Property-Based Security Testing of Smart Contracts using Echidna

Gloria Fiammengo 2254256

Security in Software Applications
Academic Year 2025–2026

1 Introduction

1.1 Fuzzing

Fuzzing is a testing technique that consists of a semi-automatic generation of semi-random or mutated inputs aimed at exploring non-trivial cases where the application may be likely to crash. It allows exploring various execution paths of the program without having to manually write tests, by exercising the program with semi-valid complex inputs without being stopped at the parser.

Therefore, what does it mean to do fuzzing? Doing fuzzing means testing a program by repeatedly executing it with large amounts of automatically generated inputs, with the goal of triggering unexpected behaviors such as crashes, assertion violations, or invariant failures.

In this project we used Echidna to do so.

1.2 Echidna

Echidna is a property-based fuzzer for Solidity smart contracts. The developer specifies invariants as functions that must always hold after the execution. Echidna generates sequences of calls to public functions and checks the invariants after each execution. Thanks to the achievable expressivity of these functions, it detects violations corresponding to illegal states rather than generic crashes.

1.3 Source Code

The full source code of the smart contracts and the Echidna test harness used in this project is available in a public GitHub repository:

`https://github.com/Flegatonte/ssa-echidna-project`

The repository is provided for completeness and reproducibility, although code submission is not required for evaluation.

2 Part 1

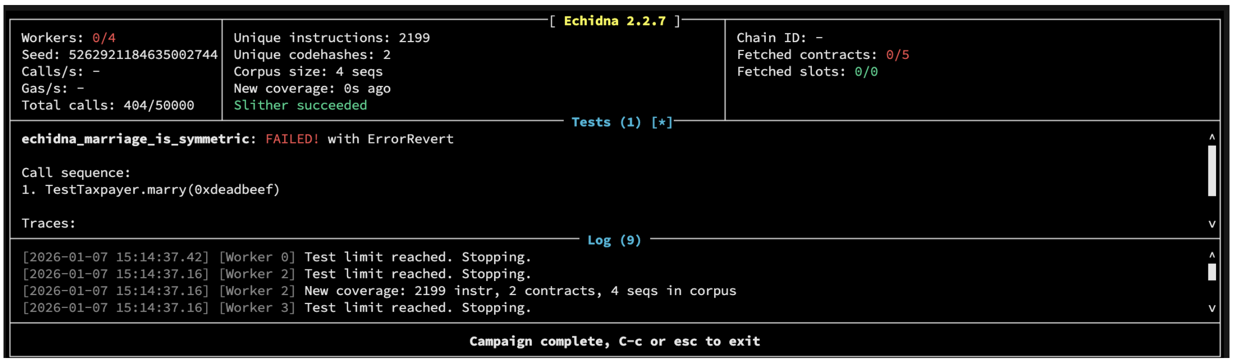
2.1 Invariant 1: Validity of the spouse

Specification If a taxpayer is married, the spouse must be another valid `Taxpayer` contract.

Property encoding This invariant is not encoded as a standalone Echidna property. Instead, it is implicitly enforced by the property `echidna_marriage_is_symmetric()`, which also captures the symmetry requirement of the marriage relationship.

In particular, whenever the taxpayer is married, the property retrieves the spouse address and attempts to query its marital state and recorded spouse through the `ITaxpayerView` interface. If the spouse address does not correspond to a deployed contract implementing the expected interface, the external calls fail and the property is violated. The implementation can be found in the following invariant.

Counterexample Echidna found a violation of this invariant with the following counterexample:



```
                                              [ Echidna 2.2.7 ]
Workers: 0/4                | Unique instructions: 2199        | Chain ID: -
Seed: 5262921184635002744   | Unique codehashes: 2             | Fetched contracts: 0/5
Calls/s: -                  | Corpus size: 4 seqs              | Fetched slots: 0/0
Gas/s: -                    | New coverage: 0s ago             |
Total calls: 404/50000      | Slither succeeded                |
                                              Tests (1) [*]
echidna_marriage_is_symmetric: FAILED! with ErrorRevert

Call sequence:
1. TestTaxpayer.marry(0xdeadbeef)

Traces:
                                              Log (9)
[2026-01-07 15:14:37.42] [Worker 0] Test limit reached. Stopping.
[2026-01-07 15:14:37.16] [Worker 2] Test limit reached. Stopping.
[2026-01-07 15:14:37.16] [Worker 2] New coverage: 2199 instr, 2 contracts, 4 seqs in corpus
[2026-01-07 15:14:37.16] [Worker 3] Test limit reached. Stopping.

                                    Campaign complete, C-c or esc to exit
```

The counterexample consists of a single call to `marry(0xdeadbeef)`. The address `0xdeadbeef` does not correspond to a contract implementing the expected `Taxpayer` interface and therefore cannot be safely queried, leading to a violation of the invariant.

Root cause The root cause is the lack of proper validation on the spouse address in the `marry` function. Arbitrary addresses could be provided as spouses without enforcing that they correspond to a valid `Taxpayer` contract.

Fix applied The issue was fixed by strengthening the validation logic in the `marry` function, enforcing that the spouse address corresponds to a valid `Taxpayer` contract, not only a generic contract or valid address. In particular, the fix prevents marriages with arbitrary addresses by requiring the spouse to be a deployed contract implementing the expected interface, as identified by the `isContract()` predicate already defined in the model.

```
require(new_spouse.code.length > 0, "spouse must be a contract");
require(Taxpayer(new_spouse).isContract(), "spouse not taxpayer");
```

Verification after the fix After applying the fix, Echidna was no longer able to violate Property 1. All tested executions preserved the validity of the spouse address. Further testing revealed an additional violation related to the symmetry of the marriage relationship, which will be addressed next by a separate property.

2.2 Invariant 2: Symmetry of the marriage relationship

Specification Marriage relationships between taxpayers must be symmetric. If a taxpayer is married, the spouse must also be married to the same taxpayer.

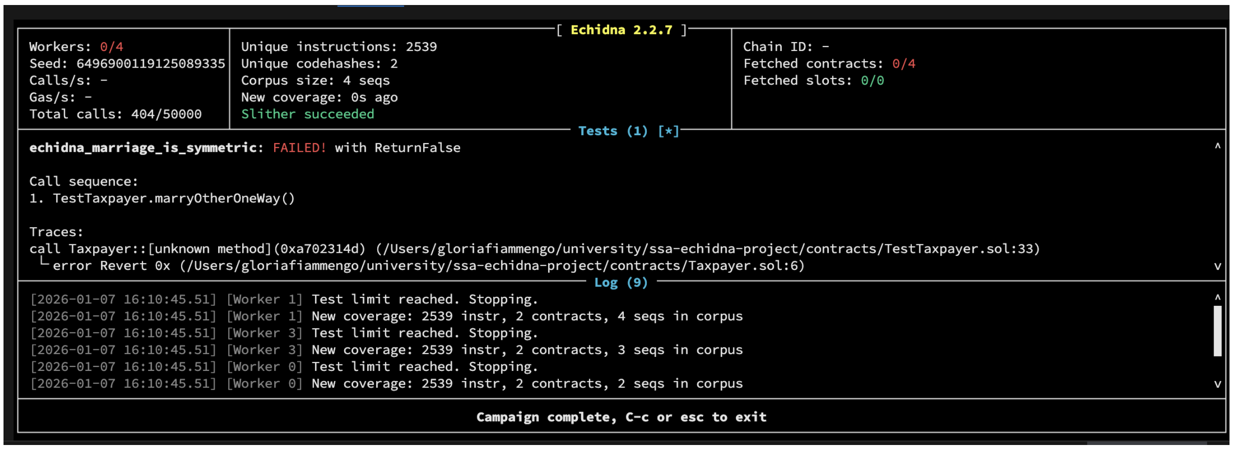
Property encoding This invariant is enforced by an Echidna property that checks the global consistency of the marital relationship through observable interactions with the spouse contract. When a taxpayer is marked as married, the property verifies that the spouse exists, is queryable as a `Taxpayer`, is also marked as married, and records the original taxpayer as its spouse.

Test harness snippet The symmetry requirement is encoded as a state-based property that inspects the joint marital state of the taxpayer and its spouse after each execution, using safe external calls to avoid assuming that the spouse address is valid.

```
function echidna_marriage_is_symmetric() public view returns (bool) {
    if (!getIsMarried()) return true;

    address sp = getSpouse();
    if (sp == address(0)) return false;

    try ITaxpayerView(sp).getIsMarried() returns (bool spMarried) {
        if (!spMarried) return false;
    } catch {
        return false;
    }

    try ITaxpayerView(sp).getSpouse() returns (address spSpouse) {
        return spSpouse == address(this);
    } catch {
        return false;
    }
}
```

The property is conditional on the local marital state. If the taxpayer is not married, the invariant vacuously holds. If the taxpayer is married, the property enforces that: (i) the spouse address is non-zero; (ii) the spouse can be queried through the expected `Taxpayer` interface; (iii) the spouse is also marked as married; and (iv) the spouse's recorded partner is exactly the original taxpayer contract. Any violation of these conditions causes the property to return `false`, allowing Echidna to detect both asymmetric marriages and marriages involving invalid spouse addresses.

Counterexample Echidna found a violation of this invariant with the following counterexample:



```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 2539          Chain ID: -
Seed: 6496900119125089335       Unique codehashes: 2               Fetched contracts: 0/4
Calls/s: -                      Corpus size: 4 seqs                Fetched slots: 0/0
Gas/s: -                        New coverage: 0s ago
Total calls: 404/50000          Slither succeeded
Tests (1) [*]
echidna_marriage_is_symmetric: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.marryOtherOneWay()

Traces:
call Taxpayer::[unknown method](0xa702314d) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:33)
 └error Revert 0x (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/Taxpayer.sol:6)
Log (9)
[2026-01-07 16:10:45.51] [Worker 1] Test limit reached. Stopping.
[2026-01-07 16:10:45.51] [Worker 1] New coverage: 2539 instr, 2 contracts, 4 seqs in corpus
[2026-01-07 16:10:45.51] [Worker 3] Test limit reached. Stopping.
[2026-01-07 16:10:45.51] [Worker 3] New coverage: 2539 instr, 2 contracts, 3 seqs in corpus
[2026-01-07 16:10:45.51] [Worker 0] Test limit reached. Stopping.
[2026-01-07 16:10:45.51] [Worker 0] New coverage: 2539 instr, 2 contracts, 2 seqs in corpus

Campaign complete, C-c or esc to exit
```

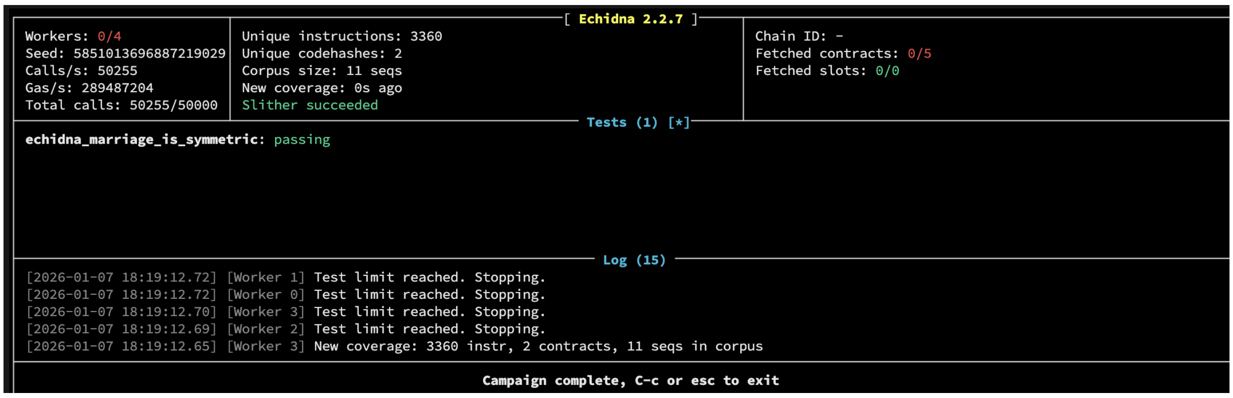
The counterexample consists of a single call to the testing helper function `marryOtherOneWay()`. This call showed that a one-sided marriage is indeed possible, where one taxpayer is marked as married while the spouse remains single, leading to a violation of the symmetry requirement.

Root cause The root cause is that the marriage procedure did not enforce symmetric state updates. A taxpayer could be marked as married without updating the spouse state, making one-sided marriages reachable.

Fix applied The issue was fixed by enforcing atomic and symmetric updates of the marriage relationship. An auxiliary function has been created whose duty is to make the spouse marry back the caller. When a taxpayer marries another taxpayer, both contracts are updated within the same transaction, ensuring that no intermediate or inconsistent marital state can be observed.

```
function _marryBack(address new_spouse) external {
    require(Taxpayer(new_spouse).isContract(), "spouse not taxpayer");
    require(!isMarried, "already married");
    require(msg.sender == new_spouse, "only spouse can confirm");

    spouse = new_spouse;
    isMarried = true;
}
```

Verification after the fix After applying the fix, Echidna was no longer able to produce states that violate the symmetry of the marriage relationship.



```
[ Echidna 2.2.7 ]
Workers: 0/4                  Unique instructions: 3360          Chain ID: -
Seed: 5851013696887219029     Unique codehashes: 2               Fetched contracts: 0/5
Calls/s: 50255                Corpus size: 11 seqs               Fetched slots: 0/0
Gas/s: 289487204              New coverage: 0s ago
Total calls: 50255/50000      Slither succeeded
                                        Tests (1) [*]
echidna_marriage_is_symmetric: passing

                                        Log (15)
[2026-01-07 18:19:12.72] [Worker 1] Test limit reached. Stopping.
[2026-01-07 18:19:12.72] [Worker 0] Test limit reached. Stopping.
[2026-01-07 18:19:12.70] [Worker 3] Test limit reached. Stopping.
[2026-01-07 18:19:12.69] [Worker 2] Test limit reached. Stopping.
[2026-01-07 18:19:12.65] [Worker 3] New coverage: 3360 instr, 2 contracts, 11 seqs in corpus
                              Campaign complete, C-c or esc to exit
```

2.3 Invariant 3: Global coherence of divorce

Specification The marriage relationship between two taxpayers must remain globally coherent between the acting parts. At any time, either both taxpayers are unmarried with the spouse field set to `0x0`, or both are married to each other. Under the assumptions of the considered model, mixed states where one taxpayer is married and the other is not are considered invalid, as they cannot be resolved by subsequent operations.

Property encoding This invariant is implemented by checking the joint state of two taxpayers. If both are unmarried, both spouse pointers must be zero. If both are married, they must reference each other. Any mixed configuration violates the invariant.

Test harness snippet The invariant is encoded as a joint-state Echidna property that explicitly inspects the global marital configuration of two taxpayers, namely the contract under test and a second taxpayer instance (`other`), after each execution.

