



Security Assessment

# Lumerin - Staking contract

CertiK Assessed on Aug 27th, 2024





Certik Assessed on Aug 27th, 2024

## Lumerin - Staking contract

The security assessment was prepared by Certik, the leader in Web3.0 security.

### Executive Summary

#### TYPES

Staking

#### ECOSYSTEM

Ethereum (ETH)

#### METHODS

Formal Verification, Manual Review, Static Analysis

#### LANGUAGE

Solidity

#### TIMELINE

Delivered on 08/27/2024

#### KEY COMPONENTS

N/A

#### CODEBASE

[Morpheus-Lumerin-Node](#)[View All in Codebase Page](#)

#### COMMITTS

[75269bd207913526bd7b4db0892307a39c0cb9b3](#)[096db06f548cd39de05447d3e4bacd6b0bf48b7f](#)[View All in Codebase Page](#)

### Vulnerability Summary



7

Total Findings

3

Resolved

0

Mitigated

0

Partially Resolved

4

Acknowledged

0

Declined

0 Critical

Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.

2 Major

1 Resolved, 1 Acknowledged



Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.

0 Medium

Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

4 Minor

2 Resolved, 2 Acknowledged



Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.

1 Informational

1 Acknowledged



Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

# TABLE OF CONTENTS | LUMERIN - STAKING CONTRACT

## I **Summary**

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

## I **Review Notes**

[Overview](#)

[Privileged Functions](#)

[External Dependencies](#)

## I **Findings**

[SMC-01 : Centralization Related Risks](#)

[SMC-03 : Reward May Be Locked Permanently](#)

[SMC-02 : Unchecked ERC-20 `transfer\(\)`/`transferFrom\(\)` Call](#)

[SMC-04 : Potential Revert Due to Unchecked Underflow](#)

[SMC-05 : Incompatibility With Deflationary Tokens](#)

[SMC-06 : Lack of Input Validation](#)

[SMC-07 : Lack of Emergency Withdrawal Mechanism](#)

## I **Formal Verification**

[Considered Functions And Scope](#)

[Verification Results](#)

## I **Appendix**

## I **Disclaimer**

# CODEBASE | LUMERIN - STAKING CONTRACT

## Repository



Morpheus-Lumerin-Node

## Commit

- 75269bd207913526bd7b4db0892307a39c0cb9b3
- 096db06f548cd39de05447d3e4bacd6b0bf48b7f

# AUDIT SCOPE | LUMERIN - STAKING CONTRACT

2 files audited ● 2 files without findings

ID	Repo	File	SHA256 Checksum
● SMC	Lumerin-protocol/Morpheus-Lumerin-Node	 StakingMasterChef.sol	e7698d2ae4c85fddf9794865868c0798451984fe7d174a00aa19347a95152eac
● SMM	Lumerin-protocol/Morpheus-Lumerin-Node	 StakingMasterChef.sol	1fc7bf63eba7934375fcd2769c9ea31e00b283faa9e98f58936cd256094695dd

## APPROACH & METHODS | LUMERIN - STAKING CONTRACT

This report has been prepared for Lumerin to discover issues and vulnerabilities in the source code of the Lumerin - Staking contract project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis, Formal Verification, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

# REVIEW NOTES | LUMERIN - STAKING CONTRACT

## Overview

The `StakingMasterChef` contract in the **Lumerin - Staking contract** project includes features such as pooling and rewards distribution, flexible lock durations, and staking capabilities.

## Privileged Functions

In the **Lumerin - Staking contract** project, the owner role is adopted to ensure the dynamic runtime updates of the project, which are specified in the finding `Centralization Related Risks` and in the `StakingMasterChef` contract.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community.

It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan.

Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project. To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

## External Dependencies

In **Lumerin - Staking contract**, the project relies on a few external contracts or addresses to fulfill the needs of its business logic.

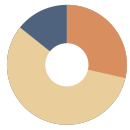
The following are third dependencies contracts used within the `StakingMasterChef` contract:

- `Ownable` : From Openzeppelin, this contract is foundational to owner permission implementation.
- `IERC20` : From OpenZeppelin, this contract is the interface of the ERC-20 standard as defined in the ERC.

