function calls. If the value of the tokenId is zero, the result is known to be 0.

Go back to Findings Summary

I3: For-loop style

Impact:	Info	Likelihood:	N/A
Target:	DistributeONFT721.sol	Type:	Gas optimization, code style

Listing 9. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L68-L73[DistributeONFT721.constructor]

```
    constructor(string memory _name, string memory _symbol, address _layerZeroEndpoint, uint[] memory _indexArray, uint[] memory _valueArray) ONFT721(_name, _symbol, _layerZeroEndpoint){
        uint _indexArrayLength = _indexArray.length;
        for(uint i; i < _indexArrayLength;) {
            tokenIds[_indexArray[i]] = _valueArray[i];
            unchecked{++i;}
        }
```

Listing 10. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L86-L88[DistributeONFT721.countAllSetBits]

```
        for(uint i = 0; i < tokenIdsLength; i++) {
            count += BitLib.countSetBits(tokenIds[i]);
        }
```

Description

The `DistributeONFT721` contract does not use a consistent for-loop style (see [Listing 9](#) and [Listing 10](#)).

The contract heavily depends on looping over arrays. Because of that, it would be ideal to use the most gas-efficient style of for-loops. The for loop from the constructor (see [Listing 9](#)) is very efficient, though it has to be ensured that the incrementation of the iterator variable can not lead to overflow.

Recommendation

Use a consistent for-loop style and consider using a more gas-efficient style for for-loops. At least, use pre-incrementation instead of post-incrementation of the iterator variable.

Go back to Findings Summary

I4: Unnecessary variables creation

Impact:	Info	Likelihood:	N/A
Target:	DistributeONFT721.sol	Type:	Gas optimization, code style

Listing 11. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L207-L210[DistributeONFT721._countTokenDistributeSize]

```
        for(uint i = 0; i < tokenIdsLength; i++) {
            uint currentTokenId = tokenIds[i];
            count += BitLib.countSetBits(currentTokenId);
            if(count >= _amount) return i + 1;
```

Listing 12. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L181-L184[DistributeONFT721._verifyAmounts]

```
        for(uint i = 0; i < _tokenDistributeLength; i++) {
            uint tempTokenIds = tokenIds[_tokenDistribute[i].index];
            uint result = tempTokenIds & _tokenDistribute[i].value;
            if(result != _tokenDistribute[i].value) return false;
```

Listing 13. Excerpt from /contracts/token/onft/extension/DistributeONFT721.sol#L109-L110[DistributeONFT721.getDistributeTokens]

```
        for(uint i = 0; i < tokenIdsLength; i++) {
            uint currentTokenId = tokenIds[i];
```

Description

The `DistributeONFT721` contract unnecessarily creates local variables in multiple places (see the listings). This is a gas optimization issue that manifests itself mainly in loops as they introduce a source block that gets executed for each iteration.

Recommendation

Where possible (where semantics don't change and where it does not affect readability), local variables should be declared outside the block and they should be reused. Such an approach will lead to gas savings.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Layer Zero: Solidity examples, 14.11.2022.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Appendix C: Fuzz tests

The following sections describe the fuzz tests that we implemented for DistributeONFT, ONFT721A and BitLib. The tests were implemented using the Woke fuzzer and testing framework. They were run thousands of times without failing. The tests implement the logic of the contracts and as such provide a reference to the expected behavior of the contracts. The state of the contracts is periodically checked during the test run against the expected state.

