Fuzzing and Formal Verification

Of the **Claim Fee Maker** Smart Contracts

Produced for **Maker**

By Deco

www.deco.tech

Table of Contents

Repositories

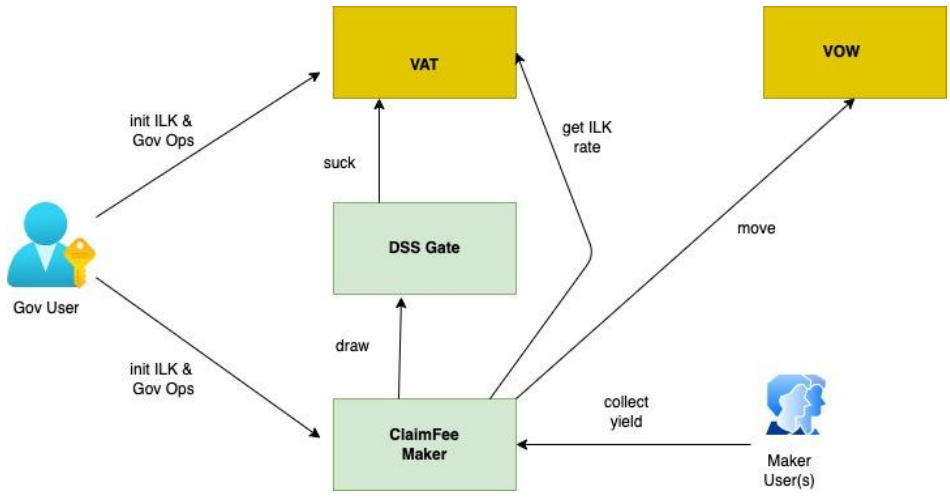
This doc entails to establish invariants for fuzzing on the below ClaimFee Contracts.

Smart Contract	Repository	Commit Hash
ClaimFee	https://github.com/deco-protocol/claim-fee-maker	4c02193
Dss-Gate	https://github.com/deco-protocol/dss-gate	80611d5

ClaimFee Contracts Overview

ClaimFee Maker (aka CFM) is an extension module of Maker Protocol that integrates to offer 'Fixed-rate vaults as a Feature' on existing collateral types like ETH-A, WBTC-A etc over a certain period of time. A balance is issued for a user with an existing Maker Vault. This 'Claim Fee' is in turn mapped to a specific ilk (Collateral Type), Issuance time and Maturity, which is coined as the term 'class'. This issued 'ClaimFee' is leveraged to offset the stability fee accrued on the Maker Vault. A user can redeem the 'ClaimFee' by executing `collect` function on ClaimFee Maker smart contract. The 'ClaimFee' smart contracts consists of two contracts namely :

- Claim Fee Maker
- DSS-Gate



[Figure 1]

In the above, the highlighted contracts (ClaimFeeMaker and DSS Gate) are ClaimFee Contracts which integrate with Core Maker contracts such as VAT and VOW.

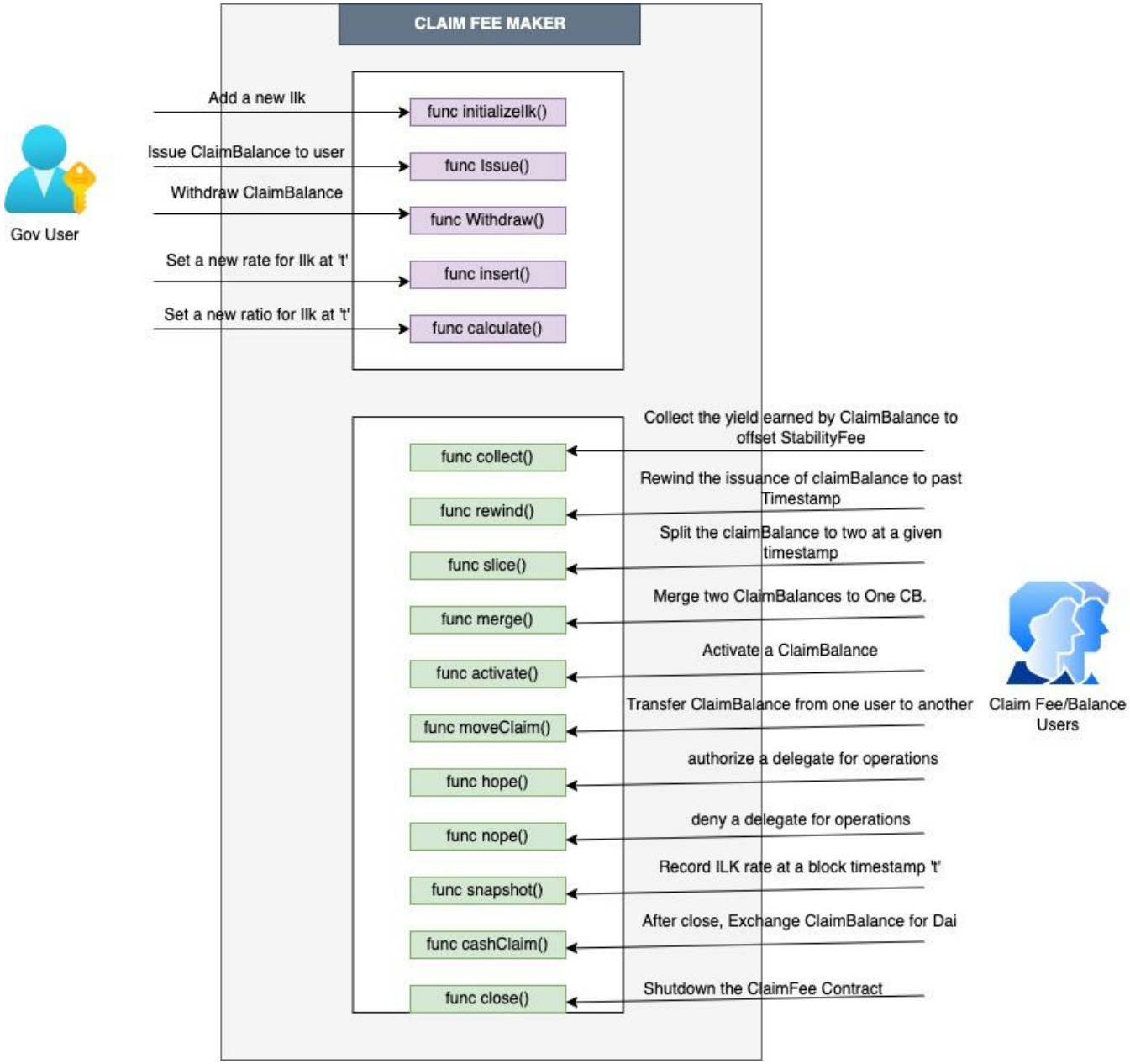
A ClaimFee Maker contract tracks all the issued balances for various users on supported ilk types, tracks ilk rates, allows privileged operations for governance users and user level operations for Maker users to collect yield and split or merge balances etc.

A DSS-Gate contract is a special governance tool which limits vat.suck() capability. This is a security lever in the hands of Maker that can impose a limitation on the amount of DAI that can be withdrawn by an authorized smart contract. It onboards various integrations (claim fee maker is one such) to suck/withdraw funds.

A typical sequence of events would look like as follows :

1. Governance initializes a supported ilk (Collateral Type) in the ClaimFee Maker contract (aka CFM).
2. Maker sells *fixed-rate* tokens i.e a balance to an existing user with a regular Maker Vault. The mode of sales is not disclosed yet.
3. Upon a sale of this balance, Governance will invoke the `issue` function in CFM to record the sale event. This balance is issued for a definitive term that tracks both issuance and maturity time period.
4. After Vault maturity, A Maker user invokes the `collect` function in CFM to collect the yield. The CFM utilizes a VAT authorized contract (i.e dss-gate) to suck/withdraw the required DAI and thus transfer to the user.
5. The DAI collected by the user will be leveraged to offset the stability fee accrued in his/her regular Maker Vault. And thus offering a fixed rate vault solution.

The CFM exposed additional functional capabilities for both by Governance and Maker users as depicted below:


CLAIM FEE MAKER
Gov User
Add a new Ilk
func initializeIlk()
Issue ClaimBalance to user
func Issue()
Withdraw ClaimBalance
func Withdraw()
Set a new rate for Ilk at 't'
func insert()
Set a new ratio for Ilk at 't'
func calculate()
Collect the yield earned by ClaimBalance to offset StabilityFee
func collect()
Rewind the issuance of claimBalance to past Timestamp
func rewind()
Split the claimBalance to two at a given timestamp
func slice()
Merge two ClaimBalances to One CB.
func merge()
Activate a ClaimBalance
func activate()
Transfer ClaimBalance from one user to another
func moveClaim()
authorize a delegate for operations
func hope()
deny a delegate for operations
func nope()
Record ILK rate at a block timestamp 't'
func snapshot()
After close, Exchange ClaimBalance for Dai
func cashClaim()
Shutdown the ClaimFee Contract
func close()
Claim Fee/Balance Users

[Figure 2]

ECHIDNA Fuzzing

Fuzzing is a type of property based testing technique to discover software vulnerabilities. A process of generating a massive number of random scenarios, to find our values that produce unexpected results. In other words, Fuzzing helps to find call sequences to falsify the predicates(aka assertions) made against an Ethereum smart contract. In addition, It also assists in reporting max gas usage based on plotted fuzz campaigns. "Fuzzing" is a type of software testing that involves providing random or semi-random inputs to a program in order to detect unexpected behavior or crashes. In the context of smart contracts, fuzzing can be used to test the behavior of a contract when it receives unexpected or malformed inputs. This can help identify potential bugs or vulnerabilities in the contract's code that could be exploited by an attacker. The goal of fuzzing is to uncover any unknown or undiscovered weaknesses in the contract's code, which can then be fixed before the contract is deployed on a blockchain.

Echidna as Fuzzer

'Echidna' is a property-based fuzzer, which is based on abstract state machine modeling technique and automatic test case generation for verifying the properties of a program by executing a large set of inputs generated stochastically. Echidna is a library for generating random sequences of calls against a given contract's ABI and thus preserves user-defined predicates/invariants.

Echidna can be used to detect, assess and analyze the following :

1. Incorrect Access Control
2. Incorrect State Machine
3. Incorrect Arithmetic
4. User-defined custom Assertions
5. Consumption of Gas
6. Coverage and Corpus collection
7. Performance evaluation

`echidna-test` is an executable that takes a contract and list of user defined invariants as input. For each invariant, it generates random sequences of calls to the contracts and checks if the user-defined predicate/invariant holds. If it can find some way to falsify the invariant, it prints the call sequence that does so. If it can't, you have some assurance the contract is safe.

ECHIDNA Versioning

Echidna is compatible with various solidity compiler versions. Echidna has two versions namely Echidna 1.0 and 2.0 (being the latest stable release). Echidna 2.0 is compatible with solc 0.8+ only.

ClaimFree Maker Contract Invariants

In the context of smart contracts, an "invariant" is a condition that should always be true for the contract's state, regardless of the inputs or actions taken. Invariants are often used to describe certain properties of the contract that must be maintained throughout its execution, such as the balance of an account or the total supply of a token.

A list of ClaimFee Maker invariants are established as follows:

1. ILK can be initialized by authorized wards (admins) only
    a. Wards are the admins who have special privilege to initialize ILK. Only one of these wards has the privilege to initialize ILK in claim fee.

2. Relying/Denying a new/existing ward is restricted to only existing wards.
    a. An existing ward can add a new or remove ward from the contract. In other words, a normal user cannot add/remove a ward.

3. Balance can be issued by a ward/governance only

4. Rate Insertion for a ILK can be performed by wards only
    a. Administrators (Governance) are the only authorized entities who can insert ilk rate at a given timestamp.

5. A governance user cannot withdraw Balance after the deco instance is shutdown.

6. A governance user is the only authorized to insert rate

7. A governance user can set ratio after the deco close

8. A governance user cannot set ratio before the deco close

9. A user cannot rewind to past after the deco instance is closed.

10. A user can add a new delegate to manage on their behalf

11. A user cannot slice after the maturity ts

12. A user cannot slice before issuance ts

13. Activation of balance can be performed by a normal user or ward.

14. An ilk cannot be initialized until its initialized in VAT

15. Ilk can be initialized only ONCE, No reinitialization allowed
16. A user cannot move another users' Balance
17. Rate cannot be inserted in future
18. Overwriting of rates at a given timestamp is not permitted.
19. A user cannot withdraw their own balance
20. A user can collect DAI only when the collect timestamp is after the issuance.
21. ClaimFee cannot be issued after close
22. A User cannot collect DAI of another user
23. User cannot rewind issuance timestamp after close
24. A user cannot merge Balance of another user
25. ClaimFee contract cannot be closed while VAT is not shut down
26. In order to activate a Balance, the issuance rate must not be recorded.
27. In order to activate a Balance, the rate must be recorded at activation ts.
28. A user cannot rewind to past ts if past rate is not recorded.
29. A user cannot take a snapshot of uninitialized ILKs
30. Users can take a snapshot and thus the rate is recorded at current timestamp..
31. A governance user inserts a rate and thus is recorded for respective Ilk.
32. ILK cannot be initialized in Claim Fee until it is initialized in VAT
33. A user can transfer both activated and unactivated balance upon sufficient book/ balance.
34. Claimfee cannot be issued if the rate is not recorded
35. User transfers the claim fee to another user. The balances are adjusted accordingly.

36. A governance user issues Balance, and thus the balances and totalSupply are adjusted accordingly.

37. suck/withdraw capability

38. When a user executes a slice function, their balances MUST be adjusted accordingly.

39. When a user executes a merge function, their balances MUST be merged to single balance.

40. A governance user can withdraw from a user only after close, and thus their balances are adjusted accordingly

41. The amount of DAI transferred from user to vow cannot exceed the difference computed based on rewindRate and issuanceRate.

42. A user loses yield until the Ilk is activated at a given ts.

Echidna Fuzz Testing Infrastructure

- Fuzzing Software : `Echidna 2.0`
- Cloud Compute : `AWS EC2 , Linux AMI (ami-0d9858aa3c6322f73)`
- Runtime : `Docker`
- Build : GNU Make
- Docker Image : `trailofbits/eth-security-toolbox`
- CPU : `32 Cores`
- Memory : `60 GB`
- Architecture : `x86_64`

Echidna Test Configuration

- # Echidna Test Mode
  - **`testMode: "assertion"`**
- # Total number of test sequences to run
  - **`testLimit: 1000000 ( 1 million)`**
- Defines how many transactions are in a test sequence
  - **`seqLen: 200`**
- # solcArgs allows special args to solc
  - **`solcArgs: "--optimize --optimize-runs 200"`**
- # Maximum time between generated txs; default is one week
  - **`maxTimeDelay: 15778800 # approximately 6 months`**
- # Deployer is address of the contract deployer (who often is privileged owner, etc.)

    - **`deployer: "0x41414141"`**
- # Sender is set of addresses transactions may originate from
    - **`sender: ["0x42424242", "0x43434343"]`**

Echidna Test Results


me(): passed!
test_activate_invalid_timestamp(uint256): passed!
holder_addr(): passed!
test_nope(address): passed!
test_rewind(bytes32,address,uint256,uint256,uint256,uint256): passed!
test_activate_invalid_activationrate(uint256): passed!
ETH_A(): passed!
holder(): passed!
test_hope(address): passed!
test_calculate_wardonly(uint256): passed!
usr_addr(): passed!
test_already_initialized_ilk(): passed!
test_slice_unauthorized(uint256): passed!
test_collect_unauthorized(uint256): passed!
test_collect(bytes32,address,uint256,uint256,uint256,uint256): passed!
test_collect_invalid_collection(uint256): passed!
test_collect_post_maturity(uint256): passed!
AssertionFailed(..): passed!
Unique instructions: 32628
Unique codehashes: 6
Corpus size: 20
Seed: 767713293143226206

```
Echidna 2.0.1
Tests found: 69
Seed: 5503150210251831719
Unique instructions: 31833
Unique codehashes: 6
Corpus size: 15
Tests
AssertionFailed(..): PASSED!
assertion in test_collect_post_maturity(uint256): PASSED!
assertion in test_collect_invalid_collection(uint256): PASSED!
assertion in test_collect(bytes32,address,uint256,uint256,uint256,uint256): PASSED!
assertion in test_collect_unauthorized(uint256): PASSED!
assertion in test_slice_unauthorized(uint256): PASSED!
assertion in test_already_initialized_ilk(): PASSED!
assertion in usr_addr(): PASSED!
assertion in test_calculate_wardonly(uint256): PASSED!
assertion in test_hope(address): PASSED!
assertion in holder(): PASSED!
assertion in ETH_A(): PASSED!
assertion in test_activate_invalid_activationrate(uint256): PASSED!
assertion in test_rewind(bytes32,address,uint256,uint256,uint256,uint256): PASSED!
assertion in test_nope(address): PASSED!
assertion in holder_addr(): PASSED!
assertion in test_activate_invalid_timestamp(uint256): PASSED!
assertion in me(): PASSED!
assertion in usr2_addr(): PASSED!
assertion in gov_user(): PASSED!
assertion in test_cannot_initialize_ilk_vatmiss(): PASSED!
assertion in test_move_insufficient_balance(uint256): PASSED!
assertion in gov_addr(): PASSED!
assertion in test_calculate_afterclose(uint256): PASSED!
assertion in test_withdraw_afterclose(uint256): PASSED!
assertion in test_rely(address): PASSED!
assertion in test_rewind_unauthorized(uint256): PASSED!
assertion in test_activate_invalid_issuancerate(uint256): PASSED!
assertion in test_withdraw_wardonly(uint256): PASSED!
assertion in test_activate_unauthorized(uint256): PASSED!
assertion in test_ilk_init(): PASSED!
assertion in ETH_Z(): PASSED!
assertion in test_close_conditions(): PASSED!
assertion in test_merge(bytes32,address,uint256,uint256,uint256,uint256): PASSED!
assertion in test_snapshot(bytes32): PASSED!
assertion in gate(): PASSED!
assertion in test_slice_beforeissuance(uint256): PASSED!
assertion in test_insert_pastrate_not_present(): PASSED!
assertion in test_slice_afternaturity(uint256): PASSED!
assertion in test_cannot_issueClaim_unlessRateRecorded_ThrowsError(uint256): PASSED!
assertion in test_insert_rate_overwrite_notallowed(): PASSED!
assertion in vow(): PASSED!
assertion in test_calculate_beforeclose(): PASSED!
assertion in test_rewind_afterclose(uint256): PASSED!
assertion in cfm(): PASSED!
assertion in test_issue(address,uint256,uint256,uint256): PASSED!
assertion in usr(): PASSED!
assertion in test_snapshot_not_initialized(): PASSED!
```

CERTORA FORMAL VERIFICATION

Issues Found

Critical - No Critical issues found.
Major - No Major issues found
Minor - No Minor issues found.

Disclaimer

The Certora Prover takes input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. The Certora Prover is scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification. The content of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Deco or any of its employees be liable for any claim, damages or other liability arising, whether in an action of the contract, tort or otherwise, arising from, out of or in connection with the results reported here.

CVL Rules	Repository	Commit Hash
ClaimFee.spec	https://github.com/deco-protocol/claim-fee-maker/	9d4182f

Verification Conditions

Notation

✔ Indicates the rule is formally verified on the latest reviewed commit. Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

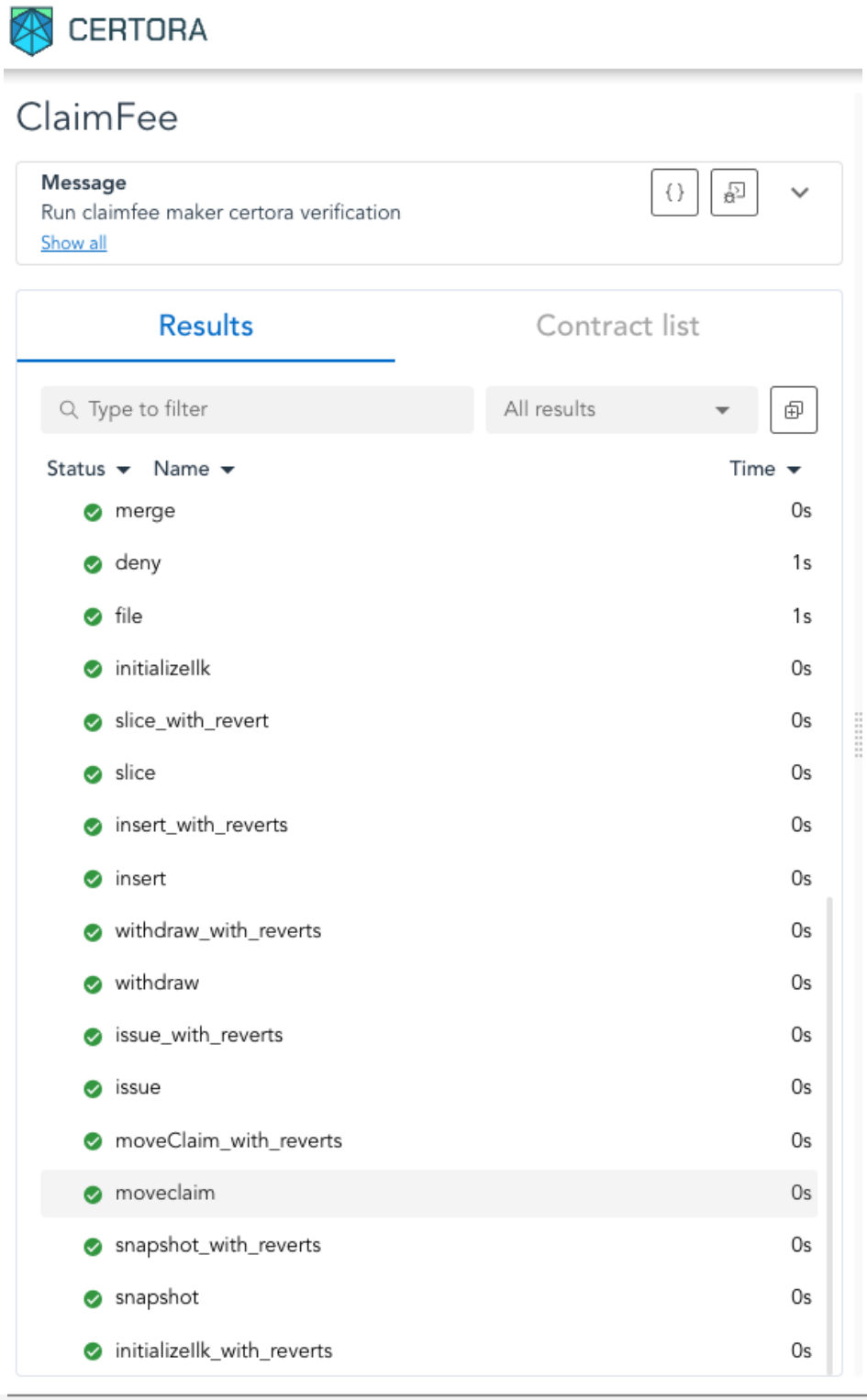
CVL Rules

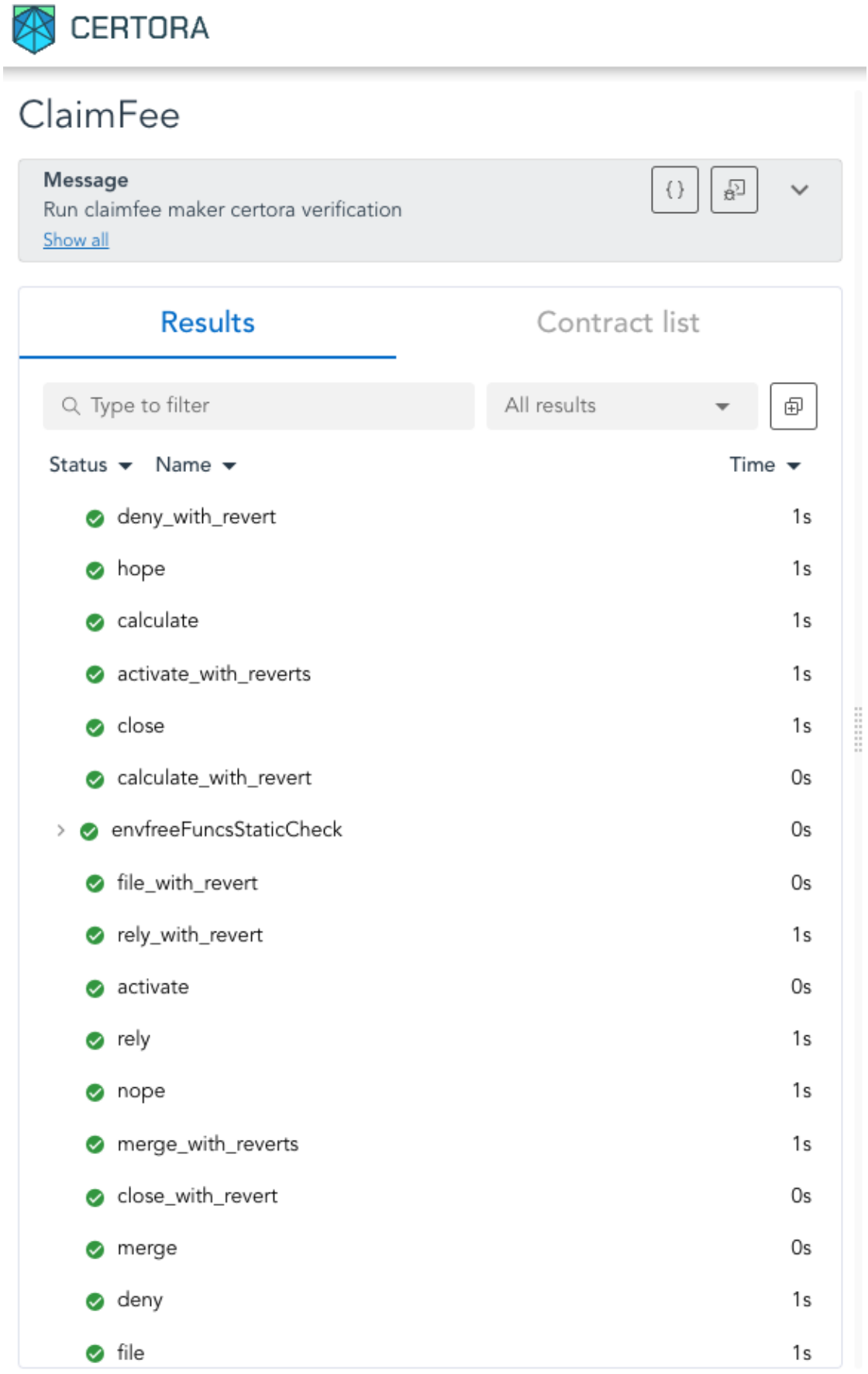
CVL Rule Description	Implementation	Status
`Assert new ilk added by governance`	`initializeIlk`	✔
`Verify all the revert rules when adding a new ilk`	`initializeIlk_with_reverts`	✔

Assert the recorded ilk rate at a given timestamp.	snapshot	✔
Verify all the revert rules when recording ilk rate at block timestamp.	snapshot_with_reverts	✔
Assert the balances when a user moves Balance to another.	moveclaim	✔
Verify all the revert rules when a user moves their claim balance to another.	moveClaim_with_reverts	✔
Verify that governance can issue Balance	issue	✔
Verify all the reverts for the issue. A governance can issue Balance as long as the deco instance is active.	issue_with_reverts	✔
A governance user can withdraw their claim balance	withdraw	✔
Verify all the reverts when a governance can withdraw a claim balance.	withdraw_with_reverts	✔
Verify that governance can set the rate value at a given timestamp	insert	✔
Verify all the revert rules when a governance can insert a rate value	insert_with_reverts	✔
Verify that slice method splits the claim balance into two balances at a given timestamp.	slice	✔
Verify all the revert rules when a claim balance is split into two. The user balances are thus adjusted accordingly.	slice_with_revert	✔
Verify the process of merging claim balances. The claim balances should be adjusted accordingly.	merge	✔
Verify all the revert rules upon merging two claim balances to single claim balance	merge_with_reverts	✔
Verify that if ratio value is set correctly for a given ilk	calculate	✔
Verify all the reverts rules while setting the ratio value for ilk.	calculate_with_revert	✔
Verify the deco instance shutdown.	close	✔
Verify all the reverts captured during shutdown of deco instance.	close_with_revert	✔
Verify if user can activate Balance and thus the Balance gets adjusted accordingly.	activate	✔
Verify all the reverts when a user activates	activate_with_reverts	✔

balance		
Verify that method 'deny' works as expected,	deny	✔
Verify that method 'deny' reverts when sender is not authorized	deny_with_revert	✔
Verify that method 'rely' works as expected.	rely	✔
Verify that method 'rely' reverts when sender is not authorized.	rely_with_revert	✔
Verify that method 'hope' works as expected.	Hope, hope_with_revert	✔
Verify that method 'nope' works as expected.	Nope, nope_with_revert	✔
Verify if the 'gate' address can be updated as expected.	file	✔
Verify the the file method reverts as expected.	file_with_revert	✔

Certora Test Results




CERTORA
ClaimFee
Message
Run claimfee maker certora verification
Show all
Results
Contract list
Type to filter
All results
Status
Name
Time
deny_with_revert 1s
hope 1s
calculate 1s
activate_with_reverts 1s
close 1s
calculate_with_revert 0s
envfreeFuncsStaticCheck 0s
file_with_revert 0s
rely_with_revert 1s
activate 0s
rely 1s
nope 1s
merge_with_reverts 1s
close_with_revert 0s
merge 0s
deny 1s
file 1s

Glossary

1. *ClaimFee* - A special type of balance that is issued for a Maker user with vault to offset stability fee.
2. VAT - Vault Engine is a database that tracks all user balances, accounting, fungibility etc
3. *VOW* - A settlement module which tracks the debt in system
4. *ilk* - A collateral type such as ETH, WBTC etc which is onboarded to Maker platform
5. *Class* - A keccak256 encoded hash of ilk, issuance and maturity.
6. *Governance User* - A user with MKR and has power of voting.
7. ClaimFee User/ Maker User - A user who has a vault in Maker.
8. *Ward* - A user with administrator privileges and has ability to invoke privileged operations.
9. *DAI* - A stablecoin cryptocurrency on ETH pegged against USD.

References

[1] Dss-gate
[2] Claim-Fee-Maker Contract
[3] Contract Diagrams
[4] Echidna
[5] Trail Of Bits
[6] Maker Units
[7] Maker Docs