Security In Software Applications - Project Review

Andrea Maggiore, 1947898

1 Introduction and Tool Overview

Smart contracts are autonomous programs deployed on a blockchain that manage assets and enforce rules without the possibility of modification after deployment. While immutability is a key advantage, it also represents a major security risk: any logic error becomes permanent and may lead to irreversible consequences such as financial losses or inconsistent states.

Traditional testing approaches, including unit and integration tests, are often insufficient for smart contracts. The extremely large state space induced by multiple users, long interaction sequences, and state-dependent behaviors makes it difficult to anticipate all possible execution traces. As a result, many vulnerabilities emerge only under unexpected interaction patterns.

To address these limitations, this project adopts a *property-based security testing* approach. Instead of validating specific scenarios, contract behavior is verified against a set of *security properties* that must always hold, independently of the executed transaction sequence.

The project focuses on the analysis and hardening of two Solidity smart contracts:

- **Taxpayer**, modeling tax allowances and family relationships;
- **Lottery**, implementing a commit–reveal lottery protocol.

For each contract, security-relevant properties are defined and tested using the Echidna fuzzing tool. Whenever a property violation is detected, the corresponding execution trace is analyzed, the root cause is identified, and the contract logic is fixed. The tests are then re-executed to validate the effectiveness of the applied patch.

1.1 Property-Based Fuzz Testing

Property-based testing focuses on verifying *invariants*, namely conditions that must never be violated during execution. In smart contracts, these invariants often encode security requirements such as value conservation, relationship consistency, and correctness of state transitions.

Properties are expressed as boolean predicates over the contract state. A testing tool attempts to falsify them by generating random inputs and arbitrary sequences of function calls. If a violation is found, a concrete execution trace reproducing the bug is provided.

This approach is particularly effective for contracts with complex state machines, multiple interacting users, and long execution sequences.

1.2 Echidna Overview

Echidna is a property-based fuzzer designed for Ethereum smart contracts written in Solidity. It executes randomly generated transaction sequences against a target contract while continuously checking user-defined properties.

Each property is implemented as a Solidity function returning a boolean value. When a property evaluates to `false`, Echidna reports a failure and stores the triggering sequence in a corpus, enabling deterministic reproduction of the bug.

In this project, Echidna is used both to discover vulnerabilities and to validate that applied fixes correctly enforce the intended security properties. All experiments are performed with fixed configurations and stored artifacts to ensure reproducibility.

2 Methodology

The methodology adopted in this project follows a systematic and evidence-driven approach to smart contract security analysis. Rather than relying on ad-hoc testing or manual inspection, the validation process is structured around the definition and verification of explicit security properties.

Each contract is analyzed through an iterative workflow composed of the following steps:

1. definition of security properties,
2. automated fuzzing and counterexample discovery,
3. root cause analysis,
4. contract patching,
5. re-testing and validation.

This workflow ensures that every vulnerability is not only detected, but also explained, fixed, and validated against regression.

2.1 Security Properties Definition

Security properties represent conditions that must **never be violated**, independently of the executed transaction sequence. Typical examples include consistency of bidirectional relationships, conservation of numerical quantities, correctness of state transitions, and prevention of illegal actions such as double participation.

Each property is encoded as a Solidity function returning a boolean value. During fuzzing, Echidna continuously checks that all properties evaluate to `true`; any violation immediately signals an error in the contract logic.

Properties are expressed as *state invariants*, meaning that they depend only on the observable contract state and not on a specific execution path. This design choice reduces flakiness and improves test robustness.

2.2 Fuzzing and Counterexample Discovery

Once properties are defined, Echidna explores the contract behavior through randomized testing. The fuzzer generates arbitrary sequences of function calls, simulating interactions from multiple users.

When a property is violated, Echidna reports the failing invariant and stores a concrete execution trace (counterexample) in a persistent corpus. Each counterexample provides a minimal and reproducible proof of incorrect behavior, often revealing vulnerabilities that are difficult to detect through manual testing.

2.3 Root Cause Analysis, Patching, and Re-testing

For each counterexample, a manual analysis is performed to identify the underlying cause of the violation, such as missing validations or inconsistent state transitions. The contract is then patched by applying minimal and targeted changes, including strengthened input validation, symmetric state updates, and stricter phase enforcement.

After patching, the fuzzing process is repeated using the same configuration. A fix is considered valid only if the corresponding property holds for long fuzzing runs without producing new counterexamples.

Failing and passing executions are documented through logs and screenshots, providing concrete evidence of the effectiveness of the applied fixes.