C.1. DistributeONFT fuzz tests

The fuzz tests focused to simulate the real-world usage of the contract. They randomized function calls and their arguments. The randomized calls were: minting, transferring, cross-chain transferring (simulated on one chain) and distributing. After each call, the state of the contracts was checked against the expected state.

```
class Test:
    init_val: int
    init_value_list: List[int]
    num_of_ids: int
    accounts: Dict[str, Address]
    # dict from (chainId, nftId) to Adress
    nft_owners: Dict[Tuple[int, int], Address]
    # dict from chainId to List of ints (same as in the contract), which
represent nftIds
    mintable_ids: Dict[int, List[int]]
    # dict from chainId to set of minted nftIds
    minted_ids: Dict[int, Set[int]]
    e1: LZEndpointMock
    e2: LZEndpointMock
    d1: DistributeONFT721Mock
    d2: DistributeONFT721Mock
    #count of set bits chainId -> numOfSetBits
    set_bits: Dict[int, int]
```

```
    #list of all deployed distributable contracts, in sorted order of
chainIds
    distributable: List[DistributeONFT721Mock]
    chain_ids: List[int]


    def __init__(self):
        self.init_val = int(250*"1"+6*"0", 2)
        self.num_of_ids = 30
        self.init_value_list = [self.init_val for i in range(
self.num_of_ids)]
        self.accounts: Dict[str, Address] = {
            "alice": Address("0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266"),
            "bob":   Address("0x70997970c51812dc3a010c7d01b50e0d17dc79c8"),
            "charlie": Address(
"0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc"),
            "david": Address("0x90f79bf6eb2c4f870365e785982e1f101e93b906"),
            "owner": Address("15d34aaf54267db7d7c367839aaf71a00a2c6a65"),
        }

        self.nft_owners = {}
        self.mintable_ids = {1 : self.init_value_list + [0 for i in range
(self.num_of_ids)],
                             2 : [0 for i in range(self.num_of_ids)] +
self.init_value_list}
        self.minted_ids = {1 : set(), 2 : set()}
        self.e1 = LZEndpointMock.deploy(1, from_=self.accounts["owner"])
        self.e2 = LZEndpointMock.deploy(2, from_=self.accounts["owner"])
        self.chain_ids = [1, 2]
        self.d1 = DistributeONFT721Mock.deploy(1, Address(self.e1), [i for
i in range(self.num_of_ids)],
                                               self.init_value_list,
from_=self.accounts["owner"])
        self.d2 = DistributeONFT721Mock.deploy(2, Address(self.e2), [i for
i in range(self.num_of_ids, 2*self.num_of_ids)],
                                               self.init_value_list,
from_=self.accounts["owner"])
        self.set_bits = {1 : self.num_of_ids * NUM_TOKENS_PER, 2 :
self.num_of_ids * NUM_TOKENS_PER}
        #distributables have to be added in the order of chainIds
        self.distributable = [self.d1, self.d2]
        self.e1.setDestLzEndpoint(Address(self.d2), Address(self.e2))
```

```
        self.e2.setDestLzEndpoint(Address(self.d1), Address(self.e1))
        self.e1.setDestLzEndpoint(Address(self.d2), Address(self.e2))
        self.e2.setDestLzEndpoint(Address(self.d1), Address(self.e1))
        self.d1.setTrustedRemoteAddress(self.e2.getChainId(),
bytes.fromhex(str(Address(self.d2))[2:]), from_=self.accounts["owner"])
        self.d2.setTrustedRemoteAddress(self.e1.getChainId(),
bytes.fromhex(str(Address(self.d1))[2:]), from_=self.accounts["owner"])
        for i in range(2*self.num_of_ids):
            if i < self.num_of_ids:
                assert self.d1.tokenIds(i) == self.init_val
                assert self.d2.tokenIds(i) == 0
            else:
                assert self.d1.tokenIds(i) == 0
                assert self.d2.tokenIds(i) == self.init_val


    def add_ether(self):
        for d in self.distributable:
            Address(d).balance = Wei.from_ether(10**6)
        Address(self.e1).balance = Wei.from_ether(10**6)
        Address(self.e2).balance = Wei.from_ether(10**6)