The following are external addresses used within the `StakingMasterChef` contract:

- `_stakingToken` and `_rewardToken` : both are IERC20 tokens and will be provided in the constructor function.

## FINDINGS | LUMERIN - STAKING CONTRACT



7  
Total Findings

0  
Critical

2  
Major

0  
Medium

4  
Minor

1  
Informational

This report has been prepared to discover issues and vulnerabilities for Lumerin - Staking contract. Through this audit, we have uncovered 7 issues ranging from different severity levels. Utilizing the techniques of Static Analysis, Formal Verification & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
SMC-01	Centralization Related Risks	Centralization	Major	● Acknowledged
SMC-03	Reward May Be Locked Permanently	Logical Issue	Major	● Resolved
SMC-02	Unchecked ERC-20 <code>transfer()</code> / <code>transferFrom()</code> Call	Volatile Code	Minor	● Resolved
SMC-04	Potential Revert Due To Unchecked Underflow	Logical Issue	Minor	● Resolved
SMC-05	Incompatibility With Deflationary Tokens	Volatile Code	Minor	● Acknowledged
SMC-06	Lack Of Input Validation	Logical Issue	Minor	● Acknowledged
SMC-07	Lack Of Emergency Withdrawal Mechanism	Design Issue	Informational	● Acknowledged