3 Taxpayer Contract — Marriage Symmetry

This section analyzes the first security-critical component of the project: the `Taxpayer` smart contract. The focus is on the correctness of the marriage relationship, modeled as a bidirectional association between two addresses.

From a security perspective, marriage represents shared mutable state that must remain consistent under all execution traces. Violations of this consistency may lead to invalid states or unintended privilege propagation.

3.1 Threat Model

The attacker is modeled as an arbitrary externally owned account interacting with public contract functions. The objective is to force the contract into an inconsistent marriage state, for example through:

- asymmetric spouse references,
- self-marriage,
- dangling references after divorce.

No assumptions are made on call ordering or honest behavior.

3.2 Security Properties

The intended behavior of the marriage logic is captured by the following invariants.

P1.1 — Marriage Symmetry If an address `a` is married to `b`, and `b` is not the zero address, then `b` must also be married to `a`:

$$spouse(a) = b \land b \neq 0 \Rightarrow spouse(b) = a$$

P1.2 — No Self-Marriage An address cannot be married to itself:

$$spouse(a) \neq a$$

P1.3 — Coherent Unmarried State If an address `a` is unmarried, no other address may reference it as a spouse:

$$spouse(a) = 0 \Rightarrow \neg\exists x \; : \; spouse(x) = a$$

Together, these invariants guarantee global consistency of the marriage relationship.

3.3 Property-Based Testing

The properties were implemented as invariants in an Echidna test contract. Echidna generated arbitrary sequences of marriage and divorce operations, enabling exploration of unexpected call interleavings.

3.4 Discovered Vulnerability and Fix

During fuzzing, Echidna produced a counterexample violating the symmetry property, resulting in a state where a marriage was updated on only one side. The root cause was the absence of a strictly enforced bidirectional update in the marriage logic.

The contract was patched by enforcing atomic updates of both spouse mappings and rejecting any operation that could result in an inconsistent state. After re-running the same fuzzing configuration, no further counterexamples were found, confirming the correctness of the fix.

4 Taxpayer Contract — Allowance Baseline and Pooling

This section analyzes the financial logic of the `Taxpayer` contract, focusing on tax allowance management and redistribution between married taxpayers. Since allowances are numerical state variables, strong conservation invariants are required to prevent value corruption.

Incorrect handling of allowances may lead to unintended value creation, loss of funds, or inconsistent financial states.

4.1 Threat Model

The attacker is modeled as an arbitrary user invoking public functions to manipulate allowance values. Relevant attack strategies include:

- transferring allowance between non-spouses,
- exploiting marriage and divorce transitions,
- forcing unmarried taxpayers to retain non-baseline values.

No assumptions are made about call ordering or honest behavior.

4.2 Security Properties

The intended behavior of the allowance system is captured by the following invariants.

P2.1 — Baseline Allowance An unmarried taxpayer must always have the baseline allowance:

$$spouse(a) = 0 \Rightarrow allowance(a) = 5000$$

P2.2 — Pooling Conservation For a married couple, the total allowance must remain constant:

$$spouse(a) = b \land spouse(b) = a \Rightarrow allowance(a) + allowance(b) = 10000$$

P2.3 — Authorized Pooling Allowance transfers are allowed only between mutually married taxpayers:

$$spouse(a) = b \land spouse(b) = a$$

Together, these invariants prevent unauthorized transfers and value corruption.

4.3 Property-Based Testing and Findings

The properties were implemented as Echidna invariants and tested under arbitrary sequences of marriage, divorce, and allowance transfer operations.

Fuzzing revealed multiple violations, including:

- unmarried taxpayers retaining modified allowances after divorce,
- allowance transfers occurring under asymmetric spouse relationships.

These violations broke conservation guarantees and resulted in inconsistent financial states.

4.4 Fix and Validation

The contract was patched by resetting allowances upon divorce, enforcing mutual spouse checks before transfers, and rejecting operations involving unmarried parties.

After applying these fixes, extended fuzzing runs produced no further counterexamples, confirming that allowance pooling is conservative and robust under arbitrary call sequences.

5 Lottery Commit–Reveal Protocol

This section analyzes a simplified *commit–reveal lottery protocol*, focusing on input validation, state machine correctness, and robustness against adversarial behavior. The goal is not to provide cryptographically strong randomness, but to ensure that *critical safety properties are never violated* under arbitrary interaction sequences.

5.1 Protocol Overview

The lottery follows a time-driven protocol composed of three phases:

- **NotStarted**: no interaction allowed,
- **Commit**: users submit commitments,
- **Reveal**: users reveal their values.

Phase transitions are enforced internally through timestamps. The protocol exposes four main functions: `startLottery`, `commit`, `reveal`, and `endLottery`.