        for a in self.accounts.keys():
            self.accounts[a].balance = Wei.from_ether(10**6)


    def get_msb(self, n: int) -> int:
        in_bin = '{0:0256b}'.format(n)
        pos = in_bin.find("1")
        return 255 - pos if n else 0

    def replace_char_in_str(self, s: str, pos: int, c: str) -> str:
        return s[:pos] + c + s[pos+1:]

    def get_mint_id(self, chain_id: int) -> int:
        ids = self.mintable_ids[chain_id]
        for cnt, i in enumerate(ids):
            if i != 0:
                pos = self.get_msb(i)
                in_bin = '{0:0256b}'.format(i)
                in_bin = self.replace_char_in_str(in_bin, 255-pos, "0")
                ids[cnt] = int(in_bin, 2)
                self.set_bits[chain_id] -= 1
```

```
                return (255 - pos) + (cnt * NUM_TOKENS_PER) + 1
        #TODO what if no mintale id?
        return 0


    def get_nft_not_owned_by_contract(self):
        nft = choice(list(self.nft_owners.keys()))
        #cant send nft from the address of the base contract
        #on the other hand if the contract owns an nft there must be a
corresponding owner on the other chain
        if self.nft_owners[nft] == Address(self.distributable[nft[0]-1]):
            #TODO make more general to support multiple chains
            nft = (1 if nft[0] == 2 else 2, nft[1])
        return nft


    @flow
    @weight(70)
    def flow_mint(self):
        self.add_ether()
        d = choice(self.distributable)
        a = choice(list(self.accounts.values()))
        minted_id = d.mint(from_=a)
        minted_id_ref = self.get_mint_id(d.getChainId())
        print(f"chainId: {d.getChainId()}, minted_id: {minted_id},
minted_id_ref: {minted_id_ref}")
        assert minted_id == minted_id_ref
        self.nft_owners[(d.getChainId(), minted_id)] = a
        #nft can be only minted once - upon mint it has to be fresh
        for cid in self.chain_ids:
            assert minted_id not in self.minted_ids[cid]
        self.minted_ids[d.getChainId()].add(minted_id)
        assert self.nft_owners[(d.getChainId(), minted_id)] ==
d.ownerOf(minted_id)


    @flow
    @weight(50)
    def flow_send(self):
        self.add_ether()
        if not len(self.nft_owners):
            return
```

```
        #get a random nft (and a random chainId)
        nft = self.get_nft_not_owned_by_contract()
        #get the corresponding owner
        owner = self.nft_owners[nft]
        new_owner = choice(list(self.accounts.values()))
        #the first elem of the nft tuple is the chainId (chain id starts
from 1 - we have to subtract 1)
        d = self.distributable[nft[0]-1]
        d.transferFrom(owner, new_owner, nft[1], from_=owner)
        self.nft_owners[nft] = new_owner
        assert self.nft_owners[nft] == d.ownerOf(nft[1])


    @flow
    @weight(50)
    def flow_send_crosschain(self):
        self.add_ether()
        if not len(self.nft_owners):
            return
        nft = self.get_nft_not_owned_by_contract()
        owner = self.nft_owners[nft]
        new_owner = choice(list(self.accounts.values()))
        d = self.distributable[nft[0]-1]
        dstChain = choice(self.chain_ids)
        #get a random dstChain that is different from the current chain
        while dstChain == nft[0]:
            dstChain = choice(self.chain_ids)
        owner.balance = Wei(10**20)
        d.sendFrom(owner, dstChain, bytes.fromhex(str(new_owner)[2:]),
nft[1], owner, owner, bytes(), from_=owner, value=Wei(10**18))
        self.nft_owners[nft] = Address(d)
        #the nft is now owned by any user (or could be owned by the
contract itself)
        assert (dstChain, nft[1]) not in self.nft_owners or
self.nft_owners[(dstChain, nft[1])] == Address(self.distributable[dstChain-
1])
        self.nft_owners[(dstChain, nft[1])] = new_owner
        assert(self.nft_owners[nft] == d.ownerOf(nft[1]))
        dstDistributable = self.distributable[dstChain-1]
