Formal Verification of Celo Governance Protocol

Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Alexander Nutz, Lior Oppenheim, Or Pistiner, Mooly Sagiv, John Toman, James Wilcox

1. Summary

This report presents work and conclusions based on a collaboration between Celo and Certora using the Certora Prover to formally verify security rules of Celo's governance protocol. We provided Celo with the Certora Prover, which is a tool for formally verifying that smart contracts satisfy specifications written in a language called Specify.

The cLabs team working on Celo worked with Certora to build specifications of the protocol. The main focus of the specifications was to ensure a set of invariants provided by the cLabs team holds in all possible cases. The Certora Prover tool verified that the implementation of the Celo protocol satisfies these specifications. The Certora prover was integrated into the CI system in order to guarantee that future updates to the code will not create new violations.

The project consisted of two phases. During the first phase, from July 30th 2019 through August 25th 2019, specifications were developed for the first major iteration of the protocol: the specifications developed during this period continue to be used to verify later versions of the code. The latest commit that was monitored is 9d2ec0d9399a6959565062f57efa037d03fd4fbc.

Following significant rewrites to some of the contracts inspected (BondedDeposits becoming Accounts and LockedGold), we started the second phase of the project including specification writing for Election and ReleaseGold contracts. The latest commit that was monitored is a46ce55ebd3867fa69290bf3be040846d433525d.

The Certora Prover proved the Celo Governance protocol implementation to be correct with respect to the formal rules written. During the verification process, the Certora Prover discovered a number of bugs in the code listed in Table 1. **All the high severity issues were promptly corrected prior to releasing the protocol, and the fixes were verified to satisfy the specifications.** Section 2 formally defines high level specifications of the protocol. The actual checked rules are available from the Celo git repository. Section 3 elaborates more on several of the interesting bugs found.

Table 1: List of main bugs discovered

Bug	Affected code	Description	Severity	Fix
Break sortedness of list	SortedLinkedList - An unbounded linked list of keys sorted according to values	The sortedness of the list can be violated by providing invalid arguments to the insert function (Section 3.1).	High	Proper check that the arguments are consistent with the current state of the list.
Front running & asset loss in elections	Elections - activation and revocation of votes	A rounding error in computing voting units from arguments. Other accounts could lose their assets. This can also lead to self-loss of assets (Section 3.2).	High	Increase precision of stored and computed vote units.
Asset loss	ReleaseGold	Total balance computation neglected the pending withdrawals. This can cause asset loss (Section 3.3).	High	Fixed total balance computation
Illegal group's voting state	Elections - reward distribution	System should not distribute rewards to a group without votes.	Low. (Triggering this bug is impossible in the current Go client.)	Add additional requirements to the Solidity code to guarantee that this cannot happen even in future versions of Go code.
Redundant code	Election	Redundant subtraction operation of an always zero value.	Low	Code removed.
Dead code	LockedGold	Unused nested structures.	Low	Code removed
Precision	FixidityLib - a library for fixed decimal	A potential precision loss during	Low	The product of the fractional parts

	point numeric computations	multiplication, due to scaling down of the fractional parts that could nullify their contribution to the product.		cannot overflow, so no need to scale down the fractional parts before multiplying.

1.1. Technology overview

The Certora Prover is based on well-studied techniques from the formal verification community. Specifications define a set of rules that call into the contract under analysis and make various assertions about their behavior. These rules, together with the contract under analysis, are compiled to a logical formula called a verification condition, which is then proved or disproved by the solver Z3. If the rule is disproved, the solver also provides a concrete test case demonstrating the violation.

The rules of the specification play a crucial role in the analysis. Without good rules, only very shallow properties can be checked (e.g. that no assertions in the contract itself are violated). To make effective use of Certora Prover, users must write rules that describe the high-level properties they wish to check of their contract. Certora Prover cannot make any guarantees about cases that fall outside the scope of the rules provided to it as input. Thus, in order to understand the results of this analysis, one must carefully understand the specification's rules.

1.2. Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

2. High Level Specification

2.1. Accounts

The Accounts contract provides infrastructure for the management of different roles of addresses in the Celo network. Some roles for addresses are signers (3 kinds: Attestation, Voting, and Validation; we refer to these as `SignerType`s below) and wallets. An account can authorize an address for a signing role on its behalf. An account can also set a wallet address.

The following attributes are used in the verification process:

```
isAccount(address x) : bool
      True when address x is a valid account
walletAddress(address x) : address
      The wallet (receiving) address of address x
authorizedBy(address x) : address
      Returns the address that authorizes address x
signerOftype(SignerType ty, address x) : address
      Returns the address that address x authorized for type ty.
```

The following operation is used in the verification process:

```
addSigner(SignerType ty, address x, signer d)
      Authorize address d to be a signer of type ty for address x
```

✔ **Valid account invariant**
An address that has a wallet or a signer of any type, must be an account.

$$\forall \texttt{address } x,d,\ x\neq d, x\neq 0, d\neq 0.\ \texttt{walletAddress}(x) = d \implies \texttt{isAccount}(x)$$

$$\forall \texttt{address } x,d,\ x\neq d, x\neq 0, d\neq 0. \forall \texttt{SignerType ty.}\ \texttt{signerOftype}(ty,x) = d \implies \texttt{isAccount}(x)$$

✔ **Valid authorization account invariant**
An address can be authorized only by a valid account.

$$\forall \texttt{address } d.\ d\neq 0.\ \ \texttt{authorizedBy}(d)\neq 0 \implies \texttt{isAccount}(\texttt{authorizedBy}(d))$$

A valid account can only be authorized-by itself

$$\forall \texttt{address } x,d,\ x\neq 0, d\neq 0.$$
$$(\texttt{isAccount}(d) \wedge \texttt{authorizedBy}(d) = x) \implies x = d$$

✔ **Valid signer invariant**
If address d has any signer permission for account x then d is authorized by x.

```
∀address x,d. x≠d,x≠0,d≠0. ∀SignerType ty.
      signerOftype(ty,x) = d ⟹ authorizedBy(d) = x
```

✔ **Unique authorized by**
An address can be authorized by at most one account.

```
∀address x,y,d. y≠0,x≠0,d≠0. ∀SignerType ty,ty'
      (signerOftype(ty,x) = d ∧ signerOftype(ty',y) = d) ⟹ x=y
```

✔ **Multiple signers**
An account may have multiple signers of different types.

```
{pairwise_distinct(x,d,d',0) ∧ ty≠ty'} addSigner(ty, x, d) ; addSigner(ty', x, d')
{authorizedBy(d) = authorizedBy(d') = x}
```

Here we use Hoare triples of the form `{p} C {q}`, which means that if program C executes starting in any state satisfying `p`, then it will end in a state satisfying `q`.

✔ **Persistent authorized by**
Once an address was authorized by an account, this authorization can no longer be changed.

```
∀address x,d. x≠d,x≠0,d≠0. authorizedBy(d) = x ⟹ Next authorizedBy(d) = x
```

Here **Next** is a temporal operator which denotes the next state after any operation of the contract.

2.2. LockedGold

LockedGold contract manages Celo Gold that is not available for spending. It provides mechanisms for the user to deposit Celo Gold and lock it in the contract, enabling participation in elections and governance. The contract manages conditions for unlocking Celo Gold and withdrawing it for other usages.

Attributes:

```
totalNonvotingLockedGold() : uint
      Returns the total amount of non-voting locked gold in contract
nonvotingLockedGold(address account) : uint
      Returns the total amount of non-voting locked gold for an account
pendingWithdrawals(address account, uint i) :uint
      Returns the amount of pending withdrawal for account available at the i-th entry
totalPendingWithdrawals(address account) : uint
      Returns the total amount of pending withdrawal for account
pendingReleased(address account, uint i) : bool
      True when index i of pending withdrawals of account is available for withdrawal
```

`totalBalance(address account) : uint`

Returns the total amount of assets of account, including pending withdrawals, non-voting locked gold and balance.

`unlockingPeriod() : uint`

Returns the waiting time before withdrawal

Operations:

`lock(address account, uint value)`

Deposit value to account's non-voting locked gold.

`unlock(address account, uint value) : uint`

Move value from account's non-voting into account's pending locked gold, returns a new entry of pending withdrawal

`withdraw(address account, uint value)`

Transfer of value from account's pending locked gold to account's balance.

✔ **Integrity of non-voting locked gold:**
The current total non-voting locked gold is the sum of non voting locked gold of all accounts.

`totalNonvotingLockedGold = ∑ address account. nonvotingLockedGold(account)`

✔ **Fixed `totalBalance` over time** (except on slash, decrementNonVotingAccount, incrementNonVotingAccountBalance)
Total balance of an account does not change by locked-gold operations.

`totalBalance(account) = x ⟹ Next totalBalance(account) = x`

✔ **No impact on assets by other accounts**
Methods performed by different addresses cannot affect each other's locked gold

`{a≠b ∧ old = nonvotingLockedGold(a)} b.op() {old = nonvotingLockedGold(a)}`
`This rule fetches the value of nonvotingLockedGold(a) and ensures that it does not change across any execution of b.op().`

✔ **No premature withdrawal**
Withdraw is possible only after unlocking period

`Once i=unlock(account,x) => x ≤ nonvotingLockedGold(account) + totalPendingWithdrawals(account) Until pendingReleased(account,i)`

This is a rather complex property. It states that during the unlock period, `x` cannot exceed the total amount of locked and pending gold. Notice the use of `Once` and `Until` temporal operators.

✔️ **Withdrawal possible**

`totalPendingWithdrawals(account) = x ⟹` **Eventually** `withdraw(account, x)`

✔️ **Only an account can have pending withdrawals**

`totalPendingWithdrawals(account) > 0 ⟹ Accounts.isAccount(account)`

This rule connects the security of multiple contracts as `isAccount` is an attribute of the Accounts contract.

2.3. Elections

This contract manages the voting processes and election of validators.
The following attributes and operations are used in the verification process:

`unitsForGroup(address group) : uint`

Returns the voting units for group (without increased precision)

`votesForGroup(address group) : uint`

Returns the value of assets for group including rewards

`unitsForGroupByAccount(address group, address account) : uint`

Returns the total voting units for group by account

`votesForGroupByAccount(address group, address account) : uint`

Returns the value in assets of voting units for group by account

`pendingVotesForGroup(address group) : uint`

Returns the pending votes for group

`pendingVotesForGroupByAccount(address group, address account) : uint`

Returns the pending votes for group by account

`totalBalance(address account) : uint`

Returns the total amount of assets of account, including LockedGold.totalBalance(account), pending votes and value of voting units of account to all groups

Operations:

`revokeActive(address group, address account, address value) : uint`

revoke value amount of assets for group by account

✔️ **Integrity of voting units**

`unitsForGroup(group) = ∑ address account. unitsForGroupByAccount(group,account)`

✔️ **Integrity of total assets**

```
votesForGroup(group) = ∑ address account. votesForGroupByAccount(group,account)
```

✔️ **Integrity of pending votes**

```
pendingVotesForGroup(group) = ∑ address account.
pendingVotesForGroupByAccount(group,account))
```

✔️ **Integrity of total assets with respect to voting units**

A group's total asset is more than its voting units

```
unitsForGroup(group)  ≤ votesForGroup(group)
```

✔️ **Emptiness**

A group with no voting units can not have any assets

```
unitsForGroup(group) = 0 ⟹ votesForGroup(group) = 0
```

This rule discovered the illegal group's voting state issue on `distributeEpochRewards` in case that rewarding a group with no votes.

✔️ **Whole assets**

An account holding all voting units if and only if it has all assets of the group

```
unitsForGroupByAccount(group, account) = unitsForGroup(group) ⟺
votesForGroupByAccount(group, account) = votesForGroup(group)
```

✔️ **`totalBalance` only increases over time (except on `forceDecrementVotes`)**

```
totalBalance(account) <= Next totalBalance(account)
```

✔️ **Additivity of revoking active votes**

This is specified in terms of code equivalence denoted by `P1 ~ P2`.

```
revokeActive(group, account, x) ; revokeActive(group, account, y) ~
revokeActive(group, account, x+y)
```

This requires that for every initial state `s`, revoking active votes of `x` before revoking `y` has the same effect of revoking `x + y` simultaneously starting from `s`.

2.4. Release Gold

This contract manages an amount of Celo Gold "granted" to an address (beneficiary) that is released over a defined schedule.

The following attributes and operations are used in the verification process. Notice that these operations have side effects on the state of the contract. For example, a successful withdrawal updates the remaining balance.

```
withdraw(uint x): bool
      A successful withdrawal of x gold
totalWithdrawn(): uint
      Amount of withdrawn
totalBalance(): uint
      Total amount of assets
totalReward(): uint
      Total amount of rewards given
releasedTotalAmount(): uint
      Total released
MAX_WITHDRAWL: uint
      Total amount of gold granted
```

✔️ **Additivity to avoid frauds and lock**

withdrawal of x+y can be performed either all at once or gradually.

`withdraw(x); withdraw(y) ~ withdraw(x+y)`

Here `x` and `y` takes arbitrary values as opposed to testing which checks particular instances of `x` and `y`. Notice that this property compares two programs: one with two consecutive withdrawals and one with a single withdrawal.

In the left side the withdrawal of `y` is performed after the withdrawal of `x` and in the right side, the sum of `x` and `y` is withdrawn in a single call. Notice that both programs start in the same initial state.

✔️ **Max limit to avoid frauds (except on slashing)**

Total of amount to be withdrawn is limited by `MAX_WITHDRAWL`

`∀uint x. withdraw(x) ⟹ totalWithdrawn() + x ≤ MAX_WITHDRAWL + totalReward()`

✔️ **Withdraw not locked (except on revocation)**

Eventually withdrawal up to a total of `MAX_WITHDRAWL` will be possible. Notice that this is a temporal property which must hold after some time.

`∀uint x. 0 < x ≤  totalBalance() - totalWithdrawn() ⟹ Eventually withdraw(x)`

Notice the usage of the operator **`Eventually`** which means that at some point in the future it will be possible to withdraw `x`. This is a special case of linear temporal logic.

✔️ **No premature withdrawal**

Withdrawal of more than the total released gold at the current time is impossible

```
∀uint x. withdraw(x) ⟹  x ≤ releasedTotalAmount() - totalWithdrawn()
```

✔ **`totalBalance` only increases over time (except on slashing)**

```
totalBalance() <= Next totalBalance()
```

This rule discovered the asset loss bug in ReleaseGold.

✔ **`releasedTotalAmount` only increases over time**

```
releasedTotalAmount() <= Next releasedTotalAmount()
```

LinkedList and SortedLinkedList

The LinkedList and SortedLinkedList are libraries used throughout the codebase. The below rules were checked on the "Sorted" implementations (bytes32 key, uint256 key, and address key) but could be easily adapted to check the non-sorted variation alone.
Attributes:

```
contains(uint key): bool
      True if list contains element key
greater(uint key): uint
      Successor of key in the list
lesser(uint key): uint
      Predecessor of key in the list
tail: uint
      Tail element of the list
head: uint
      Head element of the list
numElements: uint
      Number of elements in the list
```

✔ $\forall key.\ contains(key) \Rightarrow ((greater(key) = 0 \Leftrightarrow key = head) \wedge (lesser(key) = 0 \Leftrightarrow key = tail))$

✔ $head = 0 \Leftrightarrow numElements = 0$

✔ $head = 0 \Leftrightarrow tail = 0$

✔ $(\exists key.\ contains(key)) \Rightarrow (head \neq 0 \wedge contains(head) \wedge tail \neq 0 \wedge contains(tail))$

✔ $\neg contains(0)$

✔ $\forall key.\ contains(key) \Rightarrow value(tail) \leq value(key) \leq value(head)$ (SortedLinkedList)

✔ For successful insert, remove (LinkedList):

- New value of key is the provided value

- next key of new/updated key is either 0, or another element in the list
- previous key of new/updated key is either 0, or another element in the list

✔ For successful insert, remove (SortedLinkedList):

- For newly inserted/updated key, $value(lesser(key)) \leq value(key) \leq value(greater(key))$
- $key.next.prev = key$ (if it's a real key and not 0)
- $key.prev.next = key$ (if it's a real key and not 0)

✔ Insert succeeds if and only if all its preconditions hold:

- $key \neq 0$
- $key \neq lesser \ \wedge key \neq greater \ \wedge \neg contains(key)$
- $numElements < MAXINT$
- $lesser \neq 0 \ \vee greater \neq 0 \ \vee numElements = 0$
- $lesser = 0 \vee contained(lesser)$
- $greater = 0 \vee contained(greater)$
- $msg.value \neq 0$
- Either the lesser key is "correct" or the greater key is "correct", where "correct" refers to it really being the correct adjacent element of the new key in terms of order.

✔ For successful remove (SortedLinkedList):

- Removed key is no longer contained in the list
- Value of removed key is nullified (set to zero)
- Relevant pointers of next and previous keys are updated, and only them:
  - $lesser(key)$ next will point to $greater(key)$
  - $greater(key)$ previous will point to $lesser(key)$
  - $greater(greater(key))$ and $lesser(lesser(key))$ do not change

✔ Remove succeeds only if key is contained in the list

✔ Irrelevant elements' values do not change in either insert, update or remove, so sortedness is preserved

StableToken

StableToken is a standard ERC20 token that is used to implement cUSD. It allows its users to adjust balances to inflation. We verified it against Certora's standard ERC20 token specification. Under the assumption that the inflation factor and rate stay constant at 1, this token satisfies the standard specification.

3. Description of the high severity bugs

The process of running the tool and analyzing the resulting bug reports uncovered the following concerns:

3.1 LinkedList and SortedLinkedList

- It is possible to invalidate the sortedness of the list by providing invalid `lesserKey` and `greaterKey` values. This could make the new key the maximal element even though it is not. After applying a patch, the rules hold.
- Clients of `AddressLinkedList` or `AddressSortedLinkedList` must not insert directly from the internal list library, since this will skip the expansion of the address to a proper `bytes32` key.

3.2 Election

The election contract supports distributing rewards to voters of a group that participated in the validation process. In order to distribute rewards in a fast manner and according to each voter's voting units, the contract stores:

- `groupAssets` -total assets of a group which includes assets invested by voters and rewards distributed to the group.
- `totalVotingUnits` - total voting units of users to the group.
- A mapping from users to voting units

The contract's conversion function unitsToVotes, from voting units to assets is:

```
unitsToVotes(u) = groupAssets * u / totalVotingUnits
```

And the inverse function for converting assets to voting units :

```
votesTounits(x) =  totalVotingUnits * x / groupAssets
```

The bug arises from rounding error in Solidity, when a voter withdraws part of her assets, the system converts to units and rounds down to integers, and updates the system with the computed units.

Let's take for example, a group with 50 voting units, two voters (out of more), and 150 rewards given to that group and follow the case where one voter withdraws part of her assets:

Step	A's voting units	B's voting units	totalVotingUnits	totalAssets
Start	10 units worth 40 assets	40 units worth 160 assets	50 units	200 (50 from voting, 150 from reward)
user A withdraws 19 assets - converted to 4 units due to rounding error (instead of 4.75)				

state after withdrawal	6 units worth 23 assets (instead of 21)	40 units worth 157 assets (instead of 160)	46 units	181 assets

In this scenario A received 2 assets on account of other shareholders of the group causing a front-running violation.

Another vulnerability from this bug is when depositing x assets, the calculated voting units may be worth less than expected. For example, if member C chooses to add 19 assets to the start point where the total assets is 200 and total voting units is 50. Due to the rounding error C receives 4 units (instead of 4.68) and now the group has 54 voting units with 219 assets. However C can now withdraw only 16 assets (16=floor(4*219/54), thus losing 3 assets.

This bug becomes more problematic as the ratio between totalUnits to totalAssets is bigger. It was fixed by storing units as a high precision integer.

3.3 Release Gold

The Release Gold contract tracks the total balance of assets it holds (Through getTotalBalance()), which also includes the assets that are being locked in the Locked Gold and Election contracts on its behalf. When the Release Gold contract sees that there are no more assets on its total balance, it terminates the contract. Specifically, it deletes the refund address that should get the remaining grant if the beneficiary is revoked.

However, the Release Gold contract disregards assets that are in a "pending withdrawal" state within the Locked Gold contract, when it is calculating its total balance. This might cause the Release Gold contract to mistakenly terminate itself, although there are still assets that are bound to it. Thus, for example, if an already revoked user withdraws all the assets he is entitled to from the Release Gold contract, while there are still unaccounted-for assets in a "pending withdrawal" state, then those assets will be lost indefinitely, since the refund address that should have gotten them does not exists anymore.

The bug was trivially fixed by including all the assets in the “pending withdrawal” queue in the calculation of the total balance of the Release Gold contract.

4. Conclusion

Certora Prover's verification increased confidence in the security of the Celo Governance protocol and smart contracts. During the process, several flaws in earlier versions of the implementation were discovered and fixed by the cLabs team. The rules that were proven by the Certora prover were integrated into the CI system to guarantee correctness of future updates. We thank the cLabs team for their collaboration on this project.