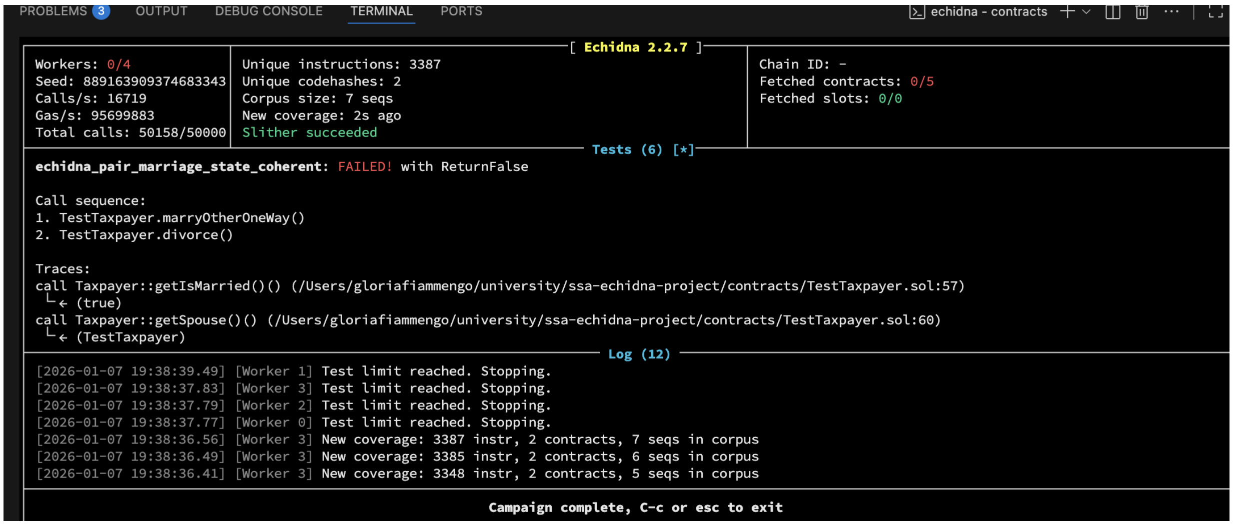
```
function echidna_pair_marriage_state_coherent() public view returns (bool) {
    bool meM = getIsMarried();
    bool otM = other.getIsMarried();

    address meS = getSpouse();
    address otS = other.getSpouse();

    if (!meM && !otM) {
        return meS == address(0) && otS == address(0);
    }

    if (meM && otM) {
        return meS == address(other) && otS == address(this);
    }

    return false;
}
```

The property explicitly enumerates the only two admissible global configurations. If both taxpayers are unmarried, both spouse pointers must be cleared. If both taxpayers are married, the spouse pointers must be mutually consistent and reference each other. Any mixed configuration, in which only one taxpayer is married or the spouse pointers do not match symmetrically, causes the property to return `false`, allowing Echidna to detect globally incoherent marriage or divorce states.

Counterexample Echidna found a counterexample consisting of the following call sequence:

```
marryOtherOneWay()
divorce()
```

After this sequence, the first taxpayer becomes unmarried, while the spouse remains married and still references the first contract, violating global coherence.



```
[ Echidna 2.2.7 ]
Workers: 0/4                   Unique instructions: 3387          Chain ID: -
Seed: 889163909374683343       Unique codehashes: 2               Fetched contracts: 0/5
Calls/s: 16719                 Corpus size: 7 seqs                Fetched slots: 0/0
Gas/s: 95699883                New coverage: 2s ago
Total calls: 50158/50000       Slither succeeded
                                    Tests (6) [*]
echidna_pair_marriage_state_coherent: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.marryOtherOneWay()
2. TestTaxpayer.divorce()

Traces:
call Taxpayer::getIsMarried()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:57)
 └← (true)
call Taxpayer::getSpouse()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:60)
 └← (TestTaxpayer)
                                    Log (12)
[2026-01-07 19:38:39.49] [Worker 1] Test limit reached. Stopping.
[2026-01-07 19:38:37.83] [Worker 3] Test limit reached. Stopping.
[2026-01-07 19:38:37.79] [Worker 2] Test limit reached. Stopping.
[2026-01-07 19:38:37.77] [Worker 0] Test limit reached. Stopping.
[2026-01-07 19:38:36.56] [Worker 3] New coverage: 3387 instr, 2 contracts, 7 seqs in corpus
[2026-01-07 19:38:36.49] [Worker 3] New coverage: 3385 instr, 2 contracts, 6 seqs in corpus
[2026-01-07 19:38:36.41] [Worker 3] New coverage: 3348 instr, 2 contracts, 5 seqs in corpus

                          Campaign complete, C-c or esc to exit
```

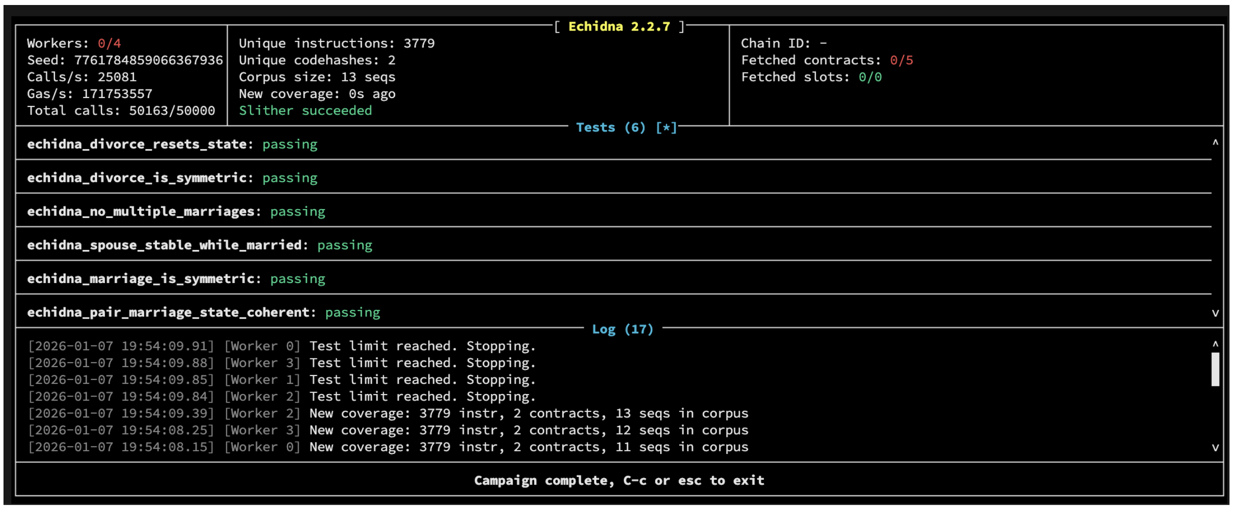
The issue arises because the `divorce` function only resets the local state of the caller and does not update the spouse's state. As a result, the marriage relationship becomes asymmetric after a unilateral divorce.

Root cause The root cause is that the divorce operation updates only the local taxpayer state. The spouse state is left unchanged, allowing asymmetric marriage configurations to persist after divorce.

Fix applied The divorce operation was modified to enforce symmetric state updates. Similarly to the marriage procedure, an auxiliary function has been created. When a taxpayer initiates a divorce, the spouse contract is explicitly involved to clear the marriage state on both sides, preventing the reachability of inconsistent configurations.

This ensures that the divorce operation preserves global coherence and that no intermediate asymmetric state can be observed.

Verification after the fix After applying the fix, Echidna was no longer able to generate any execution trace that violates the invariant.



```
[ Echidna 2.2.7 ]
Workers: 0/4                   Unique instructions: 3779          Chain ID: -
Seed: 7761784859066367936      Unique codehashes: 2               Fetched contracts: 0/5
Calls/s: 25081                 Corpus size: 13 seqs               Fetched slots: 0/0
Gas/s: 171753557               New coverage: 0s ago
Total calls: 50163/50000       Slither succeeded
                                    Tests (6) [*]
echidna_divorce_resets_state: passing                                                    ^

echidna_divorce_is_symmetric: passing

echidna_no_multiple_marriages: passing

echidna_spouse_stable_while_married: passing

echidna_marriage_is_symmetric: passing

echidna_pair_marriage_state_coherent: passing                                            v
                                    Log (17)
[2026-01-07 19:54:09.91] [Worker 0] Test limit reached. Stopping.                        ^
[2026-01-07 19:54:09.88] [Worker 3] Test limit reached. Stopping.
[2026-01-07 19:54:09.85] [Worker 1] Test limit reached. Stopping.
[2026-01-07 19:54:09.84] [Worker 2] Test limit reached. Stopping.
[2026-01-07 19:54:09.39] [Worker 2] New coverage: 3779 instr, 2 contracts, 13 seqs in corpus
[2026-01-07 19:54:08.25] [Worker 3] New coverage: 3779 instr, 2 contracts, 12 seqs in corpus
[2026-01-07 19:54:08.15] [Worker 0] New coverage: 3779 instr, 2 contracts, 11 seqs in corpus  v

                          Campaign complete, C-c or esc to exit
```

3 Part 2

3.1 Invariant 4: Authorization of tax allowance updates

Specification Only authorized entities should be allowed to modify the tax allowance of a taxpayer. In the intended model, these entities are the taxpayer itself, its spouse, and the registered lottery contract.

Property encoding This is an authorization property: an unauthorized caller must not be able to trigger state transitions that are reserved to authorized entities. In property-based fuzzing, access-control violations are typically detected through their *effects* on observable state rather than by checking for explicit reverts.

Instead of asserting that a specific call fails, the test harness models an explicitly unauthorized caller and checks that no forbidden state becomes reachable. In this project, the forbidden effect is the ability to arbitrarily modify the tax allowance.

Test harness snippet Unauthorized calls are modeled via a dedicated attacker contract that directly invokes `setTaxAllowance` on a victim taxpayer:

```
// AttackerTaxpayer (defined in TestTaxpayer.sol)
contract AttackerTaxpayer {
    function isContract() external pure returns (bool) {
        return true;
    }

    function attackSetAllowance(address victim, uint256 x) external {
        Taxpayer(victim).setTaxAllowance(x);
    }
}
```

The Echidna harness exposes this behavior as an action that can be freely interleaved with the other operations:

```
// TestTaxpayer.sol
function act_attack_set_allowance(uint x) external {
    attacker.attackSetAllowance(address(this), x);
}
```

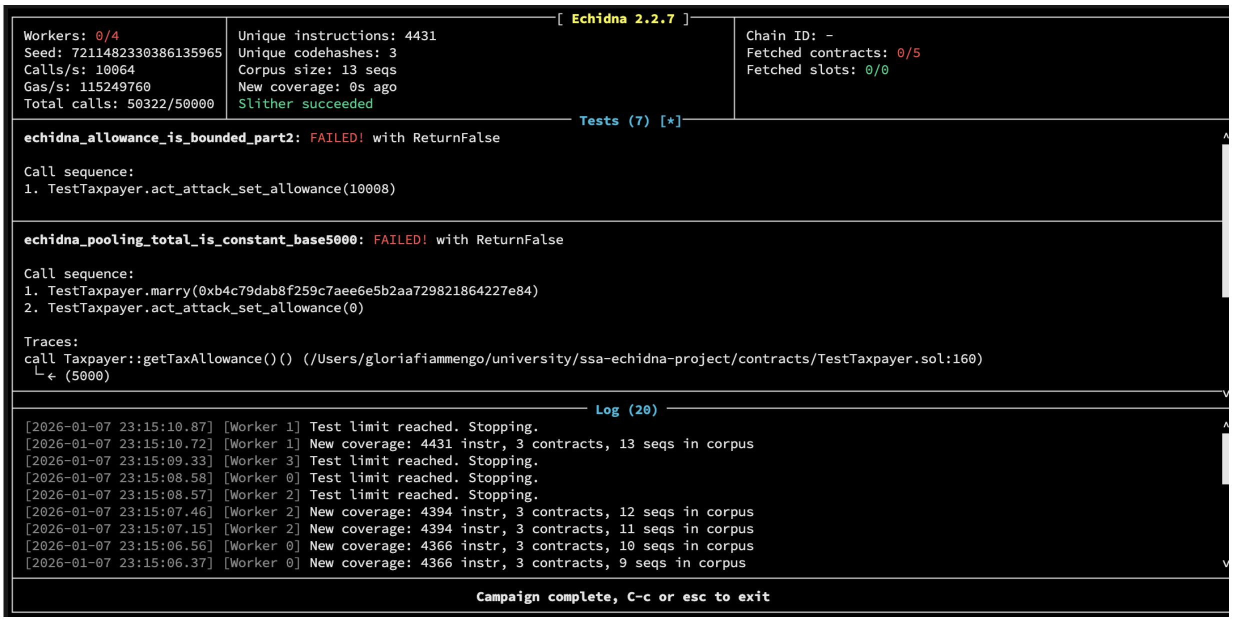
The authorization requirement is then checked indirectly through a state-based invariant. In the Part 2 model (before introducing age-based allowances), the maximum tax allowance that a single taxpayer can legitimately reach is bounded. Any execution that exceeds this bound necessarily implies that an unauthorized update has occurred.

```
// TestTaxpayer.sol
function echidna_allowance_is_bounded_part_2() public view returns (bool) {
    return getTaxAllowance() <= DEFAULT_ALLOWANCE * 2;
}
```

This bound is conservative and intentionally scoped to Part 2. It does not encode the full pooling semantics, but acts as a detector for unauthorized state transitions caused by bypassing access control.

Counterexample Echidna found a violation of this invariant because `setTaxAllowance` did not correctly enforce access control. Authorization was initially based on trusting a caller-provided interface predicate (e.g., a contract claiming to be a valid taxpayer), which can be trivially forged.

Echidna produced a counterexample in which an attacker contract successfully invokes `setTaxAllowance`, causing an unauthorized modification of `tax_allowance`.



```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 4431          Chain ID: -
Seed: 7211482330386135965       Unique codehashes: 3               Fetched contracts: 0/5
Calls/s: 10064                  Corpus size: 13 seqs               Fetched slots: 0/0
Gas/s: 115249760                New coverage: 0s ago
Total calls: 50322/50000        Slither succeeded
Tests (7) [*]
echidna_allowance_is_bounded_part2: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.act_attack_set_allowance(10008)

echidna_pooling_total_is_constant_base5000: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.marry(0xb4c79dab8f259c7aee6e5b2aa729821864227e84)
2. TestTaxpayer.act_attack_set_allowance(0)

Traces:
call Taxpayer::getTaxAllowance()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:160)
 └← (5000)
Log (20)
[2026-01-07 23:15:10.87] [Worker 1] Test limit reached. Stopping.
[2026-01-07 23:15:10.72] [Worker 1] New coverage: 4431 instr, 3 contracts, 13 seqs in corpus
[2026-01-07 23:15:09.33] [Worker 3] Test limit reached. Stopping.
[2026-01-07 23:15:08.58] [Worker 0] Test limit reached. Stopping.
[2026-01-07 23:15:08.57] [Worker 2] Test limit reached. Stopping.
[2026-01-07 23:15:07.46] [Worker 2] New coverage: 4394 instr, 3 contracts, 12 seqs in corpus
[2026-01-07 23:15:07.15] [Worker 2] New coverage: 4394 instr, 3 contracts, 11 seqs in corpus
[2026-01-07 23:15:06.56] [Worker 0] New coverage: 4366 instr, 3 contracts, 10 seqs in corpus
[2026-01-07 23:15:06.37] [Worker 0] New coverage: 4366 instr, 3 contracts, 9 seqs in corpus

Campaign complete, C-c or esc to exit
```

Root cause The root cause is reliance on interface trust for authorization. The contract treated externally supplied contract behavior as proof of legitimacy, allowing malicious contracts to impersonate authorized entities and bypass access control.