        assert(self.nft_owners[(dstChain, nft[1])] ==
dstDistributable.ownerOf(nft[1]))
```

```
    @flow
    @weight(5)
    def flow_distribute_valid_ids(self):
        d = choice(self.distributable)
        token_distribute = []
        num_of_changed_bits = 0
        for cnt, tk_val in enumerate(self.mintable_ids[d.chainId()]):
            #we only want to distribute ids that are not minted -> we care
only about values > 0
            if tk_val != 0:
                #distribute the given id only with 50% chance
                if choice([True, False]):
                    continue
                in_bin = '{0:0256b}'.format(tk_val)
                distribute = ""
                for bit_pos, bit in enumerate(in_bin):
                    if bit == "0":
                        distribute += "0"
                    else:
                        #add a bit==1 with a 5% chance
                        if randint(0, 100) <= 5:
                            distribute += "1"
                            #if we distribute a bit==1 we need to change
the bit in the mintable_ids to 0
                            in_bin = self.replace_char_in_str(in_bin,
bit_pos, "0")
                            num_of_changed_bits += 1
                        else:
                            distribute += "0"
                self.mintable_ids[d.chainId()][cnt] = int(in_bin, 2)
                token_distribute.append((cnt, int(distribute, 2)))
        dstChain = choice(self.chain_ids)
        while dstChain == d.chainId():
            dstChain = choice(self.chain_ids)
        #modify the num of bits according to the number of bits that are to
be distributed
        self.set_bits[d.chainId()] -= num_of_changed_bits
        self.set_bits[dstChain] += num_of_changed_bits
        #distribute the ids to the other chain - only modifies the local
python state
        for i, val in token_distribute:
```

```
            #take the ids from the dstChain
            token_id_bin = '{0:0256b}'.format(
self.mintable_ids[dstChain][i])
            val_bin = '{0:0256b}'.format(val)
            for pos, bit in enumerate(val_bin):
                if bit == "1":
                    #if the bit is 1 then this bit is distributed and we
need to set this bit to 1 on the dstChain
                    token_id_bin = self.replace_char_in_str(token_id_bin,
pos, "1")
            self.mintable_ids[dstChain][i] = int(token_id_bin, 2)
        d.distributeTokens(dstChain, token_distribute, self
.accounts["owner"], self.accounts["owner"],  from_=self.accounts["owner"],
value=Wei(10**18))
        contract_token_ids = d.getTokenIds()
        #woke returns list of tuples, this untuples the list
        contract_token_ids = [i for i in contract_token_ids]
        assert self.mintable_ids[d.chainId()] == contract_token_ids
        contract_token_ids = self.distributable[dstChain-1].getTokenIds()
        assert self.set_bits[d.chainId()] == d.countAllSetBits()
        assert self.set_bits[dstChain] == self.distributable[dstChain-
1].countAllSetBits()


    #tests that each contract can mint the correct amount of tokens
    @invariant
    def invariant_mintable_ids(self):
        self.add_ether()
        for chain_id in self.chain_ids:
            d = self.distributable[chain_id-1]
            contract_token_ids = d.getTokenIds()
            #woke returns list of tuples, this untuples the list
            contract_token_ids = [i for i in contract_token_ids]
            assert self.mintable_ids[d.chainId()] == contract_token_ids


    #tests that nfts have the correct owner
    @invariant
    def invariant_nft_owner(self):
        self.add_ether()
        logger.info("invariant_nft_owner")
        for chain_id in self.chain_ids:
```

```
            for nft_id in self.minted_ids[chain_id]:
                assert self.distributable[chain_id-1].ownerOf(nft_id) ==
self.nft_owners[(chain_id, nft_id)]
```