The winner is deterministically selected as the sum of revealed values modulo the number of valid reveals. Randomness quality is considered out of scope.

5.2 Threat Model

An adversarial participant may:

- call functions in arbitrary order,
- commit or reveal multiple times,
- reveal without committing,
- interact with the contract in invalid phases,
- exploit incomplete cleanup between rounds.

The attacker cannot break cryptographic primitives, but may exploit missing checks or inconsistent state updates.

5.3 Security Properties

The protocol correctness is captured by the following invariants.

L1 — Commit–Reveal Binding A revealed value must match the submitted commitment:

$$\texttt{keccak256(reveal)} = \texttt{commit}$$

L2 — No Reveal Without Commit A participant cannot reveal without a prior valid commit.

L3 — Unique Participation Each address may commit and reveal at most once per lottery round.

L4 — Phase Correctness Each function is callable only during its intended protocol phase.

L5 — State Cleanup After finalization, all per-user state and participant lists must be reset.

L6 — Winner Validity The selected winner must belong to the set of revealed participants.

5.4 Property-Based Testing and Fixes

All properties were encoded as Echidna invariants in `Echidna_Lottery.sol`. Fuzzing was performed with up to 300,000 transactions using a fixed corpus.

Initial runs revealed violations such as early reveals, double participation, and incomplete state cleanup. These issues were fixed by enforcing strict phase checks, participation flags, and full state resets.

After patching, all properties passed both smoke and long fuzzing runs, confirming the correctness of the protocol under arbitrary call sequences.

6 Reproducibility and Experimental Setup

Reproducibility is a fundamental requirement of security testing. All experiments in this project were designed to be fully reproducible, allowing an external evaluator to re-run the tests and obtain equivalent results.

This section summarizes the toolchain, configuration, and execution workflow used for property-based fuzzing with Echidna.

6.1 Toolchain

The project relies on:

- **Solidity compiler**: compatible with `pragma solidity 0̂.8.22`,
- **Echidna**: `echidna-test`, executed via the official Trail of Bits Docker image,
- **Docker**: used to ensure a consistent execution environment.

Using Docker avoids discrepancies related to local installations, operating systems, or compiler versions.

6.2 Repository Structure

The repository separates contracts, tests, configurations, and artifacts:

- `contracts/`: Solidity contracts,
- `tests/echidna/`: Echidna harnesses and properties,
- `echidna/`: YAML configuration files,
- `artifacts/`: logs, corpora, and selected evidence.

This structure enables a clear mapping between properties, executions, and results.

6.3 Echidna Configuration and Execution

Each test suite is executed using a dedicated YAML configuration file specifying: `testLimit`, `seqLen`, `shrinkLimit`, and `corpusDir`.

For the Lottery protocol, the following configuration was used:

```
testLimit:   300000
seqLen:      80
shrinkLimit: 8000
corpusDir:   artifacts/corpus/lottery
```

All experiments can be reproduced with a single command, for example:

```
docker run --rm --platform linux/amd64 \
  -v "$PWD":/src -w /src \
  trailofbits/echidna:latest \
  echidna-test tests/echidna/Echidna_Lottery.sol \
    --config echidna/lottery.yaml \
    --contract Echidna_Lottery
```

Both smoke runs and long release runs were executed, and all properties passed consistently in the final configuration.

6.4 Evidence and Limitations

Collected artifacts include Echidna logs, counterexample traces, and corpus snapshots. Final passing runs are stored in `artifacts/evidence/`.

The main limitations are that fuzzing provides empirical validation rather than formal proofs, randomness quality is simplified, and gas-related properties were not explicitly modeled.

7 Conclusions

This project applied property-based fuzzing to the security analysis of two smart contracts, *Taxpayer* and *Lottery*, with the goal of identifying state inconsistencies, missing validations, and protocol-level vulnerabilities.

For the **Taxpayer** contract, explicitly defined invariants enabled Echidna to automatically discover issues related to asymmetric marriages, incorrect allowance redistribution, and age-based rule violations. All identified vulnerabilities were fixed by strengthening input validation and enforcing coherent state transitions.

The **Lottery** contract highlighted the importance of modeling protocols as explicit state machines. Missing phase enforcement initially allowed invalid call sequences such as early reveals and double participation. After introducing strict phase checks and validating strong invariants, the protocol behaved correctly under extensive fuzzing.

Overall, this work confirms that smart contract security is effectively expressed in terms of *properties describing what must never happen.* Property-based fuzzing proved particularly effective at exploring complex and unexpected execution traces that are difficult to capture with manual testing.

While fuzzing does not provide formal guarantees, the adopted workflow offers a practical and scalable approach to improving smart contract robustness.