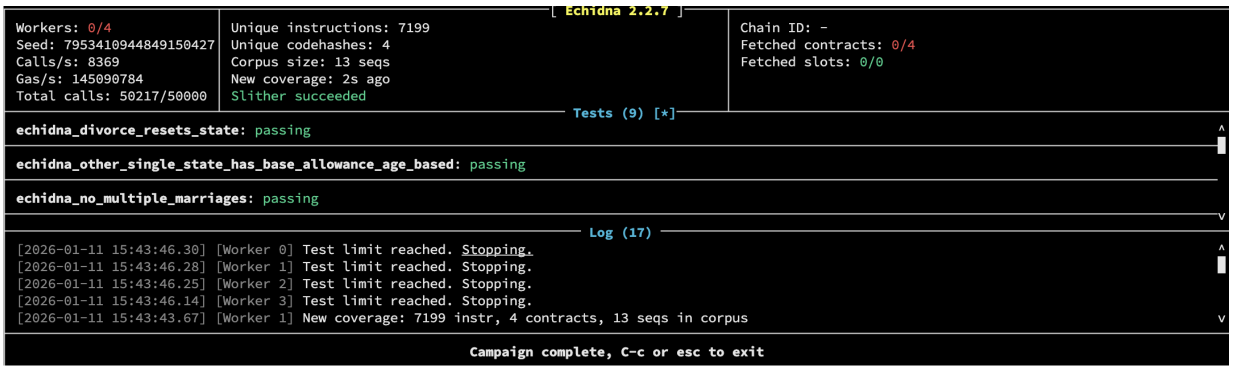
Fix applied The fix enforces explicit access control in `setTaxAllowance` by restricting callers to: (i) the taxpayer itself, (ii) its spouse, or (iii) the registered lottery contract.

```
// Taxpayer.sol
function setTaxAllowance(uint ta) public {
    require(
        msg.sender == address(this) ||
            msg.sender == spouse ||
            (lottery != address(0) && msg.sender == lottery),
        "not authorized"
    );

    tax_allowance = ta;
}
```

To prevent environment reconfiguration during fuzzing, the lottery reference is also restricted to a one-time initialization:

```
// Taxpayer.sol
function setLottery(address lot) public {
    require(msg.sender == admin, "only admin");
    require(lottery == address(0), "lottery already set");
    require(lot != address(0), "invalid lottery");
    require(lot.code.length > 0, "lottery must be a contract");

    lottery = lot;
}
```

Verification after the fix After applying the fix, calls performed through `AttackerTaxpayer` no longer produce any observable modification of `tax_allowance`. Echidna is unable to generate any execution trace that violates the authorization invariant.



```
                                        [ Echidna 2.2.7 ]
Workers: 0/4                  Unique instructions: 7199          Chain ID: -
Seed: 7953410944849150427     Unique codehashes: 4               Fetched contracts: 0/4
Calls/s: 8369                 Corpus size: 13 seqs               Fetched slots: 0/0
Gas/s: 145090784              New coverage: 2s ago
Total calls: 50217/50000      Slither succeeded
                                          Tests (9) [*]
echidna_divorce_resets_state: passing
echidna_other_single_state_has_base_allowance_age_based: passing
echidna_no_multiple_marriages: passing
                                            Log (17)
[2026-01-11 15:43:46.30] [Worker 0] Test limit reached. Stopping.
[2026-01-11 15:43:46.28] [Worker 1] Test limit reached. Stopping.
[2026-01-11 15:43:46.25] [Worker 2] Test limit reached. Stopping.
[2026-01-11 15:43:46.14] [Worker 3] Test limit reached. Stopping.
[2026-01-11 15:43:43.67] [Worker 1] New coverage: 7199 instr, 4 contracts, 13 seqs in corpus
                                Campaign complete, C-c or esc to exit
```

3.2 Invariant 5: Pooling operations are gated by marriage

Specification Pooling of tax allowance is only meaningful in the context of a valid marriage. Therefore, redistribution of tax allowance must not change any state while the taxpayer is single.

Property encoding This requirement is encoded as a stateful property, represented by the function `echidna_pooling_is_gated_by_marr`
When the taxpayer is not married, the current value of the tax allowance is snapshotted and the invariant checks that this value remains unchanged across subsequent executions. When a marriage occurs, the snapshot is reset and no constraint is imposed, allowing legitimate pooling operations to take place.

Test harness snippet The invariant is implemented as a snapshot-based Echidna property that tracks the value of the tax allowance while the taxpayer is single.

```
// validated property: pooling operations must have no effect while single
function echidna_pooling_is_gated_by_marriage() public returns (bool) {
    if (!getIsMarried()) {
        uint a = getTaxAllowance();

        if (!single_snapshot_taken) {
            single_allowance_snapshot = a;
            single_snapshot_taken = true;
            return true;
        }

        return a == single_allowance_snapshot;
    }

    single_snapshot_taken = false;
    single_allowance_snapshot = 0;
    return true;
}
```

When the taxpayer is single, the property records the first observed value of `tax_allowance` in a snapshot. For all subsequent executions while the taxpayer remains unmarried, the invariant enforces that the allowance remains equal to this snapshot value. Any change in `tax_allowance` while single causes the property to fail, making unauthorized pooling effects observable.

If a marriage occurs, the snapshot state is explicitly cleared and the invariant places no restriction on allowance changes, allowing legitimate pooling operations to take place only in the married state.

Root cause No root cause was identified, as the invariant holds after fixing the authorization logic.

Verification after the fix After fixing the access control vulnerability on `setTaxAllowance`, Echidna was no longer able to generate any execution trace that violates this property.

4 Part 3

4.1 Invariant 6: Age-based pooling of tax allowance

Specification According to the specification, the total tax allowance of a married couple must depend on the current age category of each taxpayer. Taxpayers aged 65 or older are entitled to a higher base allowance (`ALLOWANCE_OAP`).

Property encoding This invariant expresses a global constraint on tax allowance pooling based on age. According to the specification, the total allowance available to a married couple must equal the sum of the age-based base allowances of the two spouses. For single taxpayers, the allowance must never exceed the age-based base value.

Rather than forcing age progression explicitly, the property is formulated in a way that is independent of how the age threshold is reached. Echidna is free to explore executions where age changes occur naturally through calls to `haveBirthday()`, and the invariant checks that the allowance state is always consistent with the current ages and marital status.

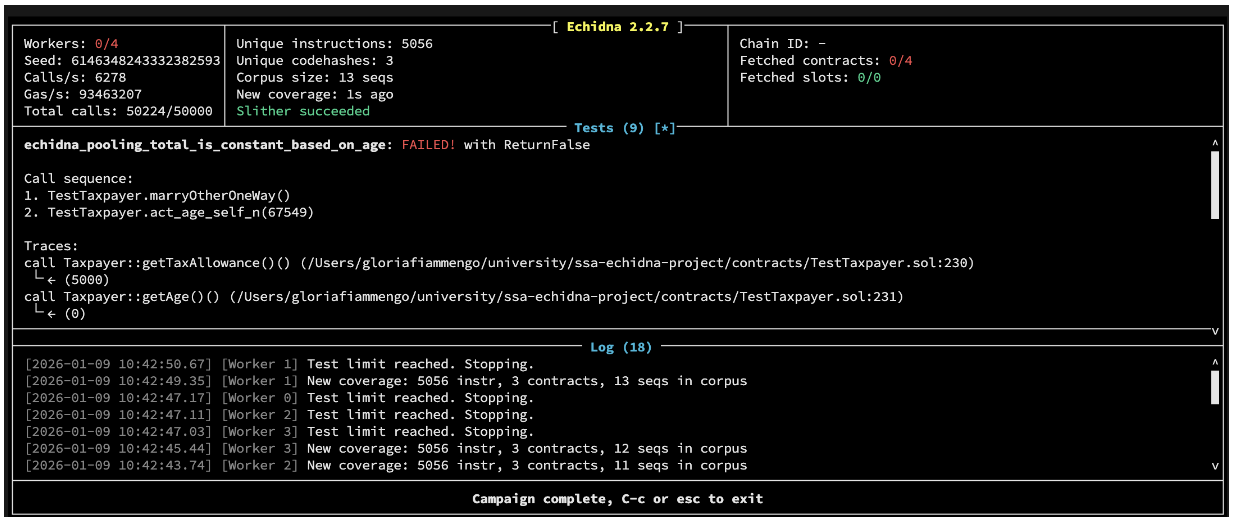
Test harness snippet The invariant is encoded as a state-based property that relates the observable tax allowances of the taxpayer and its spouse to their respective ages.

```
function echidna_pooling_total_is_constant_based_on_age()
    public
    view
    returns (bool)
{
    uint a = getTaxAllowance();
    uint base = (getAge() < 65) ? DEFAULT_ALLOWANCE : ALLOWANCE_OAP;

    if (!getIsMarried()) {
        return a <= base;
    }

    address sp = getSpouse();
    if (sp == address(0)) return false;

    ITaxpayerView t = ITaxpayerView(sp);
    uint b = t.getTaxAllowance();
    uint baseSpouse = (t.getAge() < 65) ? DEFAULT_ALLOWANCE : ALLOWANCE_OAP;

    if (a > type(uint).max - b) return false;
    return a + b == base + baseSpouse;
}
```

When the taxpayer is single, the property enforces that the tax allowance never exceeds the base value determined by the taxpayer's current age category. When the taxpayer is married, the invariant checks that: (i) a valid spouse exists, (ii) the combined allowance of the couple is well-defined (no arithmetic overflow), and (iii) the sum of the two allowances is exactly equal to the sum of the two age-based base allowances.

This formulation ensures that allowance pooling neither creates nor destroys value, and that any change in allowance is fully explained by age transitions or marital status changes. Any deviation from this constraint is observable as an invariant violation.

Counterexample After enabling age progression, Echidna found executions in which a taxpayer crossed the age threshold without a corresponding update of the tax allowance.

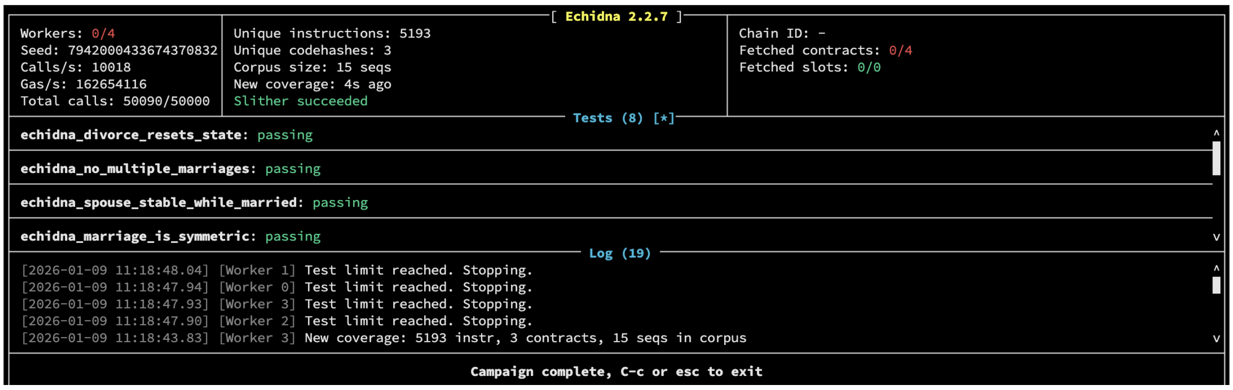


```
[ Echidna 2.2.7 ]
Workers: 0/4                  Unique instructions: 5056          Chain ID: -
Seed: 6146348243332382593     Unique codehashes: 3               Fetched contracts: 0/4
Calls/s: 6278                 Corpus size: 13 seqs               Fetched slots: 0/0
Gas/s: 93463207               New coverage: 1s ago
Total calls: 50224/50000      Slither succeeded
                                   Tests (9) [*]
echidna_pooling_total_is_constant_based_on_age: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.marryOtherOneWay()
2. TestTaxpayer.act_age_self_n(67549)

Traces:
call Taxpayer::getTaxAllowance()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:230)
 └← (5000)
call Taxpayer::getAge()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:231)
 └← (0)
                                   Log (18)
[2026-01-09 10:42:50.67] [Worker 1] Test limit reached. Stopping.
[2026-01-09 10:42:49.35] [Worker 1] New coverage: 5056 instr, 3 contracts, 13 seqs in corpus
[2026-01-09 10:42:47.17] [Worker 0] Test limit reached. Stopping.
[2026-01-09 10:42:47.11] [Worker 2] Test limit reached. Stopping.
[2026-01-09 10:42:47.03] [Worker 3] Test limit reached. Stopping.
[2026-01-09 10:42:45.44] [Worker 3] New coverage: 5056 instr, 3 contracts, 12 seqs in corpus
[2026-01-09 10:42:43.74] [Worker 2] New coverage: 5056 instr, 3 contracts, 11 seqs in corpus
                          Campaign complete, C-c or esc to exit
```

Root cause The root cause is that the function `haveBirthday` updates the age variable but does not consistently synchronize the tax allowance when the age threshold is crossed.

Fix applied The fix ensures that the tax allowance is updated when the taxpayer reaches the age of 65. The allowance adjustment is performed directly within `haveBirthday`, guaranteeing consistency between age and allowance.

Verification after the fix After applying the fix, Echidna was no longer able to generate any execution trace that violates the age-based pooling invariant.



```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 5193      Chain ID: -
Seed: 7942000433674370832       Unique codehashes: 3           Fetched contracts: 0/4
Calls/s: 10018                  Corpus size: 15 seqs           Fetched slots: 0/0
Gas/s: 162654116                New coverage: 4s ago
Total calls: 50090/50000        Slither succeeded
Tests (8) [*]
echidna_divorce_resets_state: passing
echidna_no_multiple_marriages: passing
echidna_spouse_stable_while_married: passing
echidna_marriage_is_symmetric: passing
Log (19)
[2026-01-09 11:18:48.04] [Worker 1] Test limit reached. Stopping.
[2026-01-09 11:18:47.94] [Worker 0] Test limit reached. Stopping.
[2026-01-09 11:18:47.93] [Worker 3] Test limit reached. Stopping.
[2026-01-09 11:18:47.90] [Worker 2] Test limit reached. Stopping.
[2026-01-09 11:18:43.83] [Worker 3] New coverage: 5193 instr, 3 contracts, 15 seqs in corpus
Campaign complete, C-c or esc to exit
```

4.2 Invariant 7: Age-based allowance restoration after divorce

Specification According to the specification, after a divorce each taxpayer must return to the age-based base allowance. Once the marriage is dissolved, the allowance of a single taxpayer must always correspond to the base value determined by the current age category.