C.2. ONFT721A fuzz tests

The fuzz tests focused to simulate the real-world usage of the contract. They randomized function calls and their arguments. The randomized calls were: minting, transferring, cross-chain transferring (simulated on one chain) and setting approvals. After each call, the state of the contracts was checked against the expected state.

```
class Test:
    accounts: Dict[str, Address]
    # dict from (chainId, nftId) to Adress
    nft_owners: Dict[Tuple[int, int], Address]
    # dict from chainId to set of minted nftIds
    minted_ids: Dict[int, Set[int]]
    e1: LZEndpointMock
    e2: LZEndpointMock
    onft_base: ONFT721AMock
    onft_secondary: ONFT721Mock
    #list of all deployed onft contracts in sorted order of chainIds
    onfts: List[ONFT721AMock]
    #list of chain ids in ascending order
    chain_ids: List[int]
    #dict from onft contract to its corresponding chainId
    cid = Dict[Address, int]
    #the id of the next nft to be minted
    current_nft_id: int
    #max amount of nfts that can be minted in a batch
    batch_size: int
    #dict from [chainId, Address] to number of nfts owned by that address
    balances: Dict[Tuple[int, Address], int]
    #dict from (chainId, nftId) to (owner, spender)
    approvals: Dict[Tuple[int, int], Tuple[Address, Address]]
```

```
    def __init__(self):
        self.accounts: Dict[str, Address] = {
            "alice": Address("0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266"),
            "bob":   Address("0x70997970c51812dc3a010c7d01b50e0d17dc79c8"),
            "charlie": Address(
"0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc"),
            "david": Address("0x90f79bf6eb2c4f870365e785982e1f101e93b906"),
            "owner": Address("15d34aaf54267db7d7c367839aaf71a00a2c6a65"),
        }
        self.nft_owners = {}
        self.minted_ids = {1: set(), 2: set()}
        self.e1 = LZEndpointMock.deploy(1, from_=self.accounts["owner"])
        self.e2 = LZEndpointMock.deploy(2, from_=self.accounts["owner"])
        self.chain_ids = [1, 2]
        self.oft_base =      ONFT721AMock.deploy("onfta", "onfta",
Address(self.e1), from_=self.accounts["owner"])
        self.oft_secondary = ONFT721Mock.deploy("onft", "onft",
Address(self.e2), from_=self.accounts["owner"])
        self.onfts = [self.oft_base, self.oft_secondary]
        self.cid = {self.oft_base: 1, self.oft_secondary: 2}
        self.e1.setDestLzEndpoint(Address(self.oft_secondary),
Address(self.e2))
        self.e2.setDestLzEndpoint(Address(self.oft_base), Address(
self.e1))
        self.oft_base.setTrustedRemoteAddress(self.e2.getChainId(),
bytes.fromhex(str(Address(self.oft_secondary))[2:]), from_=self
.accounts["owner"])
        self.oft_secondary.setTrustedRemoteAddress(self.e1.getChainId(),
bytes.fromhex(str(Address(self.oft_base))[2:]), from_=self
.accounts["owner"])
        self.current_nft_id = 0
        self.batch_size = 5
        self.balances = defaultdict(lambda: 0)
        self.approvals = {}


    def add_ether(self):
        for o in self.onfts:
            Address(o).balance = Wei.from_ether(10**6)
        Address(self.e1).balance = Wei.from_ether(10**6)
        Address(self.e2).balance = Wei.from_ether(10**6)
        for a in self.accounts.keys():
```

```
            self.accounts[a].balance = Wei.from_ether(10**6)


    def get_batch_mint_ids(self, batch_sz: int) -> int:
        mint_id = self.current_nft_id
        self.current_nft_id += batch_sz
        return (mint_id, mint_id + batch_sz)


    def get_nft_not_owned_by_contract(self):
        nft = choice(list(self.nft_owners.keys()))
        #cant send nft from the address of the base contract
        #on the other hand if the contract owns an nft there must be a
corresponding owner on the other chain
        if self.nft_owners[nft] == Address(self.onfts[nft[0]-1]):
            #if the nft is owned by a contract then the owner of the same
id on the second chain should be a user
            nft = (1 if nft[0] == 2 else 2, nft[1])
        return nft


    @flow
    @weight(30)
    def flow_mint_nft(self):
        self.add_ether()
        o = self.onft_base
        a = choice(list(self.accounts.values()))
        batch_size = randint(1, self.batch_size)
        o.mint(batch_size, from_=a)
        (start_id, end_id) = self.get_batch_mint_ids(batch_size)
        print(f"chainId: {self.cid[o]}, minted_ids: {start_id} - {end_id}")
        #nft ids can't be already minted on none of the chains
        for cid in self.chain_ids:
            for nft_id in range(start_id, end_id):
                assert nft_id not in self.minted_ids[cid]
        for id in range(start_id, end_id):
            self.nft_owners[(1, id)] = a
            self.minted_ids[self.cid[o]].add(id)
            assert self.nft_owners[(self.cid[o], id)] == o.ownerOf(id)
        self.balances[(self.cid[o], a)] += batch_size
        assert self.balances[(self.cid[o], a)] == o.balanceOf(a)
```

```
    @flow
    @weight(30)
    def flow_transfer(self):
        self.add_ether()
        if not len(self.nft_owners):
            return