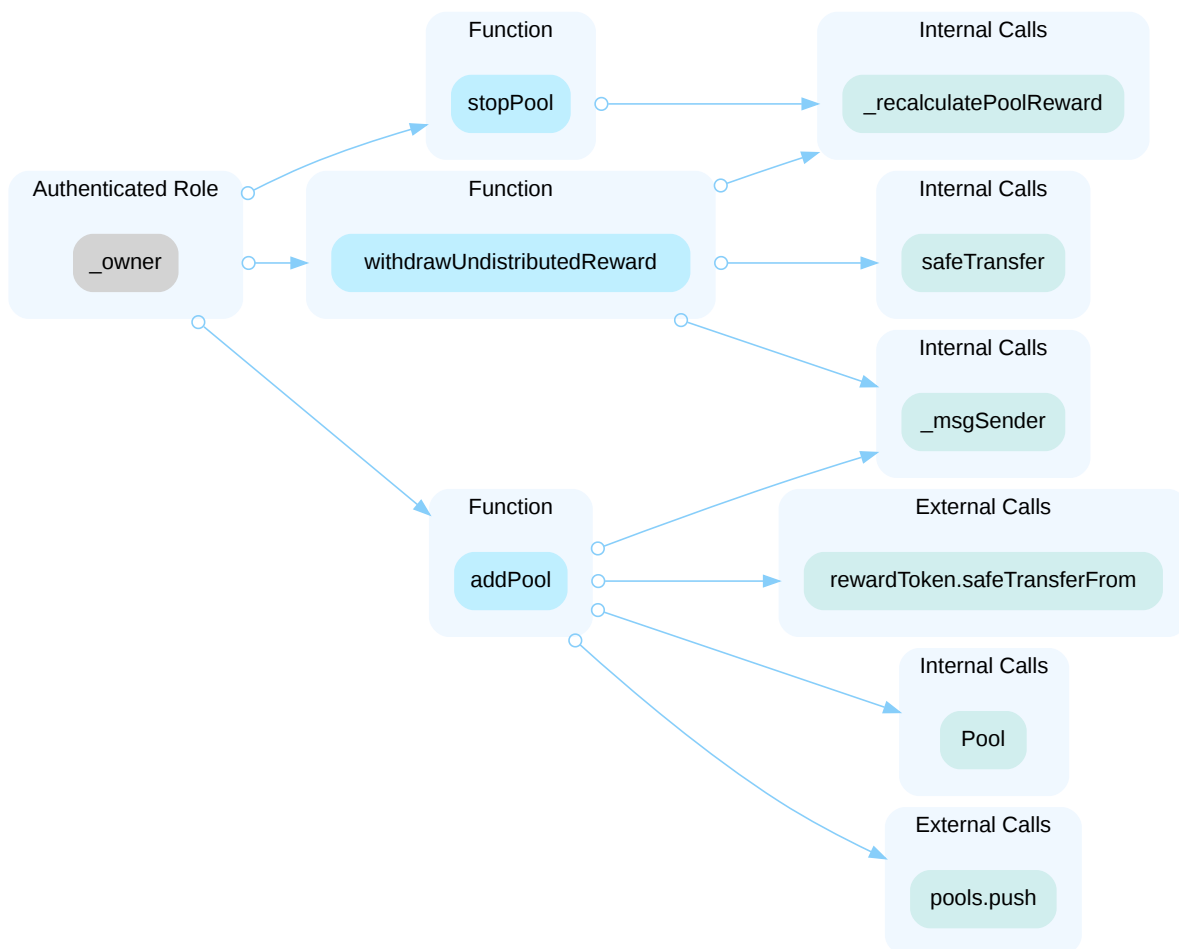


## SMC-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	● Major	StakingMasterChef.sol (08/16 - 75269b): 63, 98	● Acknowledged

### Description

In the contract `StakingMasterChef`, the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and add a new reward pool, stop the pool, withdraw undistributed reward from the specified pool.



Additionally, `StakingMasterChef` contract inherits the `Ownable` contract from OpenZeppelin, the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority, disrupt the normal access control.

## Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

### Short Term:

Timelock and Multi sign (2/3, 3/5) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;  
AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;  
AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

### Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;  
AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.  
AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

### Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.  
OR
- Remove the risky functionality.

## Alleviation

**[Lumerin Team, 08/22/2024]:**

Issue acknowledged. The team had other mitigation such as private keys stored in secure vault.

**[Certik, 08/23/2024]:**

It strongly encourages the project team periodically revisit the private key security management of all addresses related to centralized roles.

## SMC-03 | REWARD MAY BE LOCKED PERMANENTLY

Category	Severity	Location	Status
Logical Issue	● Major	StakingMasterChef.sol (08/16 - 75269b): 105~106	● Resolved

### Description

When a pool is added by the owner, `rewardToken` is transferred into the contract. The owner has the capability to halt the pool before its conclusion and retrieve any `rewardToken` that has not been distributed. However, if no users stake or if there is a period without staking activity, any `rewardToken` that remains undistributed at the end of the pool will be permanently locked within the contract.

### Proof of Concept

The POC using Foundry shows that after pool end there are some reward tokens left in the contract and cannot be withdrawn.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import {StakingMasterChef} from "../contracts/StakingMasterChef.sol";
import {LumerinToken} from "../contracts/LumerinToken.sol";
import {MorpheusToken} from "../contracts/MorpheusToken.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract StakingMasterChefTest is Test {
    StakingMasterChef public staking;
    LumerinToken public stakingToken;
    MorpheusToken public rewardToken;

    address public Alice = makeAddr("Alice");

    uint256 poolId;

    function setUp() public {
        stakingToken = new LumerinToken();
        rewardToken = new MorpheusToken();

        staking = new StakingMasterChef(
            IERC20(address(stakingToken)),
            IERC20(address(rewardToken))
        );

        // staking token distribution
        stakingToken.transfer(Alice, 100_000 * 10 ** 8);

        // adding pool
        vm.warp(1724045430); // 2024-08-19 13:30:30

        rewardToken.approve(address(staking), type(uint256).max);

        StakingMasterChef.Lock[] memory lockInfo = new StakingMasterChef.Lock[](3);

        lockInfo[0] = StakingMasterChef.Lock(7 days, 1 * 10 ** 12);
        lockInfo[1] = StakingMasterChef.Lock(30 days, 115 * 10 ** 12 / 100);
        lockInfo[2] = StakingMasterChef.Lock(180 days, 135 * 10 ** 12 / 100);

        poolId = _addPoolByOwner(lockInfo);
        console.log("add a new pool:", poolId);
    }

    // -----Test-----

    function test_POC1_LockedReward() public {
```

```
vm.warp(block.timestamp + 5 days + 120 days);
console.log("Alice stakes 50_000 LMR with 180 days locked");
vm.startPrank(Alice);
stakingToken.approve(address(staking), 50_000 * 10 ** 8);
staking.stake(poolId, 50_000 * 10 ** 8, 2);
vm.stopPrank();

console.log("Alice withdraws rewards after 180 days");
vm.warp(block.timestamp + 180 days);
vm.prank(Alice);
staking.withdrawReward(0,0);
console.log("Alice's rewards is %d", rewardToken.balanceOf(Alice));
console.log("RewardsInPool is %d", rewardToken.balanceOf(address(staking)));

vm.warp(block.timestamp + 1 days);
console.log("Alice unstakes");
vm.prank(Alice);
staking.unstake(0,0);
console.log("Alice's rewards is %d", rewardToken.balanceOf(Alice));
console.log("RewardsInPool is %d", rewardToken.balanceOf(address(staking)));
}

function _addPoolByOwner(StakingMasterChef.Lock[] memory locks) internal returns
(uint256){
    uint256 poolIdInternal = staking.addPool(block.timestamp + 5 days, 300 days,
5_000_000 * 10 ** 18, locks);
    return poolIdInternal;
}
}
```

Test result:

[illegible]

## Recommendation

It's recommended to revise the logic to ensure there would be no reward tokens locked in the contract.

## ■ Alleviation

[Lumerin Team, 08/22/2024]:

Issue acknowledged. The team resolved this issue in the commit hash: [096db06f548cd39de05447d3e4bacd6b0bf48b7f](#) by collecting the `undistributedReward` tokens and using `withdrawUndistributedReward()` function to withdraw the undistributed tokens in the contract.

```
function withdrawUndistributedReward(uint256 _poolId) external onlyOwner
poolExists(_poolId) {
    Pool storage pool = pools[_poolId];
    _recalculatePoolReward(pool);
    uint256 reward = pool.undistributedReward;
    pool.undistributedReward = 0;
    safeTransfer(_msgSender(), reward);
}
```

[CertiK, 08/26/2024]:

We recommend that the team manage the usage of `stakingToken` and `rewardToken` more carefully. In a scenario where

the amount of tokens staked significantly exceeds the amount of rewards, the calculation `rewardScaled / _pool.totalShares` might result in zero. Consequently, this would prevent any rewards from being distributed, causing them to remain trapped within the contract.



## SMC-02 | UNCHECKED ERC-20 `transfer()` / `transferFrom()` CALL

Category	Severity	Location	Status
Volatile Code	Minor	StakingMasterChef.sol (08/16 - 75269b): 84, 172, 209, 286	Resolved

### Description

The return values of the `transfer()` and `transferFrom()` calls in the smart contract are not checked. Some ERC-20 tokens' transfer functions return no values, while others return a bool value, they should be handled with care. If a function returns `false` instead of reverting upon failure, an unchecked failed transfer could be mistakenly considered successful in the contract.

Although `safeTransfer` function has been implemented in this contract, the function primarily handles transferring rewardToken without checking the result of the token transfer. Therefore, this differs from the `safeTransfer()` function logic in the `SafeERC20` library.

```
284     function safeTransfer(address _to, uint256 _amount) private {
285         uint256 rewardBalance = rewardToken.balanceOf(address(this));
286         rewardToken.transfer(_to, min(rewardBalance, _amount));
287     }
```

### Recommendation

It is advised to use the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if false is returned, making it compatible with all ERC-20 token implementations.

### Alleviation

[Lumerin Team, 08/22/2024]:

Issue Acknowledged. The team resolved this issue in the commit hash: [719804e18bf00239bbc90dd2c01f621ab413f6d8](#) by using `SafeERC20` library.

## SMC-04 | POTENTIAL REVERT DUE TO UNCHECKED UNDERFLOW

Category	Severity	Location	Status
Logical Issue	Minor	StakingMasterChef.sol (08/16 - 75269b): 105	Resolved

### Description

The `stopPool()` function can stop the pool, but if `oldEndTime` is less than `block.timestamp`, indicating that the pool ends before it stops, it will revert due to the compiler's overflow/underflow protection.

This is due to the use of subtraction in the expression `((oldEndTime - block.timestamp) * pool.rewardPerSecondScaled) / PRECISION` that doesn't account for underflows.

```
function stopPool(uint256 _poolId) external onlyOwner poolExists(_poolId) {  
    ...  
    uint256 undistributedReward = ((oldEndTime - block.timestamp) *  
pool.rewardPerSecondScaled) / PRECISION;  
    safeTransfer(_msgSender(), undistributedReward);  
}
```

### Recommendation

It is recommended to implement a check to ensure that `oldEndTime` is always greater than or equal to `block.timestamp`, one `require` statement could be used for this purpose.

### Alleviation

[Lumerin Team, 08/22/2024]:

Issue acknowledged. The team resolved this issue in the commit hash: [096db06f548cd39de05447d3e4bacd6b0bf48b7f](#) by adding check between `endTime` and `block.timestamp`, and collecting all undistributed tokens to `undistributedReward`.

## SMC-05 | INCOMPATIBILITY WITH DEFLATIONARY TOKENS

Category	Severity	Location	Status
Volatile Code	● Minor	StakingMasterChef.sol (08/16 - 75269b): 84, 172	● Acknowledged

### Description

When transferring standard ERC20 deflationary tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user stakes 100 deflationary tokens (with a 10% transaction fee) in a MasterChef contract, only 90 tokens actually arrive in the contract. However, the user can still withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Reference: <https://thoreum-finance.medium.com/what-exploit-happened-today-for-gocerberus-and-garuda-also-for-lokum-ybear-piggy-caramelswap-3943ee23a39f>

### Recommendation

We recommend regulating the set of pool tokens supported and adding necessary mitigation mechanisms to keep track of accurate balances, if there is a need to support deflationary tokens.

### Alleviation

[Lumerin Team, 08/22/2024]:

Issue acknowledged. The current implementation has no need to support deflationary tokens.

## SMC-06 | LACK OF INPUT VALIDATION

Category	Severity	Location	Status
Logical Issue	● Minor	StakingMasterChef.sol (08/16 - 75269b): 52, 63~68	● Acknowledged

### Description

The `constructor()` and `addPool()` functions are missing input validation for parameters `_stakingToken`, `_rewardToken`, `_startTime`, and `_lockDurations`. Unchecked input variables may lead to errors in zero address, time limit and lock duration.

```
constructor(IERC20 _stakingToken, IERC20 _rewardToken) Ownable(_msgSender()) {
    stakingToken = _stakingToken;
    rewardToken = _rewardToken;
}
```

```
function addPool(
    uint256 _startTime,
    uint256 _duration,
    uint256 _totalReward,
    Lock[] memory _lockDurations
) external onlyOwner returns (uint256) {
    ...
}
```

### Recommendation

It is recommended to add input parameter checks before function execution.

### Alleviation

[Lumerin Team, 08/22/2024]:

Issue acknowledged. The team decided not to change current codebase and accepted the risk from the Owner perspective at this time, including setting up the Pools.

## SMC-07 | LACK OF EMERGENCY WITHDRAWAL MECHANISM

Category	Severity	Location	Status
Design Issue	● Informational	StakingMasterChef.sol (08/16 - 75269b): 8	● Acknowledged

### Description

The `StakingMasterChef` contract lacks a feature for emergency withdrawals that exclude rewards, a functionality that is provided by Sushi's `MasterChef`. This current design choice could potentially prevent users from accessing their staked assets during emergencies.

### Recommendation

We would like to understand whether this limitation in the `StakingMasterChef` contract is intended.

### Alleviation

[Lumerin Team, 08/22/2024]:

Issue acknowledged. The team decided not to change the current codebase, because this is the intended design for staking, and locked funds will be locked for the duration of the stake.

# FORMAL VERIFICATION | LUMERIN - STAKING CONTRACT

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

## Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

### Verification of Security Properties

We verified the following security properties of public interfaces of important contracts of the project.

Property Name	Title
09-stakingmasterchef-withdrawreward-check	Verification of <code>withdrawReward</code> Function
08-stakingmasterchef-unstake-check	Verification of <code>unstake</code> Function
07-stakingmasterchef-stake-success	Verification of <code>stake</code> Function
06-stakingmasterchef-stake-fail	Check the <code>stake</code> Function Has Proper Input Validations
05-stakingmasterchef-_recalculatePoolReward-check	Verification of <code>_recalculatePoolReward</code> Function
04-stakingmasterchef-stopPool-success	Verification of <code>stopPool</code> Function
03-stakingmasterchef-stopPool-fail	Check the <code>stopPool</code> Function Has Proper Input Validations
02-stakingmasterchef-addPool-success	Verification of <code>addPool</code> Function
01-stakingmasterchef-addPool-fail	Check the <code>addPool</code> Function Has Proper Input Validations

### Verification of Standard Ownable Properties

We verified *partial* properties of the public interfaces of those token contracts that implement the Ownable interface. This involves:

- function `owner` that returns the current owner,
- functions `renounceOwnership` that removes ownership,
- function `transferOwnership` that transfers the ownership to a new owner.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
ownable-owner-succeed-normal	<code>owner</code> Always Succeeds
ownable-renounceownership-correct	Ownership is Removed
ownable-transferownership-correct	Ownership is Transferred
ownable-renounce-ownership-is-permanent	Once Renounced, Ownership Cannot be Regained

## Verification Results

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful. There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

### Detailed Results For Contract `StakingMasterChef` (smart-contracts/contracts/StakingMasterChef.sol) In Commit `75269bd207913526bd7b4db0892307a39c0cb9b3`

#### Verification of Security Properties

Detailed Results for Function `withdrawReward`

Property Name	Final Result	Remarks
09-stakingmasterchef-withdrawreward-check	● True*	<code>rewardToken</code> is a valid <code>IERC20</code> implementation.

Detailed Results for Function `unstake`

Property Name	Final Result	Remarks
08-stakingmasterchef-unstake-check	● True*	<code>stakingToken</code> and <code>rewardToken</code> are valid <code>IERC20</code> implementations.

Detailed Results for Function `stake`

Property Name	Final Result	Remarks
07-stakingmasterchef-stake-success	● True*	<code>stakingToken</code> is a valid <code>IERC20</code> implementation.
06-stakingmasterchef-stake-fail	● True	

Detailed Results for Function `_recalculatePoolReward`

Property Name	Final Result	Remarks
05-stakingmasterchef-_recalculatepoolreward-check	● True	

Detailed Results for Function `stopPool`

Property Name	Final Result	Remarks
04-stakingmasterchef-stoppool-success	● True*	<code>rewardToken</code> is a valid <code>IERC20</code> implementation.
03-stakingmasterchef-stoppool-fail	● True	

Detailed Results for Function `addPool`

Property Name	Final Result	Remarks
02-stakingmasterchef-addpool-success	● True*	<code>rewardToken</code> is a valid <code>IERC20</code> implementation.
01-stakingmasterchef-addpool-fail	● False	

**Detailed Results For Contract StakingMasterChef (smart-contracts/contracts/StakingMasterChef.sol) In Commit 096db06f548cd39de05447d3e4bacd6b0bf48b7f****Verification of Standard Ownable Properties**Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	



Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● Inconclusive	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

### Detailed Results For Contract StakingMasterChef (smart-contracts/contracts/StakingMasterChef.sol) In Commit 75269bd207913526bd7b4db0892307a39c0cb9b3

#### Verification of Standard Ownable Properties

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● Inconclusive	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

## APPENDIX | LUMERIN - STAKING CONTRACT

### Finding Categories

Categories	Description
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

### Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

### Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

### Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean

connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

## Description of the Analyzed Ownable Properties

### Properties related to function `owner`

#### ownable-owner-succeed-normal

Function `owner` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

### Properties related to function `renounceOwnership`

#### ownable-renounce-ownership-is-permanent

The contract must prohibit regaining of ownership once it has been renounced.

Specification:

```
constraint \old(owner()) == address(0) ==> owner() == address(0);
```

#### ownable-renounceownership-correct

Invocations of `renounceOwnership()` must set ownership to `address(0)`.

Specification:

```
ensures this.owner() == address(0);
```

Properties related to function `transferOwnership`

#### ownable-transferownership-correct

Invocations of `transferOwnership(newOwner)` must transfer the ownership to the `newOwner`.

Specification:

```
ensures this.owner() == newOwner;
```

### Description of the Analyzed Project-Specific Properties

Properties related to function `withdrawReward`

#### 09-stakingmasterchef-withdrawreward-check

Check the `withdrawReward` is working as expected.

`rewardToken` is a valid `IERC20` implementation.

Specification:

```
/*@
  name "09-StakingMasterChef-withdrawReward-check";
  reverts_when _stakeId >= poolUserStakes[_poolId][_msgSender()].length;
  also
  ensures poolUserStakes[_poolId][_msgSender()][_stakeId].rewardDebt ==
poolUserStakes[_poolId][_msgSender()][_stakeId].shareAmount *
pools[_poolId].accRewardPerShareScaled / PRECISION;
  ensures
    rewardToken.balanceOf(_msgSender()) ==
    \old(rewardToken.balanceOf(_msgSender()))
    + min(
      poolUserStakes[_poolId][_msgSender()][_stakeId].shareAmount
      * pools[_poolId].accRewardPerShareScaled
      / PRECISION
      - \old(poolUserStakes[_poolId][_msgSender()][_stakeId].rewardDebt),
      \old(rewardToken.balanceOf(_msgSender())));
*/
function withdrawReward(uint256 _poolId, uint256 _stakeId) external {
```

Properties related to function `unstake`

#### 08-stakingmasterchef-unstake-check

Check the `unstake` is working as expected.

`stakingToken` and `rewardToken` are valid `IERC20` implementations.

Specification:

```
/*@
  name "08-StakingMasterChef-unstake-check";
  reverts_when _stakeId >= poolUserStakes[_poolId][_msgSender()].length;
  also
  ensures pools[_poolId].totalShares == \old(pools[_poolId].totalShares -
poolUserStakes[_poolId][_msgSender()][_stakeId].shareAmount);
  ensures poolUserStakes[_poolId][_msgSender()][_stakeId].rewardDebt == 0;
  ensures poolUserStakes[_poolId][_msgSender()][_stakeId].stakeAmount == 0;
  ensures poolUserStakes[_poolId][_msgSender()][_stakeId].shareAmount == 0;
  ensures poolUserStakes[_poolId][_msgSender()][_stakeId].lockEndsAt == 0;
  also
  requires address(stakingToken) != address(rewardToken);
  ensures stakingToken.balanceOf(_msgSender()) ==
\old(stakingToken.balanceOf(_msgSender()) + poolUserStakes[_poolId][_msgSender()]
[_stakeId].stakeAmount);
*/
function unstake(uint256 _poolId, uint256 _stakeId) external {
```

Properties related to function `stake`

#### 06-stakingmasterchef-stake-fail

`stake` function should reverts if the input validations are missing.

Specification:

```
/*@
  group "05";
  name "06-StakingMasterChef-stake-fail";
  reverts_when _poolId >= pools.length;
  reverts_when _lockId >= pools[_poolId].locks.length;
  reverts_when block.timestamp < pools[_poolId].startTime;
  reverts_when block.timestamp >= pools[_poolId].endTime;
  reverts_when block.timestamp + pools[_poolId].locks[_lockId].durationSeconds >
pools[_poolId].endTime;
*/
function stake(uint256 _poolId, uint256 _amount, uint8 _lockId) external
poolExists(_poolId) returns (uint256) {
```

#### 07-stakingmasterchef-stake-success

Check the `stake` is working as expected.

`stakingToken` is a valid `IERC20` implementation.

Specification:

```
/*@
  group "06";
  name "07-StakingMasterChef-stake-success";
  ensures pools[_poolId].totalShares == \old(pools[_poolId].totalShares) + (_amount
* pools[_poolId].locks[_lockId].multiplierScaled) / PRECISION;
  ensures poolUserStakes[_poolId][_msgSender()][poolUserStakes[_poolId]
[_msgSender()].length-1].stakeAmount == _amount;
  ensures poolUserStakes[_poolId][_msgSender()][poolUserStakes[_poolId]
[_msgSender()].length-1].shareAmount == (_amount *
pools[_poolId].locks[_lockId].multiplierScaled) / PRECISION;
  ensures poolUserStakes[_poolId][_msgSender()][poolUserStakes[_poolId]
[_msgSender()].length-1].rewardDebt == (_amount *
pools[_poolId].locks[_lockId].multiplierScaled) / PRECISION *
pools[_poolId].accRewardPerShareScaled / PRECISION;
  ensures poolUserStakes[_poolId][_msgSender()][poolUserStakes[_poolId]
[_msgSender()].length-1].lockEndsAt == block.timestamp +
pools[_poolId].locks[_lockId].durationSeconds;
  also
  ensures stakingToken.balanceOf(address(this)) ==
\old(stakingToken.balanceOf(address(this))) + _amount;
*/
function stake(uint256 _poolId, uint256 _amount, uint8 _lockId) external
poolExists(_poolId) returns (uint256) {
```

Properties related to function `_recalculatePoolReward`

#### 05-stakingmasterchef-\_recalculatepoolreward-check

Check the `_recalculatePoolReward` is working as expected.

Specification:

```

/*@
  name "05-StakingMasterChef-_recalculatePoolReward-check";
  requires block.timestamp < _pool.endTime;
  requires block.timestamp > _pool.lastRewardTime;
  ensures _pool.lastRewardTime == block.timestamp;
  also
  requires block.timestamp >= _pool.endTime;
  requires _pool.endTime > _pool.lastRewardTime;
  ensures _pool.lastRewardTime == _pool.endTime;
  also
  ensures _pool.totalShares > 0 && (block.timestamp < _pool.endTime &&
block.timestamp > _pool.lastRewardTime) ==> _pool.accRewardPerShareScaled >
\old(_pool.accRewardPerShareScaled);
  ensures _pool.totalShares > 0 && (block.timestamp >= _pool.endTime &&
_pool.endTime > _pool.lastRewardTime) ==> _pool.accRewardPerShareScaled >
\old(_pool.accRewardPerShareScaled);
*/
function _recalculatePoolReward(Pool storage _pool) private {

```

#### Properties related to function `stopPool`

##### 03-stakingmasterchef-stoppool-fail

`stopPool` function should revert if the input validations are missing.

Specification:

```

/*@
  group "03";
  name "03-StakingMasterChef-stopPool-fail";
  reverts_when _poolId >= pools.length;
  reverts_when block.timestamp > pools[_poolId].endTime;
*/
function stopPool(uint256 _poolId) external onlyOwner poolExists(_poolId) {

```

##### 04-stakingmasterchef-stoppool-success

Check the `stopPool` is working as expected.

`rewardToken` is a valid `IERC20` implementation.

Specification:

```

/*@
  group "04";
  name "04-StakingMasterChef-stopPool-success";
  ensures pools[_poolId].endTime == block.timestamp;
  also
  requires block.timestamp <= pools[_poolId].endTime;
  ensures
    rewardToken.balanceOf(_msgSender()) ==
    \old(rewardToken.balanceOf(_msgSender()))
    + min(
      \old(pools[_poolId].endTime - block.timestamp)
      * pools[_poolId].rewardPerSecondScaled
      / PRECISION,
      \old(rewardToken.balanceOf(address(this)))
    );
*/
function stopPool(uint256 _poolId) external onlyOwner poolExists(_poolId) {

```

#### Properties related to function `addPool`

##### 01-stakingmasterchef-addpool-fail

`addPool` function should revert if the input validations are missing.

Specification:

```

/*@
  group "101";
  name "01-StakingMasterChef-addPool-fail";
  reverts_when _startTime < block.timestamp;
  reverts_when _totalReward == 0;
  reverts_when _lockDurations.length == 0;
*/
function addPool(
  uint256 _startTime,
  uint256 _duration,
  uint256 _totalReward,
  Lock[] memory _lockDurations
) external onlyOwner returns (uint256) {

```

##### 02-stakingmasterchef-addpool-success

Check the `addPool` is working as expected.

`rewardToken` is a valid `IERC20` implementation.

Specification:



```
/*@
    group "102";
    name "02-StakingMasterChef-addPool-success";
    ensures pools.length == \old(pools.length) + 1;
    ensures pools[pools.length-1].startTime == _startTime;
    ensures pools[pools.length-1].lastRewardTime == _startTime;
    ensures pools[pools.length-1].endTime == _startTime + _duration;
    ensures pools[pools.length-1].rewardPerSecondScaled == (_totalReward * PRECISION)
/ _duration;
    ensures pools[pools.length-1].accRewardPerShareScaled == 0;
    ensures pools[pools.length-1].totalShares == 0;
    also
    ensures rewardToken.balanceOf(address(this)) ==
\old(rewardToken.balanceOf(address(this))) + _totalReward;
*/
function addPool(
    uint256 _startTime,
    uint256 _duration,
    uint256 _totalReward,
    Lock[] memory _lockDurations
) external onlyOwner returns (uint256) {
```

## DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

# CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