Property encoding While allowance pooling invariants ensure correctness during marriage, they do not guarantee that pooled state is correctly undone after divorce. This invariant therefore focuses explicitly on the *post-divorce single state.*

Rather than encoding the entire scenario inside a single property, the test harness separates: (i) a dedicated *reachability action* that constructs a non-trivial married state with pooled allowance and then triggers a divorce, and (ii) stable invariants that must hold for any taxpayer who is single, regardless of how that state was reached.

Test harness snippet The scenario is made reachable through the following Echidna-callable action:

```
// reachability helper: marry -> positive transfer -> divorce
function act_scenario_dirty_then_divorce(uint x) external {
    if (!getIsMarried()) {
        marry(address(other));
    }

    uint a = getTaxAllowance();
    if (a > 0) {
        uint c = 1 + (x % a);
        transferAllowance(c);
    }

    divorce();
}
```

This action forces the system through a sequence where allowance pooling occurs before divorce, ensuring that any failure to restore the base allowance becomes observable.

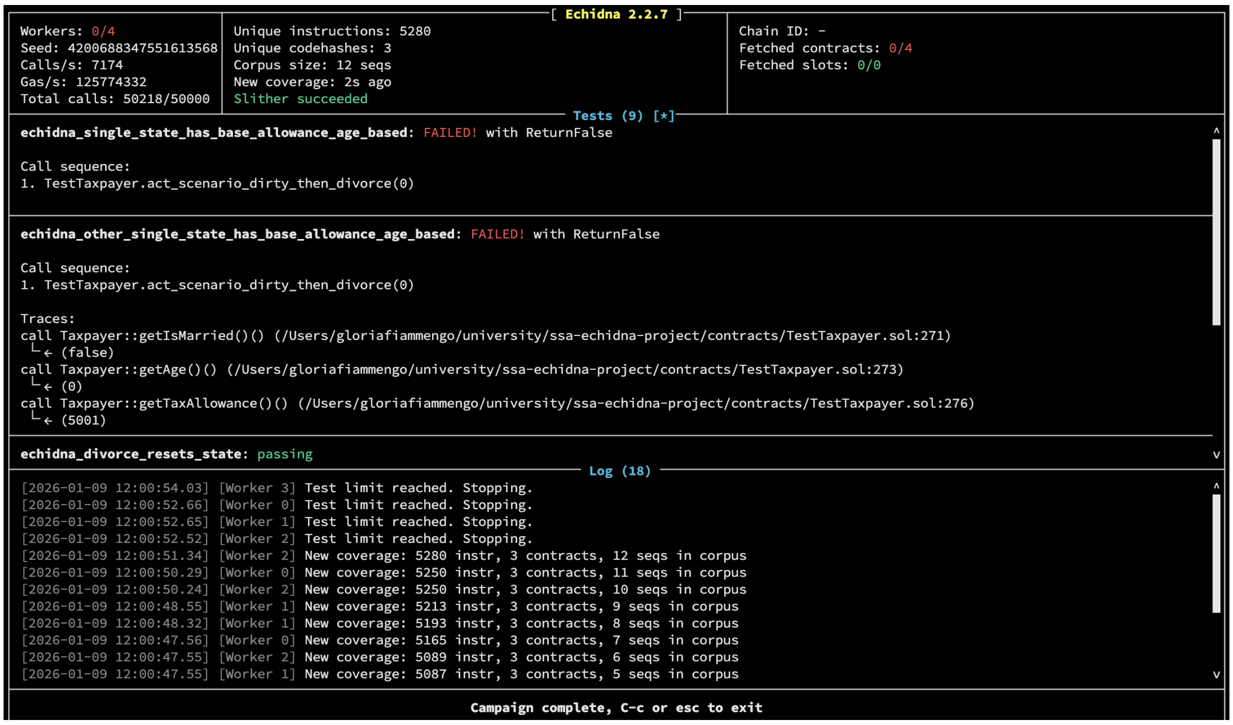
The correctness requirement is then enforced through the following invariants:

```
function echidna_single_state_has_base_allowance_age_based()
    public view returns (bool)
{
    if (getIsMarried()) return true;

    uint base = (getAge() < 65)
        ? DEFAULT_ALLOWANCE
        : ALLOWANCE_OAP;

    return getTaxAllowance() == base;
}

function echidna_other_single_state_has_base_allowance_age_based()
    public view returns (bool)
{
    if (other.getIsMarried()) return true;

    uint base = (other.getAge() < 65)
```

```
        ? DEFAULT_ALLOWANCE
        : ALLOWANCE_OAP;

    return other.getTaxAllowance() == base;
}
```

These properties assert that, whenever a taxpayer is single, its tax allowance must exactly match the age-based base value. Any residual pooled or elevated allowance after divorce violates the invariant.

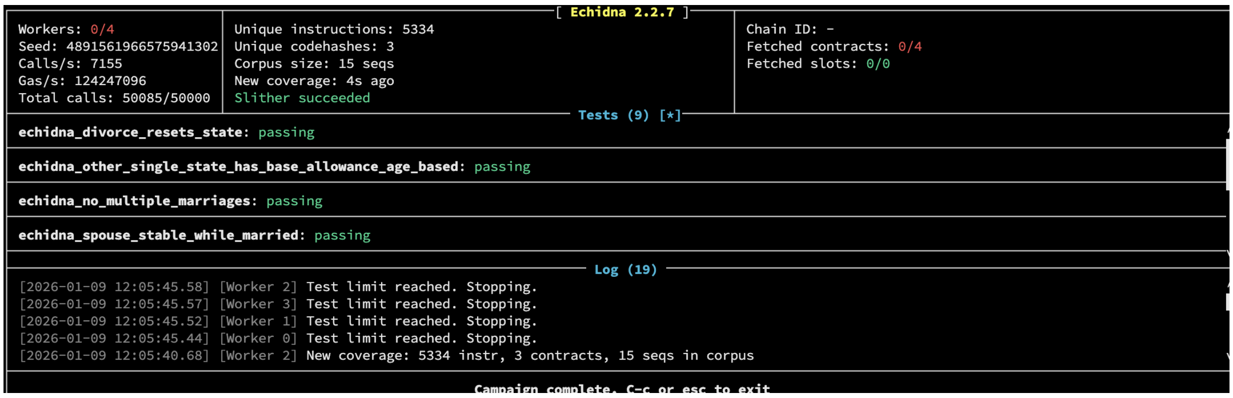
Counterexample Echidna found executions in which, after divorce, one or both taxpayers remained single with a non-base allowance, indicating that pooled state was not correctly undone.



```
[ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 5280          Chain ID: -
Seed: 4200688347551613568    Unique codehashes: 3               Fetched contracts: 0/4
Calls/s: 7174                Corpus size: 12 seqs               Fetched slots: 0/0
Gas/s: 125774332             New coverage: 2s ago
Total calls: 50218/50000     Slither succeeded
                                   Tests (9) [*]
echidna_single_state_has_base_allowance_age_based: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.act_scenario_dirty_then_divorce(0)

echidna_other_single_state_has_base_allowance_age_based: FAILED! with ReturnFalse

Call sequence:
1. TestTaxpayer.act_scenario_dirty_then_divorce(0)

Traces:
call Taxpayer::getIsMarried()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:271)
 └← (false)
call Taxpayer::getAge()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:273)
 └← (0)
call Taxpayer::getTaxAllowance()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestTaxpayer.sol:276)
 └← (5001)

echidna_divorce_resets_state: passing
                                   Log (18)
[2026-01-09 12:00:54.03] [Worker 3] Test limit reached. Stopping.
[2026-01-09 12:00:52.66] [Worker 0] Test limit reached. Stopping.
[2026-01-09 12:00:52.65] [Worker 1] Test limit reached. Stopping.
[2026-01-09 12:00:52.52] [Worker 2] Test limit reached. Stopping.
[2026-01-09 12:00:51.34] [Worker 2] New coverage: 5280 instr, 3 contracts, 12 seqs in corpus
[2026-01-09 12:00:50.29] [Worker 0] New coverage: 5250 instr, 3 contracts, 11 seqs in corpus
[2026-01-09 12:00:50.24] [Worker 2] New coverage: 5250 instr, 3 contracts, 10 seqs in corpus
[2026-01-09 12:00:48.55] [Worker 1] New coverage: 5213 instr, 3 contracts, 9 seqs in corpus
[2026-01-09 12:00:48.32] [Worker 1] New coverage: 5193 instr, 3 contracts, 8 seqs in corpus
[2026-01-09 12:00:47.56] [Worker 0] New coverage: 5165 instr, 3 contracts, 7 seqs in corpus
[2026-01-09 12:00:47.55] [Worker 2] New coverage: 5089 instr, 3 contracts, 6 seqs in corpus
[2026-01-09 12:00:47.55] [Worker 1] New coverage: 5087 instr, 3 contracts, 5 seqs in corpus

                         Campaign complete, C-c or esc to exit
```

Fix applied The divorce logic was strengthened to explicitly restore the age-based base allowance for both parties. Both `divorce` and the corresponding spouse-side cleanup ensure that, upon divorce, each taxpayer's allowance is reset according to its current age category.

Verification after the fix After applying the fix, the post-divorce single-state invariants always hold. Echidna is no longer able to generate any execution trace in which a divorced taxpayer retains a pooled or non-base allowance.



```
[ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 5334          Chain ID: -
Seed: 4891561966575941302    Unique codehashes: 3               Fetched contracts: 0/4
Calls/s: 7155                Corpus size: 15 seqs               Fetched slots: 0/0
Gas/s: 124247096             New coverage: 4s ago
Total calls: 50085/50000     Slither succeeded
                                   Tests (9) [*]
echidna_divorce_resets_state: passing

echidna_other_single_state_has_base_allowance_age_based: passing

echidna_no_multiple_marriages: passing

echidna_spouse_stable_while_married: passing
                                   Log (19)
[2026-01-09 12:05:45.58] [Worker 2] Test limit reached. Stopping.
[2026-01-09 12:05:45.57] [Worker 3] Test limit reached. Stopping.
[2026-01-09 12:05:45.52] [Worker 1] Test limit reached. Stopping.
[2026-01-09 12:05:45.44] [Worker 0] Test limit reached. Stopping.
[2026-01-09 12:05:40.68] [Worker 2] New coverage: 5334 instr, 3 contracts, 15 seqs in corpus

                         Campaign complete, C-c or esc to exit
```

5 Part 4

5.1 Invariant 8: Commitment immutability in commit/reveal lottery

Specification In a commit/reveal lottery, each participant must commit exactly one value per round. Once a participant has submitted a commitment, that commitment must remain immutable until the round ends. Allowing a participant to overwrite their commitment would enable adaptive behavior and invalidate the fairness guarantees of the protocol.

Property encoding This requirement is encoded as an Echidna property stating that, once a participant has committed a value in a given round, any subsequent attempt to commit again must not modify the stored commitment. The property is implemented as an active probe: after detecting that a commitment exists for a given participant, the test attempts to overwrite it with a different value and checks that the stored commitment remains unchanged.

Test harness snippet The immutability requirement is tested through a helper-based Echidna property. The public Echidna entry point delegates the check to an internal helper that performs an overwrite probe on a given participant.

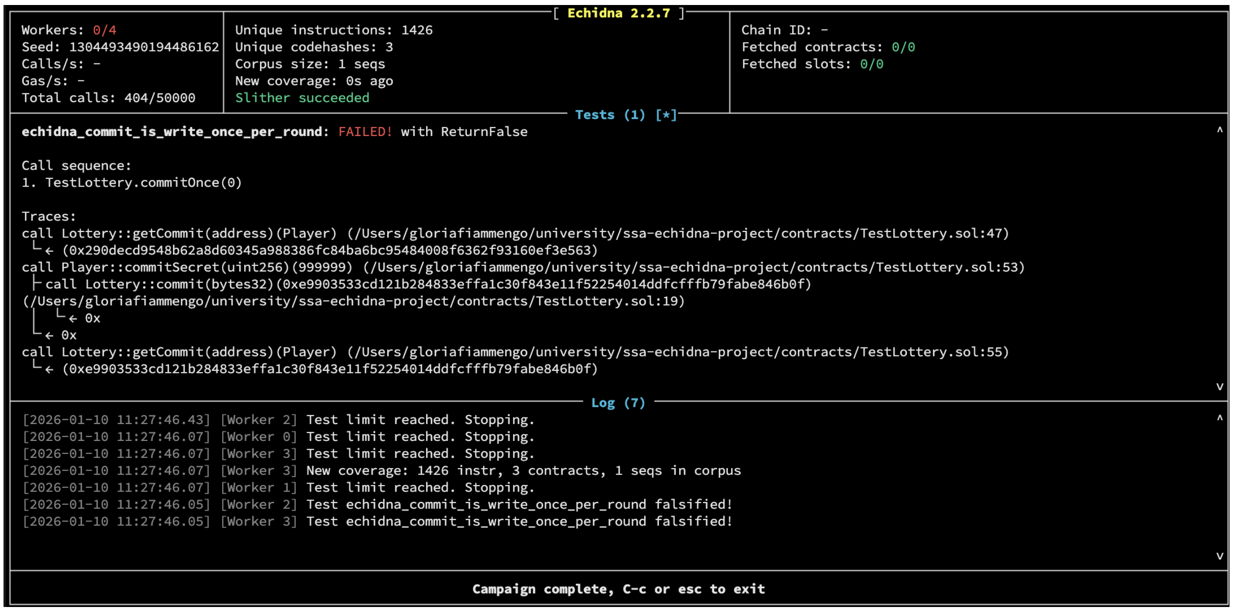
```
function echidna_commit_is_write_once_per_round() public returns (bool) {
    return _checkCommitWriteOnce(player1);
}
```

The core logic is implemented in `_checkCommitWriteOnce`:

```
function _checkCommitWriteOnce(Player p) internal returns (bool) {
    bytes32 before = lot.getCommit(address(p));

    // nothing committed yet => nothing to enforce
    if (before == bytes32(0)) return true;

    // overwrite probe: revert is fine, success must not change the stored commit
    try p.commitSecret(999999) {
        return lot.getCommit(address(p)) == before;
    } catch {
        return lot.getCommit(address(p)) == before;
    }
}
```

The helper first reads the current commitment for the participant through `lot.getCommit(address(p))`. If no commitment exists yet in the current round (`before == 0`), the property does not impose any constraint and returns `true`. Otherwise, it performs an overwrite attempt by calling `p.commitSecret(999999)`. Both outcomes are handled: if the call succeeds, the stored commitment must remain equal to `before`; if the call reverts, the commitment must still remain equal to `before`. Therefore, the invariant is violated if and only if a second commit changes the stored value within the same round.