        #get a random nft (and a random chainId)
        nft = self.get_nft_not_owned_by_contract()
        #get the corresponding owner
        owner = self.nft_owners[nft]
        new_owner = choice(list(self.accounts.values()))
        #the first elem of the nft tuple is the chainId (chain id starts
from 1 - we have to subtract 1)
        o = self.onfts[nft[0]-1]
        o.transferFrom(owner, new_owner, nft[1], from_=owner)
        #if nft is transfered then the associated approval is cleared
        if nft in self.approvals:
            del self.approvals[nft]
        self.nft_owners[nft] = new_owner
        assert self.nft_owners[nft] == o.ownerOf(nft[1])
        self.balances[(self.cid[o], owner)] -= 1
        self.balances[(self.cid[o], new_owner)] += 1
        assert self.balances[(self.cid[o], owner)] == o.balanceOf(owner)
        assert self.balances[(self.cid[o], new_owner)] ==
o.balanceOf(new_owner)


    @flow
    @weight(30)
    def flow_send_crosschain(self):
        self.add_ether()
        if not len(self.nft_owners):
            return
        nft = self.get_nft_not_owned_by_contract()
        owner = self.nft_owners[nft]
        new_owner = choice(list(self.accounts.values()))
        #if there is an approved address for the given nft, use it to test
the approve fucntionality
        sender = owner
        if nft in self.approvals:
            sender = self.approvals[nft][1]
```

```
            del self.approvals[nft]
        o = self.onfts[nft[0]-1]
        dstChain = choice(self.chain_ids)
        #get a random dstChain that is different from the current chain
        while dstChain == nft[0]:
            dstChain = choice(self.chain_ids)
        o.sendFrom(owner, dstChain, bytes.fromhex(str(new_owner)[2:]),
nft[1], owner, owner, bytes(), from_=sender, value=Wei(10**18))
        if nft in self.approvals:
            del self.approvals[nft]
        self.nft_owners[nft] = Address(o)
        #the nft is now owned by any user (or could be owned by the
contract itself)
        assert (dstChain, nft[1]) not in self.nft_owners or
self.nft_owners[(dstChain, nft[1])] == Address(self.onfts[dstChain-1])
        self.nft_owners[(dstChain, nft[1])] = new_owner
        assert(self.nft_owners[nft] == o.ownerOf(nft[1]))
        dstOnft = self.onfts[dstChain-1]
        assert(self.nft_owners[(dstChain, nft[1])] == dstOnft.ownerOf(nft[
1]))
        #decreases the senders nft balance
        self.balances[(nft[0], owner)] -= 1
        #increases the receivers nft balance on the destination chain
        self.balances[(dstChain, new_owner)] += 1
        assert self.balances[(self.cid[o], owner)] == o.balanceOf(owner)
        assert self.balances[(dstChain, new_owner)] == self.onfts[dstChain-
1].balanceOf(new_owner)


    @flow
    @weight(30)
    def flow_approve(self):
        if not len(self.nft_owners):
            return
        #approvals: (chainId, nftId) -> (owner, spender)
        nft = self.get_nft_not_owned_by_contract()
        o = self.onfts[nft[0]-1]
        owner = self.nft_owners[nft]
        new_spender = choice(list(self.accounts.values()))
        while new_spender == owner:
            new_spender = choice(list(self.accounts.values()))
        o.approve(new_spender, nft[1], from_=owner)
```

```
        self.approvals[nft] = (owner, new_spender)


    #tests that nfts have the correct owner
    @invariant
    def invariant_nft_owner(self):
        self.add_ether()
        for chain_id in self.chain_ids:
            for nft_id in self.minted_ids[chain_id]:
                assert self.onfts[chain_id-1].ownerOf(nft_id) ==
self.nft_owners[(chain_id, nft_id)]


    #tests that each account has the correct nft correct balance
    @invariant
    def invariant_balance(self):
        self.add_ether()
        for chain_id in self.chain_ids:
            for account in self.accounts.values():
                assert self.onfts[chain_id-1].balanceOf(account) ==
self.balances[(chain_id, account)]


    #tests that the approvals are correctly set
    @invariant
    def invariant_approvals(self):
        self.add_ether()
        for nft in self.approvals:
            assert self.onfts[nft[0]-1].getApproved(nft[1]) ==
self.approvals[nft][1]
```

C.3. BitLib fuzz tests

The fuzz tests focused on stress-testing the functions in BitLib and comparing the results against the expected values.

```
def test_msb_position(b: BitLibMock):
    for i in range(10**6):
        rndi = random_int(0, 2**256-1)
        b_msb = b.mostSignificantBitPosition(rndi)
```

```
        in_bin = '{0:256b}'.format(rndi)
        assert len(in_bin) == 256
        python_msb = 255 - in_bin.find('1') if rndi else 0
        assert python_msb == b_msb

def test_count_set_bits(b: BitLibMock, a: Dict[str, Address]):
    for i in range(10**6):
        rndi = random_int(0, 2**256-1)
        b_set_bits = b.countSetBits(rndi)
        in_bin = '{0:256b}'.format(rndi)
        assert len(in_bin) == 256
        python_set_bits = in_bin.count('1')
        assert python_set_bits == b_set_bits
```

ackee

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq