certora

Security Assessment
Final Report

Silo Leverage

July 2025

Prepared for Silo

Table of content

certora

Silo

Project Summary

Project Scope

Project Name	Repository (link)	Initial Commit Hash	Final Commit Hash	Platform
Silo	https://github.com/silo-finance/silo-contracts-v2	af2ba96	66a08a3	EVM

Project Overview

This document describes the findings of the manual for the **Silo Leverage Module**. The work was undertaken from **7th of July** to **11th of July**

The following contract list is included in our scope:

```
silo-core/contracts/leverage/modules/GeneralSwapModule.sol
silo-core/contracts/leverage/modules/LeverageTxState.sol
silo-core/contracts/leverage/modules/RevenueModule.sol
silo-core/contracts/leverage/LeverageUsingSiloFlashloan.sol
silo-core/contracts/leverage/LeverageUsingSiloFlashloanWithGeneralSwap.sol
```

The list of additional contracts after the fix:

```
silo-core/contracts/leverage/modules/RescueModule.sol
silo-core/contracts/leverage/LeverageRouter.sol
```

Protocol Overview

The Silo leverage module provides users with the ability to amplify their exposure to assets within Silo's lending markets through automated flashloan-based leverage operations. Users can open leveraged positions by borrowing additional capital, converting it to their desired asset, and using the combined position as collateral for the borrowed funds. Users can also close their leveraged positions by unwinding the collateral back to the borrowed asset and settling their debt. The module integrates with external DEX aggregators to facilitate efficient asset swaps.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	5	5	1
Low	13	13	4
Informational	7	7	5
Total	25	25	10

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
M-01	Fee-on-Transfer tokens corrupt Silo accounting	Medium	Acknowledged
M-02	Opening a position at or near maximum leverage will revert due to leverageFee	Medium	Acknowledged
M-03	Leverage does not enforce any checks on the validity of the “silo” targets	Medium	Acknowledged
M-04	Missing cross-contract reentrancy protection in leverageUsingSiloFlashloan contract	Medium	Acknowledged
M-05	Protected collateral can’t be closed with permit	Medium	Fixed
L-01	General swap sender should only be the leverage contract	Low	Acknowledged
L-02	Missing helper functions to show users current and maximum leverage	Low	Acknowledged
L-03	GeneralSwapModule donates any existing contract funds to next user	Low	Acknowledged
L-04	Zero amount check on swap is ignored if there are funds in the contract	Low	Acknowledged
L-05	calculateDebtReceiveApproval return value is incorrect if flashloan provider is not Silo	Low	Fixed

L-06	openLeveragePosition event does not emit leverage of the position	Low	Acknowledged
L-07	closeLeveragePosition forces full closure of the position, deleveraging partially is not possible	Low	Acknowledged
L-08	rescueTokens has no access control restriction	Low	Fixed
L-09	maxApprovals should be reset at the end of the transaction	Low	Fixed
L-10	try/catch around executePermit is flawed	Low	Acknowledged
L-11	Missing slippage protection can cause unexpected losses	Low	Acknowledged
L-12	RevenueModule is not protected by reentrancy	Low	Fixed
L-13	GeneralSwap unprotected arbitrary call.	Low	Acknowledged
I-01	Leverage does not support the same asset debt	Info	Fixed
I-02	Double call to onFlashLoan	Info	Fixed
I-03	No rescue function for native token	Info	Fixed
I-04	Reset transient variables at the end of the function flow	Info	Fixed
I-05	Rebase tokens are incompatible with the module	Info	Acknowledged
I-06	Leverage fee cannot be set to maximum value	Info	Fixed
I-07	CREATE2-based cloning is incompatible with zkSync Era	Info	Acknowledged

Medium Severity Issues

M-01. Fee-on-Transfer tokens corrupt the Silo accounting

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description:

The leverage module assumes that the token amounts remain constants during transfers, which is not the case with fee-on-transfer tokens. As a result, opening a leveraged position with FOT tokens creates and/or widens a gap between recorded asset holdings and actual token holdings.

```rust
// LeverageUsingSiloFlashloan.sol:191-199
uint256 totalDeposit = depositArgs.amount + collateralAmountAfterSwap;

// Fee is taken on totalDeposit = user deposit amount + collateral amount after swap
uint256 feeForLeverage = calculateLeverageFee(totalDeposit);

totalDeposit -= feeForLeverage;
address collateralAsset = depositArgs.silo.asset();
_deposit({_depositArgs: depositArgs, _totalDeposit: totalDeposit, _asset: collateralAsset});
```

If for example a position is opened for 100 FOT tokens with a 2% fee, then the depositArgs.amount = 100, but the actual amount of tokens transferred will be 98. This creates a discrepancy between the actual token balance (98) and the recorded asset amount (100). This will widen with every FOT transaction and lead to solvency issues.

Recommendations: Block FOT tokens from using the leverage module.

Customer's response: Fee-on-Transfer tokens are not supported in Silo V2 protocol.

M-02. Opening a position at or near maximum leverage will revert due to leverageFee

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description:

If a user tries to leverage a position at or near the theoretical maximum leverage based on LTV and position, this will almost always revert due to impact of the leverageFee.

If we have a user who wants to open a position at max leverage in a silo with 80% LTV:

- Deposit: 1000
- FlashLoan: 4000
- Expected total collateral: 5000
- Expected LTV: 4000/5000 = 80%

The logic will correctly execute but when it hits _openLeverage, the leverageFee is deducted from the total collateral and the remainder is actually deposited. Assuming 1% fee:

- totalDeposit: 5000 - 50 = 4950

This is problematic since in the subsequent borrow() function, the maximum LTV is checked and the function will revert:

- 4000/4950 = 81% > 80 => revert.

The higher the leverageFee, the more transactions close to the maximum leverage will revert.

Recommendations: Either provide helper functions that inform the user how much leverage he can take, or rework the leverage function to automatically take the maximum leverage possible if the requested leverage exceeds LTV due to fees.

Customer's response: In our case, frontend will be responsible for calculating proper input so the user can perform leverage without reverting.

M-03. Leverage does not enforce any checks on the validity of the “silo” targets

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan.sol	Status: Acknowledged	

Description: Currently there is no mechanism in place to have any validation on the targets for leverage, this might be used in some sophisticated attacks where we trick the leverage contract to call malicious custom contracts.

An example from openLeveragePosition

```
94    function openLeveragePosition(
95        FlashArgs calldata _flashArgs,
96        bytes calldata _swapArgs,
97        DepositArgs calldata _depositArgs
98    )
99        public
100       payable
101       virtual
102       whenNotPaused
103       nonReentrant
104       setupTxState(_depositArgs.silo, LeverageAction.Open, _flashArgs.flashloanTarget)
105   {
```

No validation on _depositArgs.

Recommendations: Implement a defences mechanism similar to the one that can be found in partialLiquidation.sol, having a factory clone leverage and tying down to only a single silo pair. This should overall improve separation (one of the core uses for Silo), and would mitigate a lot of attack vectors.

Note: Our analysis yields that this can mitigate most potential attack vectors.

Customer’s response: Acknowledged. Silo registry from SiloFactory is not trusted, because Silo implementation is specified by deployer. “silo“ targets can't be validated, it is a user’s input.

M-04. Missing cross-contract reentrancy protection in leverageUsingSiloFlashloan contract

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description:

The LeverageUsingSiloFlashloan contract implements only local reentrancy protection via TransientReentrancy.nonReentrant, but lacks any form of cross-contract reentrancy protection.

This is a significant issue since such protection (siloConfig.turnOnReentrancyProtection()) is used in all Silo operations involving external calls. This is especially worrisome since the leverage utilises an arbitrary external call, allowing anyone to re-enter the silo protocol at almost any entry point.

It's also worth noting that the PartialLiquidation contract, the only other contract that uses TransientReentrancy protection, does properly implement the additional cross-contract protection.

Recommendations: Apply both local and cross-contract reentrancy protection.

Customer's response: Acknowledged. It is a design choice. Leverage is a periphery contract. Liquidation is part of Silo itself. We don't couple leverage and silo the way we couple liquidation. Changes to Silo core contract should be avoided to limit potential mistakes and additional work.

M-05. Protected collateral can't be closed with permit

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Fixed	

Description: The Silo leverage system allows users to close leveraged positions using a permit-based approval for collateral withdrawal. However, the implementation only works for borrowable collateral and silently fails for protected collateral, potentially leaving users unable to close positions as expected.

```
JavaScript
   function closeLeveragePositionPermit(
        address _msgSender,
        bytes calldata _swapArgs,
        CloseLeverageArgs calldata _closeArgs,
        Permit calldata _withdrawAllowance
    )
        external
        virtual
    {
        _executePermit(_msgSender, _withdrawAllowance,
address(_closeArgs.siloWithCollateral));

        closeLeveragePosition(_msgSender, _swapArgs, _closeArgs);
    }
```

The permit is always executed on the Silo contract, but the actual withdrawal logic is:

```
JavaScript
uint256 withdrawnDeposit = closeArgs.siloWithCollateral.redeem({
    _shares: sharesToRedeem,
    _receiver: address(this),
    _owner: _txMsgSender,
```

```
    _collateralType: closeArgs.collateralType
});
```

Inside the Silo contract, the withdrawal path is:

```JavaScript
(assets, shares) = SiloERC4626Lib.withdraw(
    depositConfig.token,
    _args.collateralType == ISilo.CollateralType.Collateral
        ? depositConfig.collateralShareToken
        : depositConfig.protectedShareToken,
    _args
);
```

And the actual approval check is here:

```JavaScript
IShareToken(_shareToken).burn(_args.owner, _args.spender, shares);
```

So for CollateralType.Collateral, the Silo contract itself is the share token, so the permit works. But for CollateralType.Protected, the share token is a separate contract, and the permit on the Silo contract does not grant approval to the leverage contract to burn the user's protected share tokens. So the transaction will revert if the user has not already approved the leverage contract on the protected share token.

Recommendations: Update the leverage contract to detect the collateral type and, for protected collateral, execute the permit on the protected share token contract rather than the Silo contract.

Customer's response: Fixed in PR#1511.

Low Severity Issues

L-01. General swap sender should only be the leverage contract

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: GeneralSwapModule.sol	Status: **Acknowledged**	

Description: Currently, every contract can call fillQuote, which is a dangerous external call that really should never be called by any address except a leverage contract.

Recommendations: Enforce that each leverage contract has a clone of the general swap, and only that leverage contract can call fillQuote with a modifier such as onlyLeverage.

Customer's response: Acknowledged. It is a design choice. We are going to clone LeverageUsingSiloFlashloanWithGeneralSwap and it will use GeneralSwapModule deployed by constructor. All clones will use the same GeneralSwapModule so it cannot be protected.

L-02. Missing helper functions to show users maximum leverage

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: SiloLensLib	Status: Acknowledged	

Description: For any protocol (or user) integrating with the leverage module, it is imperative that they are aware of the maximum effective leverage they can take for any position, in order to efficiently use the functionality.

There are helper functions for LTV and position, but these are insufficient for the maximum leverage since it a dynamic variable which is depended on:

- Maximum LTV of the Silo, which is unique per Silo
- Existing collateral and debt within the Silo pair
- Collateral deposited during the openLeveragePosition call
- Flash and leverage fees

This leads to a significant degradation of the user experience since user will have to make rough guesses of what leverage they can have and have a meaningful amount of leverage calls revert.

Recommendations: Add helper functions to show the current and maximum leverage for any given position.

Customer's response: Acknowledged. It is calculated on UI.

L-03. GeneralSwapModule donates any existing contract funds to next user

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: GeneralSwapModule	Status: Acknowledged	

Description:

The GeneralSwapModule contract uses balanceOf to determine the amount of funds to send to the user.

```
ftrace | funcSig
function _transferBalanceToSender(address _token⬆) internal virtual returns (uint256 balance) {
    balance = IERC20(_token⬆).balanceOf(address(this));

    if (balance != 0) {
        IERC20(_token⬆).safeTransfer(msg.sender, balance);
    }
}
```

This is not recommended since it means that any existing funds on the contract (failed previous transactions, dust due to rounding, donations, etc..) will be donated to the next caller.

Recommendations: Use the data returned from the arbitrary call to determine the amount to be sent.

Customer's response: Acknowledged. It is a design choice.

L-04. Zero amount check on swap is ignored if there are funds in the contract

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: GeneralSwapModule	Status: Acknowledged	

Description:

The fillQuote check on zero amount is based on the contract balance.

```
    amountOut = _transferBalanceToSender(_swapArgs.buyToken);
    if (amountOut == 0) revert ZeroAmountOut();

    _transferBalanceToSender(_swapArgs.sellToken);
}
```

This is not recommended since any existing funds on the contract (1 wei donation of buyToken) would cause a malicious swap with no returned buyTokens to not revert, thereby bypassing a critical verification step.

Recommendations: No check should be depended on a variable that is so easily manipulated.

Customer's response: Acknowledged.

L-05. calculateDebtReceiveApproval return value is incorrect if flashloan provider is not Silo

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LeverageUsingSiloFlashloan	Status: Fix	

Description:

The `calculateDebtReceiveApproval` function returns the required amount to approve for the `leverageOpenPosition` function to succeed.

However, it takes into account a `flashfee` which is unique to Silo. If the user/protocol decides to use a different flashloan provider, the amount returned will be incorrect:

- If other.flashfee > silo.flashfee ⇒ approval will be insufficient
- if other.flashee < silo.flashfee ⇒ approval will be too large

Recommendations: The function should clarify in natspec and naming that it should only be used when the SILO flashloan functionality is used.

Customer's response: Fixed in PR#1481.

L-06. openLeveragePosition event does not emit leverage of the position

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description: The openLeveragePosition function emits the following event:

```rust
        emit OpenLeverage({
            borrower: _txMsgSender,
            borrowerDeposit: depositArgs.amount,
            swapAmountOut: collateralAmountAfterSwap,
            flashloanAmount: _flashloanAmount,
            totalDeposit: totalDeposit,
            totalBorrow: _flashloanAmount + _flashloanFee,
            leverageFee: feeForLeverage,
            flashloanFee: _flashloanFee
        });
```

This does not tell the user the actual leverage of the position. Which is critical information for managing the risk of the position.

Recommendations: Emit the precise leverage of the position.

Customer's response: Acknowledged.

L-07. closeLeveragePosition forces full closure of the position, deleveraging partially is not possible

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description:

The closeLeveragePosition forces the user to not only remove all leverage from the position but also to fully close the position.

Partial deleveraging is a standard feature in any leveraging protocol. As is, users would be forced to fully close their position and then reopen it at the leverage that they prefer. Thereby suffering from highly increased fees.

Recommendations: The open and close LeveragePosition functions should be refactored into one LeveragePosition function where the user can specify the degree of leverage he desires.

Customer's response: Acknowledged.

L-08. rescueTokens has no access control restriction

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: RevenueModule.sol	Status: Fixed	

Description:

The rescueTokens functions send any balance of ERC20 tokens on the contract to a specified revenueReceiver. This is admin functionality and should not be callable by anyone.

```
86    function rescueTokens(IERC20[] calldata _tokens⬆) external {
87        for (uint256 i; i < _tokens⬆.length; i++) {
88            rescueTokens(_tokens⬆[i]);
89        }
90    }
```

Recommendations: Add onlyOwner modifier.

Customer's response: Fixed in PR#1480. Tokens receiver is the user for whom the contract was cloned. Only this user can execute rescue function.

L-09. maxApprovals should be reset at the end of the transaction

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Fixed	

Description:

Currently the contract logic sets the approval of a spender to uint256.max if the given allowance is not sufficient.

Using such huge allowances is a very risky design decision since it opens a clear attack vector if the contract is compromised in any way. This should be avoided whenever possible.

Recommendations: reset the allowances back to 0 at the end of each transaction. Alternatively, implementing the recommendations of M-03 would alleviate the need to reset allowances.

Customer's response: Fixed in PR#1474. Max approvals were removed. Leverage contract approves exact amounts.

L-10. try/catch around executePermit is flawed

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description: A try/catch has been implemented around the _executePermit in order to mitigate the frontrunning DOS attack. However, allowing the function to continue regardless of the state of the permit means that any legitimately incorrect permit is ignored and the transaction will only fail on the actual `safeTransfer` call, wasting quite a bit of gas for no reason.

Recommendations:

```rust
    function _executePermit(Permit memory _permit, address _token) internal virtual {
        try IERC20Permit(_token).permit({
            owner: msg.sender,
            spender: address(this),
            value: _permit.value,
            deadline: _permit.deadline,
            v: _permit.v,
            r: _permit.r,
            s: _permit.s
        }) {
            return;
        } catch {
            // Permit potentially got frontrun, continue if allowance is sufficient
            if(IERC20(_token).allowance(owner,spender) >= value {
                return;
            }
    }
```

Customer's response: Acknowledged. Design choice.

L-11. Missing slippage protection can cause unexpected losses

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: LeverageUsingSiloFlashloan	Status: Acknowledged	

Description:

There is presently no slippage check present to ensure that the swap did not lose an exorbitant amount of tokens due to slippage. As long as enough collateral has been returned to satisfy the LTV ratio, the function call will proceed.

This can cause situations of massive loss in cases where the leverage demanded was low.

Recommendations: While it can be argued that this the user's responsibility, it is advised to add in an explicit check to avoid inexperienced users shooting themselves in the foot.

Customer's response: Acknowledged. Design choice. Swap arguments should have a proper slippage to avoid any losses.

L-12. RevenueModule is not protected by reentrancy

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: RevenueModule.sol	Status: Fixed	

Description: RevenueModule exposes external methods but is not protected by a reentrancy guard. This can lead to a complicated combined attack where funds are siphoned mid-leverage.

```
86    function rescueTokens(IERC20[] calldata _tokens⬆) external {
87        for (uint256 i; i < _tokens⬆.length; i++) {
88            rescueTokens(_tokens⬆[i]);
89        }
90    }
```

Recommendations: Should add the nonReentrant modifier to those methods.

Customer's response: Fixed in PR#1470.

L-13. GeneralSwap unprotected arbitrary call

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: GeneralSwapModule.sol	Status: Acknowledged	

Description: GeneralSwap fillQuote has an external-facing method with unprotected arbitrary call.

```
24      function fillQuote(SwapArgs memory _swapArgs⬆, uint256 _maxApprovalAmount⬆)
25          external
26          virtual
27          returns (uint256 amountOut) //Onlyleverage
28      {
29          if (_swapArgs⬆.exchangeProxy == address(0)) revert ExchangeAddressZero();
30
31          // Approve token for spending by the exchange
32          _setMaxAllowance(IERC20(_swapArgs⬆.sellToken), _swapArgs⬆.allowanceTarget, _maxApprovalAmount⬆);
33
34          // Perform low-level call to any method and any smart contract provided by the caller.
35          // solhint-disable-next-line avoid-low-level-calls
36          // silo config.
37          (bool success, bytes memory data) = _swapArgs⬆.exchangeProxy.call(_swapArgs⬆.swapCallData);
38          if (!success) RevertLib.revertBytes(data, SwapCallFailed.selector);
```

Recommendations:

1. Have a whitelist and a blacklist of valid addresses/calls that can be called from this address.

Customer's response: Acknowledged. Design choice. This contract should not have any leftovers.

Informational Issues

I-01. Leverage does not support the same asset debt

Description: The contract does not support users who have the same asset debt; if intentional, this should be documented clearly so the front end can show that to the user to reduce frustration.

Recommendation: Add documentation or a view method that checks if a user can use leverage.

Customer's response: Fixed in PR#1473.

I-02. Double call to onFlashLoan

Description: onFlashLoan has no limit to the number of times it can be called. Under normal use, it should always be called only once in a flow.

Recommendation: add a transient onlyOnce modifier to make sure this method could never be called more than once for each call.

Customer's response: Fixed in PR#1477.

I-03. No rescue function for native token

Description: While the rescueTokens function can retrieve any ERC20 token, all native token deposits cannot be rescued.

Recommendation: add a rescueNativeToken function.

Customer's response: Fixed in PR#1482.

I-04. Reset transient variables at the end of the function flow

Description: The transient variables are set at the beginning of the transaction but not explicitly reset at the end of the function flow. It is a general good practice to reset these to avoid any possibility of state corruption due to batching.

Recommendation: reset the transient variable at the end of the function flow.

Customer's response: Fixed in PR#1477.

I-05. Rebase tokens are incompatible with the module

Description: Rebase tokens are not compatible with the leverage module due to unpredictable return values and variable balances due to rebasing.

Recommendation: Add natspec to make clear that rebase tokens not never be used for leverage.

Customer's response: Acknowledged. Rebasing tokens are not supported in Silo V2 protocol

I-06. Leverage fee cannot be set to maximum value

Description: In RevenueModule.sol, the leverage fee setter enforces:

```
JavaScript
   function setLeverageFee(uint256 _fee) external onlyRole(OWNER_ROLE) {
        require(revenueReceiver != address(0), ReceiverZero());
        require(leverageFee != _fee, FeeDidNotChanged());
@>      require(_fee < MAX_LEVERAGE_FEE, InvalidFee());

        leverageFee = _fee;
        emit LeverageFeeChanged(_fee);
    }
```

This means the maximum allowed fee (5%, or 0.05e18) can never actually be set, as only values strictly less than the maximum are accepted. For example, setting the fee to exactly 5% will always revert.

Recommendation: Change the check to allow the maximum value:

```
JavaScript
    function setLeverageFee(uint256 _fee) external onlyRole(OWNER_ROLE) {
        require(revenueReceiver != address(0), ReceiverZero());
        require(leverageFee != _fee, FeeDidNotChanged());
-       require(_fee < MAX_LEVERAGE_FEE, InvalidFee());
+       require(_fee <= MAX_LEVERAGE_FEE, InvalidFee());

        leverageFee = _fee;
        emit LeverageFeeChanged(_fee);
    }
```

Customer's response: Fixed in PR#1507.

I-07. CREATE2-based cloning is incompatible with zkSync Era

Description: The LeverageRouter contract uses the Clones library and the CREATE2 opcode to deterministically deploy user-specific leverage contracts. However, zkSync Era uses a different address derivation formula for CREATE2 than Ethereum mainnet and most EVM-compatible chains. As a result, the predicted addresses and actual deployed addresses will not match, and deterministic contract creation will not work as intended.

Recommendation: If you plan to deploy on zkSync Era, do not rely on standard CREATE2 address prediction or deterministic contract creation.

Customer's response: Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.