Counterexample Echidna produces a counterexample consisting of a single commitment followed by an overwrite attempt. Since the `commit` function does not enforce immutability, the second commitment replaces the original one, violating the write-once requirement.



```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 1426            Chain ID: -
Seed: 1304493490194486162       Unique codehashes: 3                 Fetched contracts: 0/0
Calls/s: -                      Corpus size: 1 seqs                  Fetched slots: 0/0
Gas/s: -                        New coverage: 0s ago
Total calls: 404/50000          Slither succeeded
                                Tests (1) [*]
echidna_commit_is_write_once_per_round: FAILED! with ReturnFalse

Call sequence:
1. TestLottery.commitOnce(0)

Traces:
call Lottery::getCommit(address)(Player) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:47)
 └← (0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563)
call Player::commitSecret(uint256)(999999) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:53)
 ├call Lottery::commit(bytes32)(0xe9903533cd121b284833effa1c30f843e11f52254014ddfcfffb79fabe846b0f)
(/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:19)
 │   └← 0x
 └← 0x
call Lottery::getCommit(address)(Player) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:55)
 └← (0xe9903533cd121b284833effa1c30f843e11f52254014ddfcfffb79fabe846b0f)
                                Log (7)
[2026-01-10 11:27:46.43] [Worker 2] Test limit reached. Stopping.
[2026-01-10 11:27:46.07] [Worker 0] Test limit reached. Stopping.
[2026-01-10 11:27:46.07] [Worker 3] Test limit reached. Stopping.
[2026-01-10 11:27:46.07] [Worker 3] New coverage: 1426 instr, 3 contracts, 1 seqs in corpus
[2026-01-10 11:27:46.07] [Worker 1] Test limit reached. Stopping.
[2026-01-10 11:27:46.05] [Worker 2] Test echidna_commit_is_write_once_per_round falsified!
[2026-01-10 11:27:46.05] [Worker 3] Test echidna_commit_is_write_once_per_round falsified!

                        Campaign complete, C-c or esc to exit
```

The failing execution shows that: (i) a participant successfully submits an initial commitment; (ii) the same participant submits a second commitment within the same round; (iii) the stored commitment is overwritten with the new value.

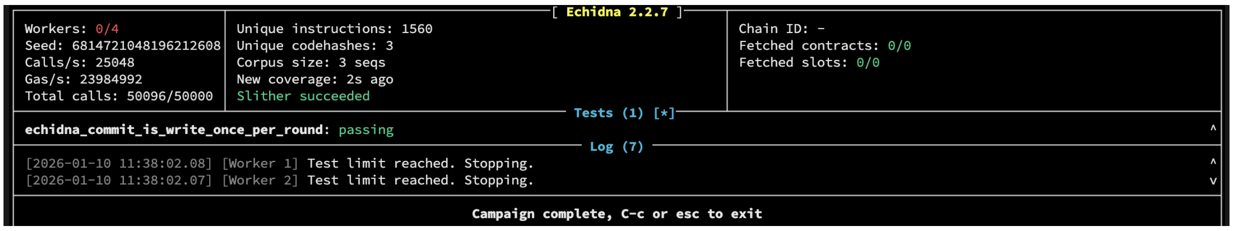
Root cause The root cause of the violation is the absence of a write-once constraint in the `commit` function. The contract does not check whether a commitment has already been recorded for the sender in the current round, and therefore allows arbitrary overwriting of the stored value.

Fix applied The `commit` function was modified to enforce write-once semantics by rejecting any second commit from the same address during an active round. Concretely, a guard is added to require that the caller has not already committed:

```
require(commits[msg.sender] == bytes32(0), "already committed");
```

This prevents commitment overwriting and restores the intended commit/reveal behavior.

Verification after the fix After applying the fix, the Echidna probe property that attempts to overwrite an existing commitment no longer finds any counterexample. Any overwrite attempt either reverts or leaves the stored commitment unchanged, and the property holds across all explored executions.



```
[ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 1560          Chain ID: -
Seed: 6814721048196212608    Unique codehashes: 3               Fetched contracts: 0/0
Calls/s: 25048               Corpus size: 3 seqs                Fetched slots: 0/0
Gas/s: 23984992              New coverage: 2s ago
Total calls: 50096/50000     Slither succeeded
                                   Tests (1) [*]
echidna_commit_is_write_once_per_round: passing                                   ^
                                   Log (7)
[2026-01-10 11:38:02.08] [Worker 1] Test limit reached. Stopping.                 ^
[2026-01-10 11:38:02.07] [Worker 2] Test limit reached. Stopping.                 v

                         Campaign complete, C-c or esc to exit
```

5.2 Invariant 9: Reveal is write-once per round

Specification In a commit/reveal lottery, each participant must be allowed to reveal at most once per round. After a valid reveal, any further reveal attempts by the same participant must not modify the lottery state. In particular, the same address must not be inserted multiple times into the list of revealed participants.

This requirement prevents a participant from obtaining multiple "tickets" in the winner selection by repeatedly revealing the same commitment.

Property encoding The invariant is encoded as an Echidna property enforcing that a participant can successfully reveal at most once per round. After a first successful reveal, any subsequent reveal attempt by the same participant in the same round must either revert or produce no observable state changes. The observable effect used to detect violations is the length of the `revealed` list, which must increase exactly once per participant per round.

Test harness snippet The property is implemented through a helper-based check. The public Echidna entry point delegates the verification to an internal helper that performs a controlled two-step reveal attempt for a given participant.

```
function echidna_reveal_is_write_once_per_round() public returns (bool) {
    return _checkRevealWriteOnce(player1, lastSecret1, hasSecret1);
}
```

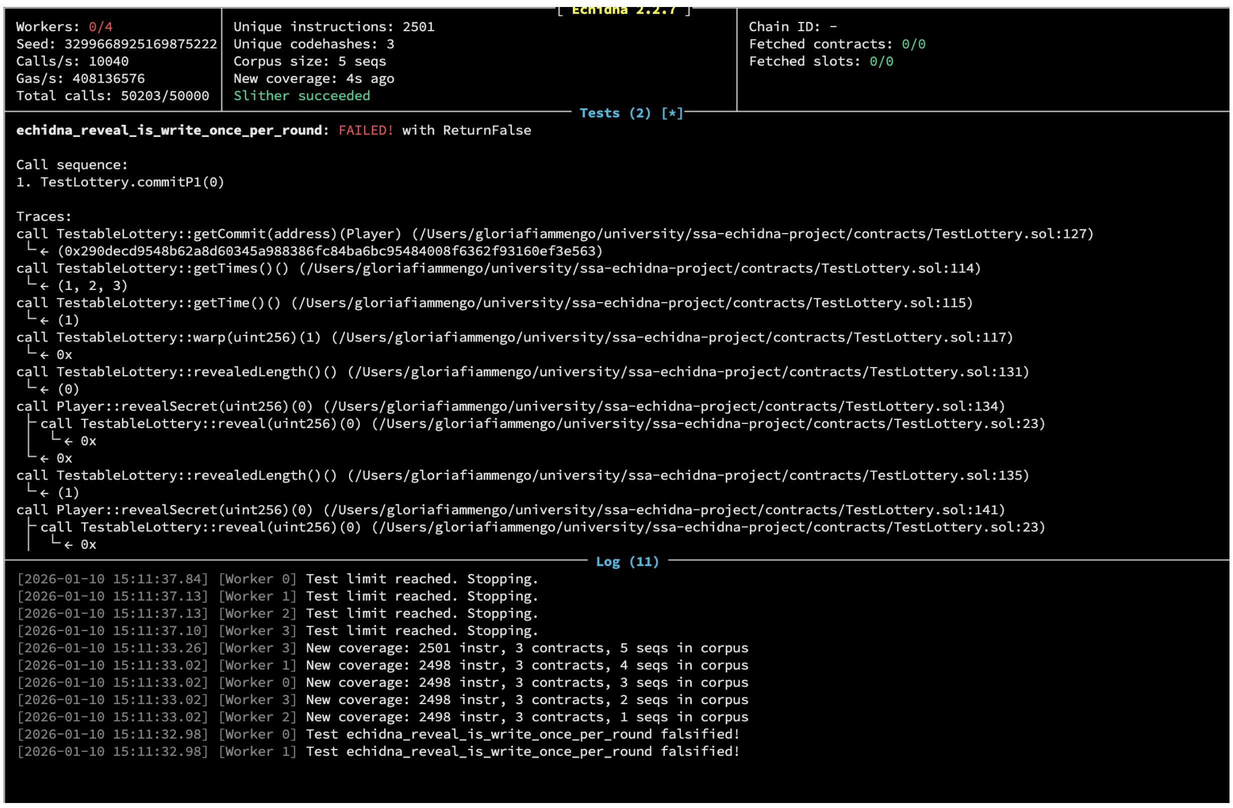
The core logic is implemented in the helper function:

```
function _checkRevealWriteOnce(
    Player p,
    uint256 sec,
    bool hasSec
) internal returns (bool) {
    if (!hasSec) return true;
    if (lot.getCommit(address(p)) == bytes32(0)) return true;

    _warpToReveal();

    uint256 lenBefore = lot.revealedLength();

    // first reveal attempt with the matching secret
    try p.revealSecret(sec) {
        uint256 lenAfterFirst = lot.revealedLength();

        // on success, exactly one ticket must be added
        if (lenAfterFirst != lenBefore + 1) return false;

        // second reveal must not add another ticket (revert ok)
        try p.revealSecret(sec) {
            return lot.revealedLength() == lenAfterFirst;
        } catch {
            return lot.revealedLength() == lenAfterFirst;
        }
    } catch {
        // if the first reveal never succeeded, write-once cannot be enforced yet
        return true;
    }
}
```

The helper first checks that the participant actually owns a secret and has previously committed. It then warps time into the reveal phase and records the initial length of the `revealed` list. If the first reveal succeeds, the property requires that exactly one entry is added. Any second reveal attempt in the same round must not increase the list length, regardless of whether the call reverts or returns normally. Therefore, the invariant is violated if and only if a participant is able to obtain more than one entry in the `revealed` list within the same round.

Counterexample In the pre-fix version of the contract, Echidna falsified the property by generating an execution in which a participant: (i) commits a secret; (ii) successfully reveals it; (iii) reveals the same secret a second time in the same round.

After the second reveal, the length of the `revealed` array increases again, showing that the same address is inserted multiple times.



```
                                                               [ Echidna 2.2.7 ]
Workers: 0/4                   Unique instructions: 2501                 Chain ID: -
Seed: 3299668925169875222      Unique codehashes: 3                      Fetched contracts: 0/0
Calls/s: 10040                 Corpus size: 5 seqs                       Fetched slots: 0/0
Gas/s: 408136576               New coverage: 4s ago
Total calls: 50203/50000       Slither succeeded
                                                                Tests (2) [*]
echidna_reveal_is_write_once_per_round: FAILED! with ReturnFalse

Call sequence:
1. TestLottery.commitP1(0)

Traces:
call TestableLottery::getCommit(address)(Player) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:127)
 └← (0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563)
call TestableLottery::getTimes()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:114)
 └← (1, 2, 3)
call TestableLottery::getTime()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:115)
 └← (1)
call TestableLottery::warp(uint256)(1) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:117)
 └← 0x
call TestableLottery::revealedLength()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:131)
 └← (0)
call Player::revealSecret(uint256)(0) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:134)
 ├call TestableLottery::reveal(uint256)(0) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:23)
 │ └← 0x
 └← 0x
call TestableLottery::revealedLength()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:135)
 └← (1)
call Player::revealSecret(uint256)(0) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:141)
 ├call TestableLottery::reveal(uint256)(0) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:23)
 │ └← 0x
                                                                 Log (11)
[2026-01-10 15:11:37.84] [Worker 0] Test limit reached. Stopping.
[2026-01-10 15:11:37.13] [Worker 1] Test limit reached. Stopping.
[2026-01-10 15:11:37.13] [Worker 2] Test limit reached. Stopping.
[2026-01-10 15:11:37.10] [Worker 3] Test limit reached. Stopping.
[2026-01-10 15:11:33.26] [Worker 3] New coverage: 2501 instr, 3 contracts, 5 seqs in corpus
[2026-01-10 15:11:33.02] [Worker 1] New coverage: 2498 instr, 3 contracts, 4 seqs in corpus
[2026-01-10 15:11:33.02] [Worker 0] New coverage: 2498 instr, 3 contracts, 3 seqs in corpus
[2026-01-10 15:11:33.02] [Worker 3] New coverage: 2498 instr, 3 contracts, 2 seqs in corpus
[2026-01-10 15:11:33.02] [Worker 2] New coverage: 2498 instr, 3 contracts, 1 seqs in corpus
[2026-01-10 15:11:32.98] [Worker 0] Test echidna_reveal_is_write_once_per_round falsified!
[2026-01-10 15:11:32.98] [Worker 1] Test echidna_reveal_is_write_once_per_round falsified!
```

Root cause The `reveal()` function did not enforce a write-once constraint per participant. As a consequence, every successful call executed `revealed.push(msg.sender)`, allowing the same address to appear multiple times in the `revealed` array.

Fix applied The fix introduces an explicit guard for every participant, using a dedicated state variable:

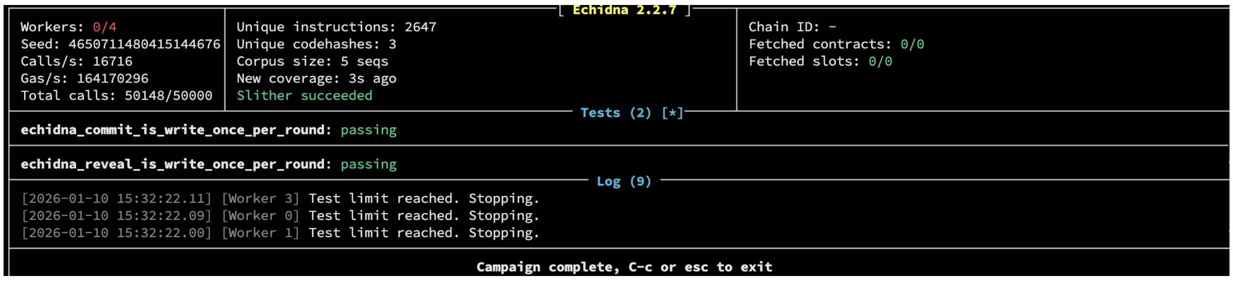
```
mapping(address => bool) hasRevealed;
```

The `reveal()` function is updated to reject a second reveal in the same round:

```
require(!hasRevealed[msg.sender], "already revealed");
hasRevealed[msg.sender] = true;
revealed.push(msg.sender);
```

To keep rounds independent, the flag is reset during round cleanup in `endLottery()` together with the other per-round state, by iterating over the tracked `committed[]` participants and setting `hasRevealed[a] = false`, then clearing both `committed` and `revealed` arrays.

Verification after the fix After applying the fix, repeated reveal attempts by the same participant in the same round revert or produce no observable effects. In particular, the length of the `revealed` array does not increase after the first successful reveal. Echidna is no longer able to falsify the property.



```
                                                               [ Echidna 2.2.7 ]
Workers: 0/4                   Unique instructions: 2647                 Chain ID: -
Seed: 4650711480415144676      Unique codehashes: 3                      Fetched contracts: 0/0
Calls/s: 16716                 Corpus size: 5 seqs                       Fetched slots: 0/0
Gas/s: 164170296               New coverage: 3s ago
Total calls: 50148/50000       Slither succeeded
                                                                Tests (2) [*]
echidna_commit_is_write_once_per_round: passing

echidna_reveal_is_write_once_per_round: passing
                                                                 Log (9)
[2026-01-10 15:32:22.11] [Worker 3] Test limit reached. Stopping.
[2026-01-10 15:32:22.09] [Worker 0] Test limit reached. Stopping.
[2026-01-10 15:32:22.00] [Worker 1] Test limit reached. Stopping.

                                          Campaign complete, C-c or esc to exit
```

5.3 Invariant 10: Safe termination with zero reveals

Specification Ending a lottery round must be safe even when no participant reveals a secret. In particular, calling `endLottery()` after the end time must not revert due to an empty `revealed` list.

This invariant prevents a denial-of-service scenario in which the lottery cannot be closed if no valid reveal is provided.

Property encoding The invariant is encoded as an Echidna property that checks that calling `endLottery()` does not revert when no participant has revealed a secret. The property does not enforce any constraint on winner selection, but only verifies that round termination is always possible, even in the absence of valid reveals.

Rather than asserting the absence of a revert indirectly through state changes, this property explicitly treats a revert of `endLottery()` as a violation.

Test harness snippet The property is implemented by forcing the system into a state where the round has ended and no participant has revealed, and then invoking `endLottery()` inside a `try/catch` block.

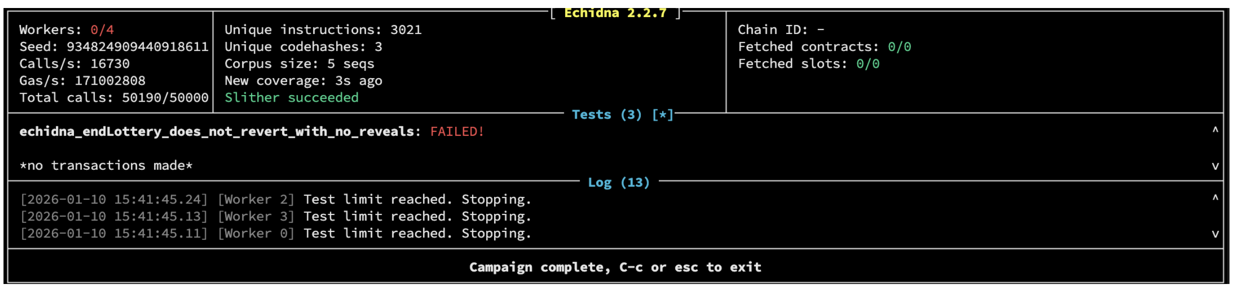
```
function echidna_endLottery_does_not_revert_with_no_reveals()
    public
    returns (bool)
{
    _warpToEnd();

    // only meaningful if nobody revealed yet
    if (lot.revealedLength() != 0) return true;

    try lot.endLottery() {
        return true;
    } catch {
        return false;
    }
}
```

The helper `_warpToEnd()` advances the observable time beyond the end of the current round. The property is only enforced when the `revealed` list is empty; if at least one participant has revealed, the check is vacuously satisfied.

The invariant holds if and only if `endLottery()` completes successfully without reverting when `revealedLength() == 0`. Any revert in this situation is treated as a violation, exposing a potential denial-of-service condition where the lottery round cannot be closed.

Counterexample In the pre-fix version of the contract, Echidna produced an execution in which: (i) the lottery is started; (ii) time advances beyond the end of the round; (iii) no participant reveals a secret; (iv) `endLottery()` is invoked.

The call reverts due to a division-by-zero error when computing the winner index, since `revealed.length == 0`.

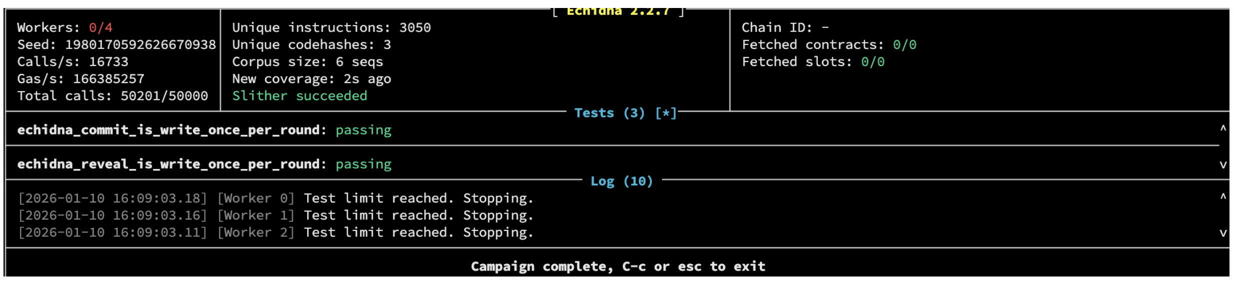


```
                                   [ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 3021        Chain ID: -
Seed: 934824909440918611        Unique codehashes: 3             Fetched contracts: 0/0
Calls/s: 16730                  Corpus size: 5 seqs              Fetched slots: 0/0
Gas/s: 171002808                New coverage: 3s ago
Total calls: 50190/50000        Slither succeeded
                                   Tests (3) [*]
echidna_endLottery_does_not_revert_with_no_reveals: FAILED!                        ^

*no transactions made*                                                             v
                                   Log (13)
[2026-01-10 15:41:45.24] [Worker 2] Test limit reached. Stopping.                  ^
[2026-01-10 15:41:45.13] [Worker 3] Test limit reached. Stopping.
[2026-01-10 15:41:45.11] [Worker 0] Test limit reached. Stopping.                  v

                          Campaign complete, C-c or esc to exit
```

Root cause The pre-fix implementation of `endLottery()` assumed that at least one participant had revealed. The winner index was computed using an expression of the form `total % revealed.length`. When `revealed.length == 0`, the modulo operation triggers a division-by-zero revert, making it impossible to terminate the round.

Fix applied The fix explicitly handles the empty-reveal case. In the final implementation, `endLottery()` checks whether any participant revealed before attempting winner selection. If `revealed.length == 0`, the function skips winner computation and proceeds directly with the normal round cleanup. Cleanup logic is executed regardless of whether a winner is selected, ensuring that the round always terminates safely.

Verification after the fix After applying the fix, `endLottery()` no longer reverts when `revealed.length == 0`. Echidna is unable to generate any execution trace that violates this invariant. The round can always be closed, eliminating the denial-of-service condition.



```
                                   [ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 3050        Chain ID: -
Seed: 1980170592626670938       Unique codehashes: 3             Fetched contracts: 0/0
Calls/s: 16733                  Corpus size: 6 seqs              Fetched slots: 0/0
Gas/s: 166385257                New coverage: 2s ago
Total calls: 50201/50000        Slither succeeded
                                   Tests (3) [*]
echidna_commit_is_write_once_per_round: passing                                    ^

echidna_reveal_is_write_once_per_round: passing                                    v
                                   Log (10)
[2026-01-10 16:09:03.18] [Worker 0] Test limit reached. Stopping.                  ^
[2026-01-10 16:09:03.16] [Worker 1] Test limit reached. Stopping.
[2026-01-10 16:09:03.11] [Worker 2] Test limit reached. Stopping.                  v

                          Campaign complete, C-c or esc to exit
```

5.4 Invariant 11: A new round must start from a clean state

Specification Each lottery round must be independent. After a round is closed, starting a new round must reset all round-specific state, so that no stale commitments, reveal flags, or participant lists leak across rounds.

Property encoding This invariant is encoded as a multi-phase stateful property that checks both round cleanup and restartability, without assuming that round termination and round restart happen in the same execution step.

The property is decomposed into three observable phases. First, after a successful invocation of `endLottery()`, the harness checks that the contract reaches a clean post-round state. Second, this clean state is recorded. Third, a subsequent attempt to start a new round must succeed.

To avoid false positives, checks are armed only after a real and successful round closure.

Test harness snippet A clean post-round state is defined by an auxiliary predicate that inspects only observable round-specific state:

```
function _isCleanClosedState() internal view returns (bool) {
    bool ok = true;

    // arrays must be empty after cleanup
    ok = ok && (lot.revealedLength() == 0);
    ok = ok && (lot.committedLength() == 0);

    // commits cleared
    ok = ok && (lot.getCommit(address(player1)) == bytes32(0));
    ok = ok && (lot.getCommit(address(player2)) == bytes32(0));

    // reveal flags cleared
    ok = ok && (lot.getHasRevealed(address(player1)) == false);
    ok = ok && (lot.getHasRevealed(address(player2)) == false);

    // commit flags cleared
    ok = ok && (lot.getHasCommitted(address(player1)) == false);
    ok = ok && (lot.getHasCommitted(address(player2)) == false);

    // revealed values cleared
    ok = ok && (lot.getReveal(address(player1)) == 0);
    ok = ok && (lot.getReveal(address(player2)) == 0);

    return ok;
}
```

The harness separates round closure from round restart. When `endLottery()` is successfully executed and the round is observed to be closed (i.e., `startTime == 0`), the cleanliness of the post-round state is recorded:

```
function endRound() external {
    try lot.endLottery() {
        (uint256 s, , ) = lot.getTimes();
        if (s == 0) {
            lastPostCloseWasClean = _isCleanClosedState();
            pendingPostCloseCheck = true;
        }
    } catch {
        // checks are armed only after a successful close
    }
}
```

A subsequent call to `startLottery()` must succeed if and only if a clean round closure was previously observed:

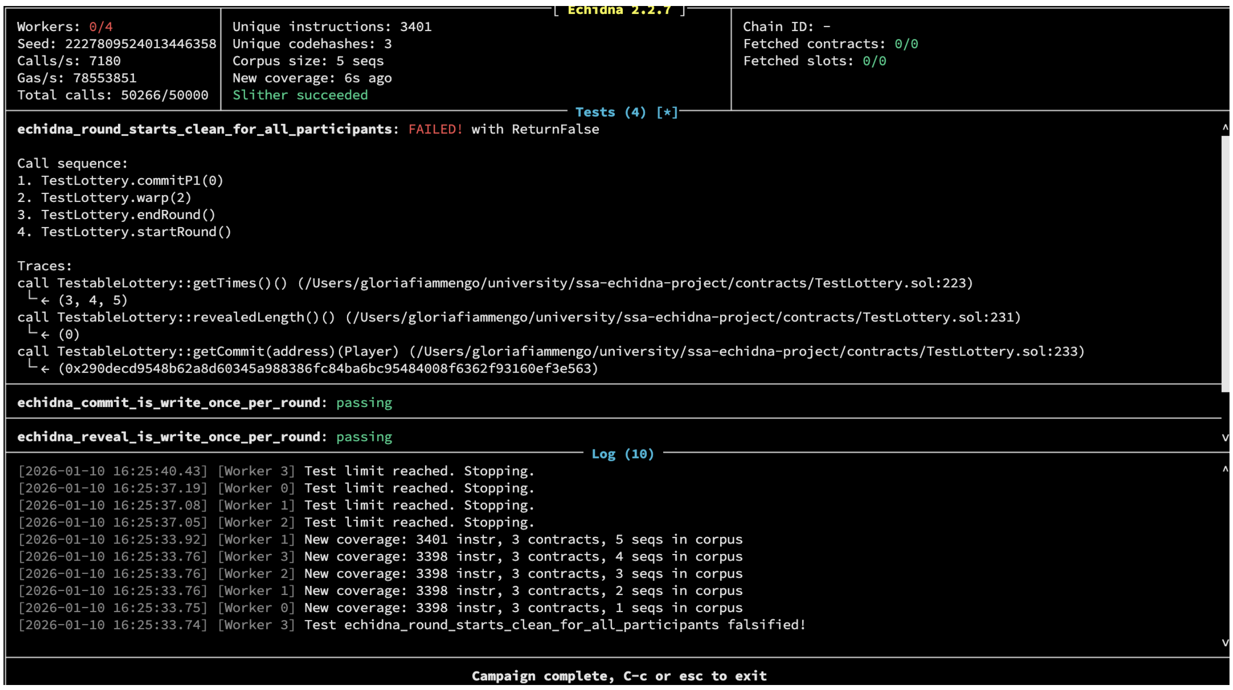
```
function startRound() external {
    if (!pendingPostCloseCheck) {
        try lot.startLottery() {} catch {}
        return;
    }

    try lot.startLottery() {
        lastStartAfterCloseSucceeded = true;
    } catch {
        lastStartAfterCloseSucceeded = false;
    }

    pendingPostCloseCheck = false;
}
```

Finally, the Echidna property asserts that every observed clean round closure can always be followed by a successful round restart:

```
function echidna_round_starts_clean_for_all_participants()
    public
    view
    returns (bool)
{
    return lastPostCloseWasClean && lastStartAfterCloseSucceeded;
}
```

The invariant is violated if either residual round-specific state survives a successful closure, or if a new round cannot be started after a clean close.

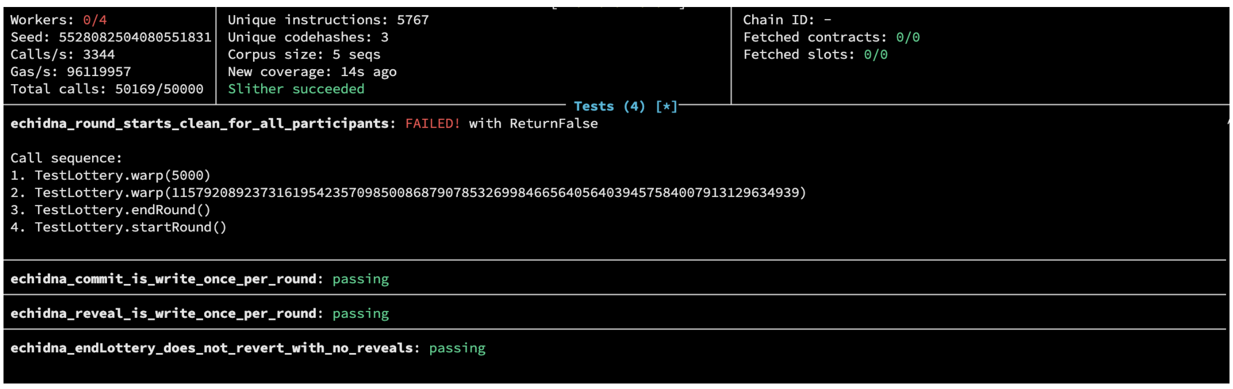
Counterexample 1 (stale round state) Echidna produced an execution in which a participant committed during a round, the round was successfully closed, and a new round was started. At the beginning of the new round, stale state from the previous round remained reachable (e.g., prior commitments or reveal flags were still set), violating the intended independence between rounds.



```
[ Echidna 2.2.7 ]
Workers: 0/4                     Unique instructions: 3401              Chain ID: -
Seed: 2227809524013446358        Unique codehashes: 3                   Fetched contracts: 0/0
Calls/s: 7180                    Corpus size: 5 seqs                    Fetched slots: 0/0
Gas/s: 78553851                  New coverage: 6s ago
Total calls: 50266/50000         Slither succeeded
                                          Tests (4) [*]
echidna_round_starts_clean_for_all_participants: FAILED! with ReturnFalse

Call sequence:
1. TestLottery.commitP1(0)
2. TestLottery.warp(2)
3. TestLottery.endRound()
4. TestLottery.startRound()

Traces:
call TestableLottery::getTimes()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:223)
 └← (3, 4, 5)
call TestableLottery::revealedLength()() (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:231)
 └← (0)
call TestableLottery::getCommit(address)(Player) (/Users/gloriafiammengo/university/ssa-echidna-project/contracts/TestLottery.sol:233)
 └← (0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563)

echidna_commit_is_write_once_per_round: passing

echidna_reveal_is_write_once_per_round: passing
                                          Log (10)
[2026-01-10 16:25:40.43] [Worker 3] Test limit reached. Stopping.
[2026-01-10 16:25:37.19] [Worker 0] Test limit reached. Stopping.
[2026-01-10 16:25:37.08] [Worker 1] Test limit reached. Stopping.
[2026-01-10 16:25:37.05] [Worker 2] Test limit reached. Stopping.
[2026-01-10 16:25:33.92] [Worker 1] New coverage: 3401 instr, 3 contracts, 5 seqs in corpus
[2026-01-10 16:25:33.76] [Worker 3] New coverage: 3398 instr, 3 contracts, 4 seqs in corpus
[2026-01-10 16:25:33.76] [Worker 2] New coverage: 3398 instr, 3 contracts, 3 seqs in corpus
[2026-01-10 16:25:33.76] [Worker 1] New coverage: 3398 instr, 3 contracts, 2 seqs in corpus
[2026-01-10 16:25:33.75] [Worker 0] New coverage: 3398 instr, 3 contracts, 1 seqs in corpus
[2026-01-10 16:25:33.74] [Worker 3] Test echidna_round_starts_clean_for_all_participants falsified!

                              Campaign complete, C-c or esc to exit
```

Root cause 1 The contract correctly reset timing variables (`startTime`, `revealTime`, `endTime`) at the end of a round, but did not initially clear all round-specific participant state. As a result, mappings and auxiliary data structures associated with the previous round could influence the behavior of the next round.

Counterexample 2 (time-model-induced restart failure) After fixing the cleanup logic, Echidna was still able to falsify the invariant under a different execution. In this case, a round was closed at an extreme timestamp close to `uint256.max`, followed immediately by a call to `startLottery()`. The new round failed to start because `startLottery()` reverted due to arithmetic overflow when computing `revealTime` and `endTime`.

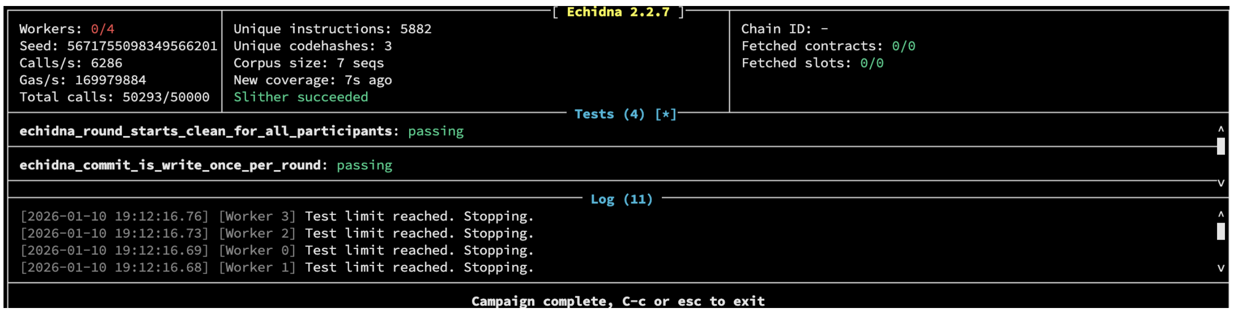


```
Workers: 0/4                     Unique instructions: 5767              Chain ID: -
Seed: 5528082504080551831        Unique codehashes: 3                   Fetched contracts: 0/0
Calls/s: 3344                    Corpus size: 5 seqs                    Fetched slots: 0/0
Gas/s: 96119957                  New coverage: 14s ago
Total calls: 50169/50000         Slither succeeded
                                          Tests (4) [*]
echidna_round_starts_clean_for_all_participants: FAILED! with ReturnFalse

Call sequence:
1. TestLottery.warp(5000)
2. TestLottery.warp(115792089237316195423570985008687907853269984665640564039457584007913129634939)
3. TestLottery.endRound()
4. TestLottery.startRound()

echidna_commit_is_write_once_per_round: passing

echidna_reveal_is_write_once_per_round: passing

echidna_endLottery_does_not_revert_with_no_reveals: passing
```

Root cause 2 This violation was not caused by residual round state. All per-round data structures were correctly cleared at `endLottery()`. Instead, the issue originated from the testing time model: the harness allowed the observable time to remain arbitrarily close to `uint256.max` after a round closure. Since Solidity `^0.8.x` enforces checked arithmetic, the additions performed in `startLottery()` (e.g., `startTime + period + period`) could overflow and revert, preventing the new round from starting despite a clean logical state. This issue does not represent a vulnerability of the contract logic, but an artifact of the adversarial time model used in the test harness.

Fix applied Two complementary fixes were applied. First, the contract was updated to explicitly track all participants involved in a round and to clear all per-round and per-participant state at `endLottery()`. Second, the testing harness was corrected by modifying the overridden time source `_now()` in `TestableLottery` to enforce a *start-safe* upper bound on the observable time whenever the lottery is closed (`startTime == 0`). This prevents artificial overflows during `startLottery()` while still allowing time to advance freely during an active round.

Verification after the fix After applying both fixes, each successful round closure can be followed by a successful round start. The new round always begins from a clean state: `committed.length == 0`, `revealed.length == 0`, and all participant-specific mappings are reset. Echidna is no longer able to falsify this invariant.



```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 5882        Chain ID: -
Seed: 5671755098349566201       Unique codehashes: 3             Fetched contracts: 0/0
Calls/s: 6286                   Corpus size: 7 seqs              Fetched slots: 0/0
Gas/s: 169979884                New coverage: 7s ago
Total calls: 50293/50000        Slither succeeded
                                Tests (4) [*]
echidna_round_starts_clean_for_all_participants: passing

echidna_commit_is_write_once_per_round: passing
                                Log (11)
[2026-01-10 19:12:16.76] [Worker 3] Test limit reached. Stopping.
[2026-01-10 19:12:16.73] [Worker 2] Test limit reached. Stopping.
[2026-01-10 19:12:16.69] [Worker 0] Test limit reached. Stopping.
[2026-01-10 19:12:16.68] [Worker 1] Test limit reached. Stopping.

                    Campaign complete, C-c or esc to exit
```

5.5 Invariant 12: The lottery is restricted to under-65 participants

Specification requirement. According to the assignment specification, participation in the lottery is restricted to under-65 participants. Any participant aged 65 or older must not be able to take part in the lottery process.

Property encoding. The invariant is encoded as an Echidna check enforcing that a participant with age greater than or equal to 65 cannot successfully join a lottery round. The property is formulated in terms of observable effects: after a commit attempt by a senior participant, no commitment must be recorded and no participant-specific state must be modified.

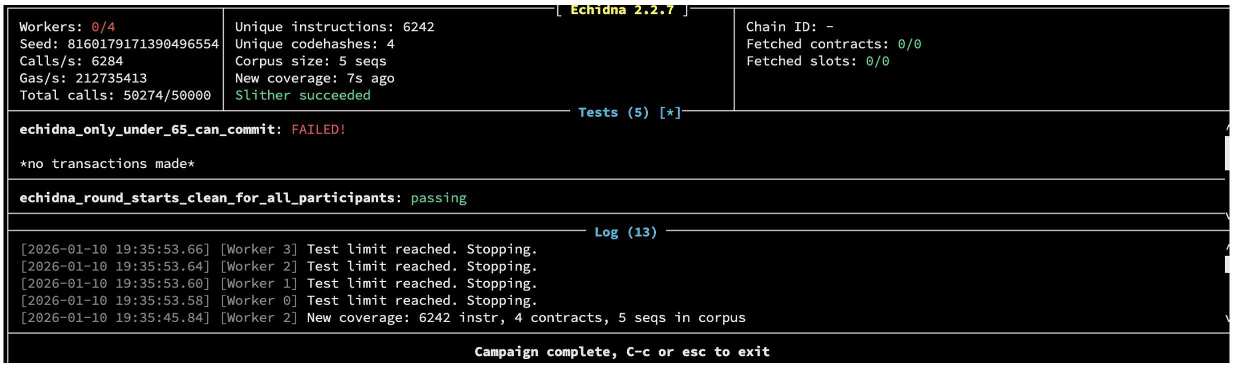
Test harness snippet The invariant is tested by modeling a senior participant and explicitly attempting to perform a commit during an active lottery round.

The harness allows the senior participant to call `commit()`, but verifies that this call produces no observable state changes. Both successful execution and revert are acceptable outcomes, as long as no commitment is recorded.

```
// property 12: under-65 only
function echidna_only_under_65_can_commit() public returns (bool) {
    // senior tries to commit: revert is fine, but state must not change
    try senior.commitSecret(123456) {} catch {}

    bool ok = true;
    ok = ok && (lot.getCommit(address(senior)) == bytes32(0));
    ok = ok && (lot.getHasCommitted(address(senior)) == false);
    return ok;
}
```

The property attempts to violate the age restriction by letting a senior participant invoke `commit()`. The invariant holds only if, after the call, no commitment is recorded and the participant is not marked as having committed. Any observable modification of commit-related state would indicate a violation of the age-based access restriction.

Counterexample. In the initial implementation, Echidna was able to generate executions in which a participant aged 65 or older successfully committed to an active lottery round. The commit was accepted and persisted in the contract state, violating the age restriction required by the specification.



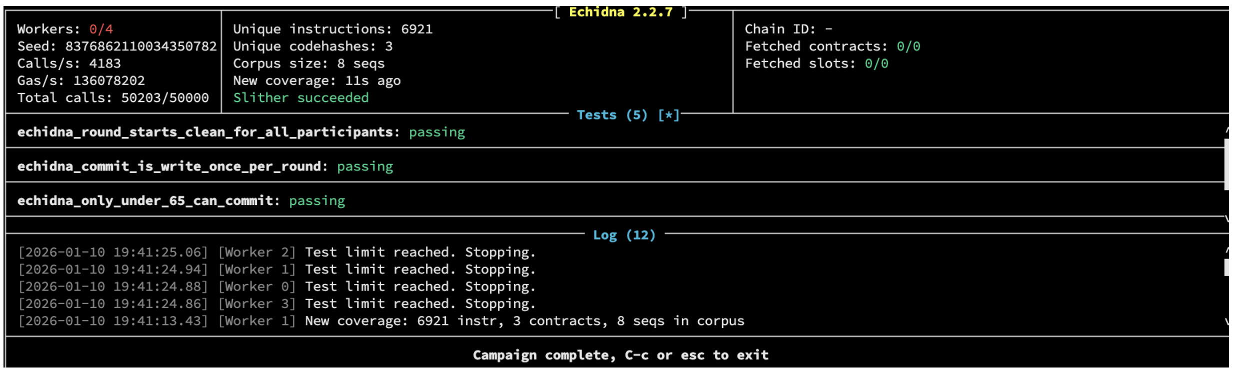
```
[ Echidna 2.2.7 ]
Workers: 0/4                    Unique instructions: 6242        Chain ID: -
Seed: 8160179171390496554       Unique codehashes: 4             Fetched contracts: 0/0
Calls/s: 6284                   Corpus size: 5 seqs              Fetched slots: 0/0
Gas/s: 212735413                New coverage: 7s ago
Total calls: 50274/50000        Slither succeeded
                                Tests (5) [*]
echidna_only_under_65_can_commit: FAILED!

*no transactions made*

echidna_round_starts_clean_for_all_participants: passing
                                Log (13)
[2026-01-10 19:35:53.66] [Worker 3] Test limit reached. Stopping.
[2026-01-10 19:35:53.64] [Worker 2] Test limit reached. Stopping.
[2026-01-10 19:35:53.60] [Worker 1] Test limit reached. Stopping.
[2026-01-10 19:35:53.58] [Worker 0] Test limit reached. Stopping.
[2026-01-10 19:35:45.84] [Worker 2] New coverage: 6242 instr, 4 contracts, 5 seqs in corpus

                    Campaign complete, C-c or esc to exit
```

Root cause. The contract did not enforce any age-based access control in the `commit()` function. Although participant age information was available through the `Taxpayer` abstraction, the lottery logic did not check this constraint, allowing senior participants to interact with the protocol as valid players.

Fix applied. The fix introduces an explicit age check in the lottery entry point. The `commit()` function now requires the caller to be a valid `Taxpayer` contract with `age < 65`, preventing senior participants from joining a round. For defensive consistency, the same age constraint is also enforced in `reveal()`, ensuring that all lottery interactions remain aligned with the eligibility rule.

The testing harness was updated accordingly, modeling eligible participants as under-65 taxpayers and introducing a dedicated senior participant wrapper for validation.

Validation after the fix. After applying the fix, any commit attempt performed by a senior participant is rejected or produces no observable state changes. Echidna is no longer able to falsify the invariant, confirming that the age restriction is correctly enforced by the contract.



```
[ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 6921        Chain ID: -
Seed: 8376862110034350782    Unique codehashes: 3             Fetched contracts: 0/0
Calls/s: 4183                Corpus size: 8 seqs              Fetched slots: 0/0
Gas/s: 136078202             New coverage: 11s ago
Total calls: 50203/50000     Slither succeeded
                                      Tests (5) [*]
echidna_round_starts_clean_for_all_participants: passing
echidna_commit_is_write_once_per_round: passing
echidna_only_under_65_can_commit: passing
                                      Log (12)
[2026-01-10 19:41:25.06] [Worker 2] Test limit reached. Stopping.
[2026-01-10 19:41:24.94] [Worker 1] Test limit reached. Stopping.
[2026-01-10 19:41:24.88] [Worker 0] Test limit reached. Stopping.
[2026-01-10 19:41:24.86] [Worker 3] Test limit reached. Stopping.
[2026-01-10 19:41:13.43] [Worker 1] New coverage: 6921 instr, 3 contracts, 8 seqs in corpus
                              Campaign complete, C-c or esc to exit
```

5.6 Invariant 13: Phase separation between commit and reveal

Specification requirement. The lottery implements a commit/reveal protocol with two distinct phases. Commitments must only be accepted during the commit phase, and reveals must only be accepted during the reveal phase. Operations attempted outside their valid time window must be rejected or must not produce any observable state changes.

Property encoding. The invariant checks protocol phase separation through two sub-requirements: (i) any `commit()` attempt performed after `revealTime` must either revert or have no observable effects, and (ii) any `reveal()` attempt performed after `endTime` must either revert or have no observable effects. The harness enforces these situations by advancing the observable time beyond the corresponding boundary and then checking that no state variables are modified.

Test harness snippet The invariant is tested by explicitly forcing the system beyond the phase boundaries of the commit–reveal protocol and observing whether late operations produce any observable state changes.

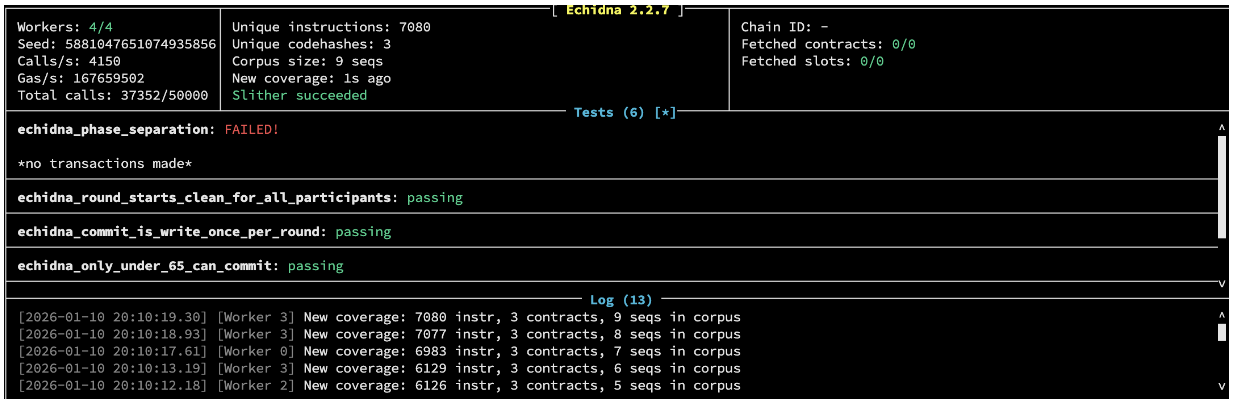
```
function echidna_phase_separation() public returns (bool) {
    bool ok = true;

    (uint256 st, uint256 rt, uint256 et) = lot.getTimes();
    uint256 nowT = lot.getTime();

    if (st == 0) return true;

    // commit after revealTime must revert or have no observable effects
    if (nowT < rt) {
        try lot.warp(rt - nowT + 1) {} catch {}
    }

    bool commitSucceeded = true;
    try player1.commitSecret(123) {} catch {
        commitSucceeded = false;
    }

    if (commitSucceeded) {
        ok = ok && (lot.getCommit(address(player1)) == bytes32(0));
        ok = ok && (lot.getHasCommitted(address(player1)) == false);
    }

    // reveal after endTime must revert or have no observable effects
    nowT = lot.getTime();
    if (nowT < et) {
        try lot.warp(et - nowT + 1) {} catch {}
    }

    bool revealSucceeded = true;
    try player2.revealSecret(456) {} catch {
        revealSucceeded = false;
    }

    if (revealSucceeded) {
        ok = ok && (lot.getHasRevealed(address(player2)) == false);
        ok = ok && (lot.getReveal(address(player2)) == 0);
    }

    return ok;
}
```

The property first advances the observable time beyond `revealTime` and attempts a late `commit()`. If the call succeeds, the invariant enforces that no commitment is recorded and the participant is not marked as having committed.

The property then advances time beyond `endTime` and attempts a late `reveal()`. If the call succeeds, the invariant checks that the participant is not marked as revealed and that no reveal value is stored.

Both reverts and successful executions are acceptable outcomes, as long as no observable state changes occur outside the valid protocol phases. Any modification of commit- or reveal-related state after the corresponding phase boundary causes the invariant to fail.

Counterexample. In the initial implementation, Echidna was able to generate executions in which commits were still accepted after the reveal phase had started and reveals were accepted after the round end time. These calls modified the contract state (e.g., recording a commitment or updating reveal-related state), violating the intended separation of protocol phases.

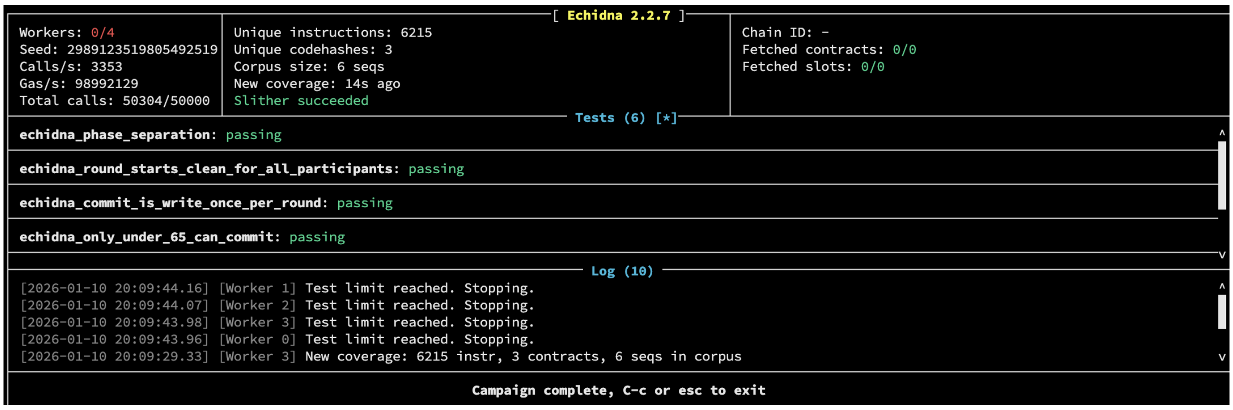


```
                                        [ Echidna 2.2.7 ]
Workers: 4/4                 Unique instructions: 7080          Chain ID: -
Seed: 5881047651074935856    Unique codehashes: 3               Fetched contracts: 0/0
Calls/s: 4150                Corpus size: 9 seqs                Fetched slots: 0/0
Gas/s: 167659502             New coverage: 1s ago
Total calls: 37352/50000     Slither succeeded
                                         Tests (6) [*]
echidna_phase_separation: FAILED!

*no transactions made*

echidna_round_starts_clean_for_all_participants: passing

echidna_commit_is_write_once_per_round: passing

echidna_only_under_65_can_commit: passing

                                          Log (13)
[2026-01-10 20:10:19.30] [Worker 3] New coverage: 7080 instr, 3 contracts, 9 seqs in corpus
[2026-01-10 20:10:18.93] [Worker 3] New coverage: 7077 instr, 3 contracts, 8 seqs in corpus
[2026-01-10 20:10:17.61] [Worker 0] New coverage: 6983 instr, 3 contracts, 7 seqs in corpus
[2026-01-10 20:10:13.19] [Worker 3] New coverage: 6129 instr, 3 contracts, 6 seqs in corpus
[2026-01-10 20:10:12.18] [Worker 2] New coverage: 6126 instr, 3 contracts, 5 seqs in corpus
```

Root cause. Although the contract tracked `revealTime` and `endTime`, these timestamps were not enforced as upper bounds in the corresponding entry points. Specifically, `commit()` lacked a guard preventing commits after `revealTime`, and `reveal()` lacked a guard preventing reveals after `endTime`.

Fix applied. Two explicit temporal guards were added. The `commit()` function now requires `_now() < revealTime`, closing the commit phase at `revealTime`, while the `reveal()` function now requires `_now() < endTime`, closing the reveal phase at `endTime`. This enforces a strict phase separation consistent with the commit–reveal protocol.

Validation after the fix. After applying the guards, any commit attempt after `revealTime` and any reveal attempt after `endTime` either revert or produce no observable state changes. Echidna is no longer able to falsify the invariant.



```
                                        [ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 6215          Chain ID: -
Seed: 2989123519805492519    Unique codehashes: 3               Fetched contracts: 0/0
Calls/s: 3353                Corpus size: 6 seqs                Fetched slots: 0/0
Gas/s: 98992129              New coverage: 14s ago
Total calls: 50304/50000     Slither succeeded
                                         Tests (6) [*]
echidna_phase_separation: passing

echidna_round_starts_clean_for_all_participants: passing

echidna_commit_is_write_once_per_round: passing

echidna_only_under_65_can_commit: passing

                                          Log (10)
[2026-01-10 20:09:44.16] [Worker 1] Test limit reached. Stopping.
[2026-01-10 20:09:44.07] [Worker 2] Test limit reached. Stopping.
[2026-01-10 20:09:43.98] [Worker 3] Test limit reached. Stopping.
[2026-01-10 20:09:43.96] [Worker 0] Test limit reached. Stopping.
[2026-01-10 20:09:29.33] [Worker 3] New coverage: 6215 instr, 3 contracts, 6 seqs in corpus

                              Campaign complete, C-c or esc to exit
```

5.7 Invariant 14: Winner validity (winner must be a revealed participant)

Specification requirement. If a lottery round ends with at least one valid reveal, the winner selected at round closure must be one of the participants who revealed in that round. This invariant expresses a minimal correctness requirement for winner extraction and does not address fairness or randomness quality.

Property encoding. The invariant is checked as a post-condition of `endLottery()`. Whenever the harness observes that a round has at least one revealed participant, it advances the observable time beyond `endTime`, snapshots the `revealed[]` list, closes the round, and verifies that the recorded `winner` belongs to the previously revealed set. Since `revealed[]` is cleared during round cleanup, the snapshot is taken before invoking `endLottery()`.

Test harness snippet The invariant is implemented as a post-condition checked at round termination. It verifies that, whenever a round ends with at least one revealed participant, the selected winner belongs to the set of addresses that successfully revealed during that round.

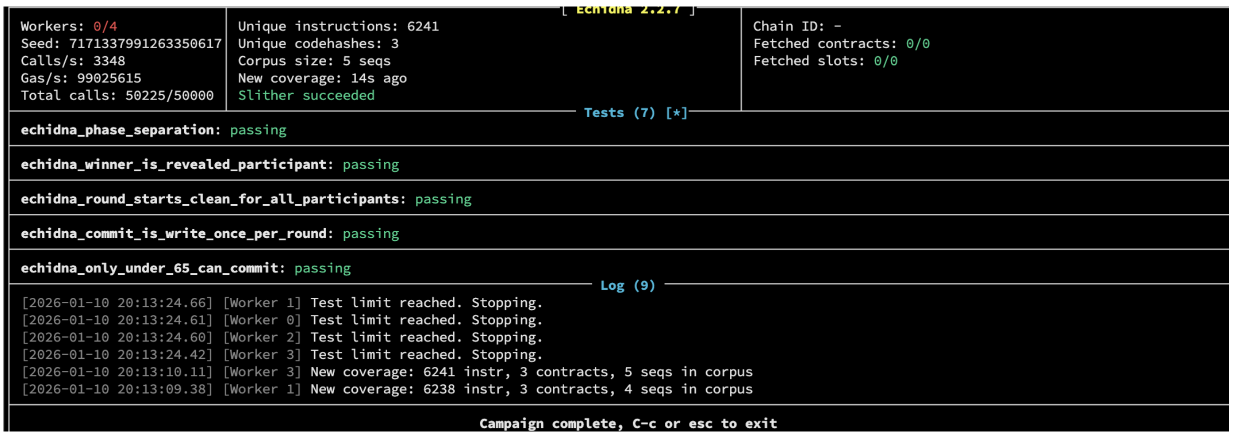
```
function echidna_winner_is_revealed_participant() public returns (bool) {
    (uint256 st, , uint256 et) = lot.getTimes();

    if (st == 0) return true;

    uint256 rlen = lot.revealedLength();
    if (rlen == 0) return true;
    // make endLottery reachable
    uint256 nowT = lot.getTime();
    if (nowT < et) {
        try lot.warp(et - nowT + 1) {} catch {}
    }

    // snapshot revealed[] before endLottery clears it
    address[] memory snap = new address[](rlen);
    for (uint256 i = 0; i < rlen; i++) {
        snap[i] = lot.revealedAt(i);
    }

    // close must succeed now
    try lot.endLottery() {} catch {
        return false;
    }

    // winner must be one of the revealed addresses
    address w = lot.getWinner();
    for (uint256 i = 0; i < rlen; i++) {
        if (snap[i] == w) return true;
    }

    return false;
}
```

The property is only enforced when a round is active and at least one participant has successfully revealed. Before invoking `endLottery()`, the harness snapshots the list of revealed addresses using the observable accessor `revealedAt(i)`, since the `revealed` array is cleared during round cleanup.

After closing the round, the invariant checks that the stored winner address belongs to the previously revealed set. The property does not impose any constraint on fairness or randomness quality; it only enforces that the winner is selected from the correct domain.

Validation result. Echidna is unable to falsify the invariant. In all explored executions where a round ends with at least one revealed participant, the winner selected by `endLottery()` is always an address contained in the corresponding `revealed[]` set.

Discussion. The absence of counterexamples confirms that the winner selection logic is correctly scoped to the set of participants who revealed in the current round. This invariant complements the previous ones addressing phase separation and round isolation, providing additional confidence in the correctness of the lottery termination logic.



```
                                   [ Echidna 2.2.7 ]
Workers: 0/4                 Unique instructions: 6241          Chain ID: -
Seed: 7171337991263350617    Unique codehashes: 3               Fetched contracts: 0/0
Calls/s: 3348                Corpus size: 5 seqs                Fetched slots: 0/0
Gas/s: 99025615              New coverage: 14s ago
Total calls: 50225/50000     Slither succeeded
                                   Tests (7) [*]
echidna_phase_separation: passing

echidna_winner_is_revealed_participant: passing

echidna_round_starts_clean_for_all_participants: passing

echidna_commit_is_write_once_per_round: passing

echidna_only_under_65_can_commit: passing
                                   Log (9)
[2026-01-10 20:13:24.66] [Worker 1] Test limit reached. Stopping.
[2026-01-10 20:13:24.61] [Worker 0] Test limit reached. Stopping.
[2026-01-10 20:13:24.60] [Worker 2] Test limit reached. Stopping.
[2026-01-10 20:13:24.42] [Worker 3] Test limit reached. Stopping.
[2026-01-10 20:13:10.11] [Worker 3] New coverage: 6241 instr, 3 contracts, 5 seqs in corpus
[2026-01-10 20:13:09.38] [Worker 1] New coverage: 6238 instr, 3 contracts, 4 seqs in corpus

                          Campaign complete, C-c or esc to exit
```

Additional validated invariants

In addition to the invariants that revealed inconsistencies, several marriage-related properties were explicitly tested and validated using Echidna.

These properties were encoded as state-based checks over the same test harness used for Invariants 1–7 and verified that no counterexample exists in the explored state space. In particular, the following conditions were validated: (i) spouse references remain stable for the entire duration of a valid marriage, and (ii) no execution allows a taxpayer to be married to more than one spouse at the same time.

After applying the fixes described in Invariants 1–7, Echidna did not find any execution trace violating these properties.