Sapienza Università di Roma

Security in Software Applications Proj 3

Alex Parri - 2140961 - Master’s Degree in Cybersecurity

A.Y. 2024/25

Abstract

This is the report for the **third project** of the Security in Software Applications course directed by Daniele Friolo for the Academic Year 24/25 for the Master's Degree in **Cybersecurity** at Sapienza University of Rome. In this homework, the goal was to experiment with **smart contracts** through **fuzz testing**, a form of software dynamic analysis.

Specifically, it was asked to use the Echidna tool to **test** a given smart contract with the goal of validating its properties through the use of **requires** and **invariants**.

The **hardware** utilized for testing is Ryzen 5800X 8-Core 16-Thread @ 4.850GHz CPU with Echidna 2.2.6 in Ubuntu 24.04.02 LTS x86_64 and 16GB of RAM.

Echidna

Fuzzing is a software testing technique in which a software is inputted with **random inputs** in order to find bugs, hangs, crashes in order to find and subsequently patch security flaws.

In this project it is done with **Echidna**, a Haskell tool designed for analyzing Ethereum smart contracts by invoking their publicly available functions to look for violations of Solidity **require** assertions or developer-defined **invariants**. The tool is particular in its own way as it uses grammar-based campaigns thus generating inputs tailored to the contract's **actual code**.

First part

The provided `Person.sol` smart contract code is the following

```
pragma solidity ^0.8.22;
// SPDX-License-Identifier: UNLICENSED

contract Person {
  uint age;
  bool isMarried;

  /* Reference to spouse if person is married, address(0) otherwise */
  address spouse;
  address mother;
  address father;

  /* welfare subsidy */
```

```
  uint constant  DEFAULT_SUBSIDY = 500;
  uint state_subsidy;

  constructor(address ma, address fa) {
    age = 0;
    isMarried = false;
    mother = ma;
    father = fa;
    spouse = address(0);
    state_subsidy = DEFAULT_SUBSIDY;
  }

  // We require new_spouse != address(0);
  function marry(address new_spouse) public {
    spouse = new_spouse;
    isMarried = true;
  }

  function divorce() public {
    Person sp = Person(address(spouse));
    sp.setSpouse(address(0));
    spouse = address(0);
    isMarried = false;
  }

  function haveBirthday() public {
    age++;
  }

  function setSpouse(address sp) public {
    spouse = sp;
  }

  function getSpouse() public returns (address) {
    return spouse;
  }
}
```

In this part the assignment was to **ensure** the following statement: *if person x is married to person y, then person y should of course also be married and to person x.*

Note: each country decides for its own laws, which includes marriage. In this context, the law of **Italy** was followed when considering which constraints to include.

Echidna invariants

The defined invariants in `Person.sol` are as follows

```
function echidna_isMarried_consistency() public view returns (bool) {
  return isMarried == (spouse != address(0));
}

function echidna_no_self_marriage() public view returns (bool) {
  return spouse != address(this);
}

function echidna_no_mother_marriage() public view returns (bool) {
  return spouse != mother;
}

function echidna_no_father_marriage() public view returns (bool) {
  return spouse != father;
}
```

However they are not enough, for two reasons regarding Echidna:

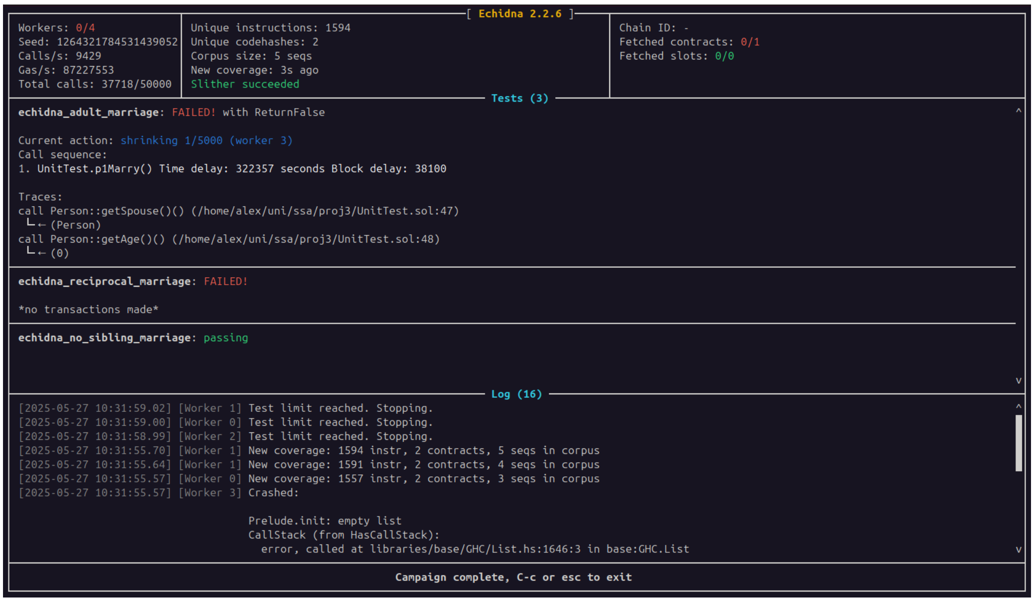
- it does **not support** multiple deployed contracts
- it requires constructors with **empty arguments**

Therefore to be able to test **contract interaction** and **mutual constraints** it was needed to create a `UnitTest.sol` contract which then imported `Person.sol`, whose invariants are as follows

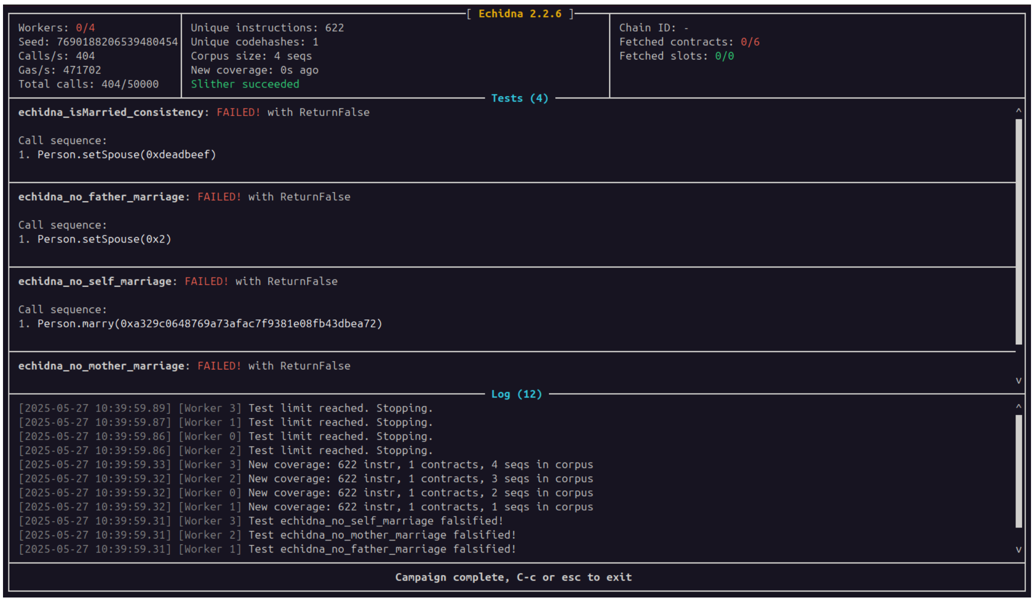
```
function echidna_reciprocal_marriage() public view returns (bool) {
  if (p1.getSpouse() != address(0)) {
    return Person(p1.getSpouse()).getSpouse() == address(p1);
  }
  else return true;
}

function echidna_adult_marriage() public view returns (bool) {
  if (p1.getSpouse() != address(0)) {
      return p1.getAge() >= 18 && Person(p1.getSpouse()).getAge() >= 18;
  }
  else return true;
}

function echidna_no_sibling_marriage() public view returns (bool) {
  if (p1.getSpouse() != address(0)) {
```

```
    return p1.getMother() != Person(p1.getSpouse()).getMother();
  }
  else return true;
}
```

By then running Echidna it is shown below that **2/3 invariants** are violated



When instead running it on `Person.sol`, **all of them** are violated



This means that there is **quite some** fixing to be done to the contract, all of which is explained below.

Fixing the contract

The following **changes** were applied, going from top to bottom of the contract.

Contract fields

```
uint8 internal age; // made internal

/* Reference to spouse if person is married, address(0) otherwise */
address internal spouse;  // made internal
address immutable mother; // made immutable
address immutable father; // made immutable
```

The `age` and `spouse` fields **shall not** be modifiable from outside the method, therefore they were directly specified as **`internal`** (by default they already are, but it's clearer now). The `getMother()` and `getAge()` **getter methods** were also introduced, `haveBirthday()` was included a limit to avoid overflows (the max value for an `uint8` is $2^8 - 1$), finally `getSpouse()` was turned into **`view`**

```
// added
function getAge() public view returns (uint) {
  return age;
}

function haveBirthday() public {
  // added
  require(age < 120, "Cannot increase age any further than 119.");

  age++;
}

// added
function getMother() public view returns (uint) {
  return mother;
}

// turned into view
function getSpouse() public view returns (address) {
  return spouse;
}
```

Meanwhile, the `mother` and `father` fields were made **`immutable`** to ensure that they **don't get** accidently modified inside the contract. It is also assumed that the latter are **biological parents**, so they do not ever change after assignment.

Constructor

The following `require` constraints have been added, as well as the `uint8 _age` field to avoid having to call `haveBirthday()` if the contract was **not** created at birth

```
constructor(address ma, address fa, uint8 _age) {
  require(ma != address(0), "Please specify a non-zero mother address.");
  require(fa != address(0), "Please specify a non-zero father address.");
  require(ma != fa, "Mother and father must be different people.");
  require(_age <= 120, "Please provide an age lower than 121.");

  age = _age;
  mother = ma;
  father = fa;
  spouse = address(0);
  isMarried = false;
  state_subsidy = DEFAULT_SUBSIDY;
}
```

Marrying

This was the most complicated method to fix due to the **large amount** of requirements needed to marry a significant other, at least in the country of Italy

```
function marry(address new_spouse) public {
  // single constraints
  require(new_spouse != address(0), "Please specify a non-zero spouse.");
  require(new_spouse != address(this), "You cannot marry yourself.");
  require(new_spouse != mother, "You cannot marry your own mother.");
  require(new_spouse != father, "You cannot marry your own father.");
  require(spouse == address(0), "You are already married.");
  require(age >= 18, "You must be at least 18 years old to get married.");

  // significant other constraints
  Person other = Person(new_spouse);
  require(other.getMother() != mother, "You cannot marry a sibling.");
  require(other.getAge() >= 18, "Your spouse must be at least 18 years old
   ↪ to get married.");
  require(other.getSpouse() == address(0), "The other person is already
   ↪ married with someone else.");

  // actually marry
```

```
  spouse = new_spouse;
  other.acceptMarriage(address(this));
  isMarried = true;
}
```

Divorcing

Fortunately this **just requires** to be married to someone, so it was a quick fix

```
function divorce() public {
  require(spouse != address(0), "You are not currently married.");

  // actually divorce
  address temp = spouse;
  spouse = address(0);
  Person(temp).acceptDivorce(address(this));
  isMarried = false;
}
```

`setSpouse()`

The `setSpouse()` method was **removed** as it needed to be **`public`** in order to ensure **mutual marriage** and **mutual divorce**, which allowed anyone to change anyone else's `spouse` freely.

To fix that, `acceptMarriage()` and `acceptDivorce()` were introduced as shown

```
// requires the significant other to already have spouse = address(this)
function acceptMarriage(address sp) public {
  require(Person(sp).getSpouse() == address(this), "The other part must
  ↪  have accepted to marry you.");
  require(isMarried == false, "You are already married to this person.");

  spouse = sp;
  isMarried = true;
}

// requires the significant other to already have spouse = address(0)
function acceptDivorce(address sp) public {
  require(spouse == sp, "You cannot divorce someone you are not married
  ↪  with.");
  require(Person(sp).getSpouse() == address(0), "The other part must have
  ↪  accepted to divorce you.");
```

```
  spouse = address(0);
  isMarried = false;
}
```

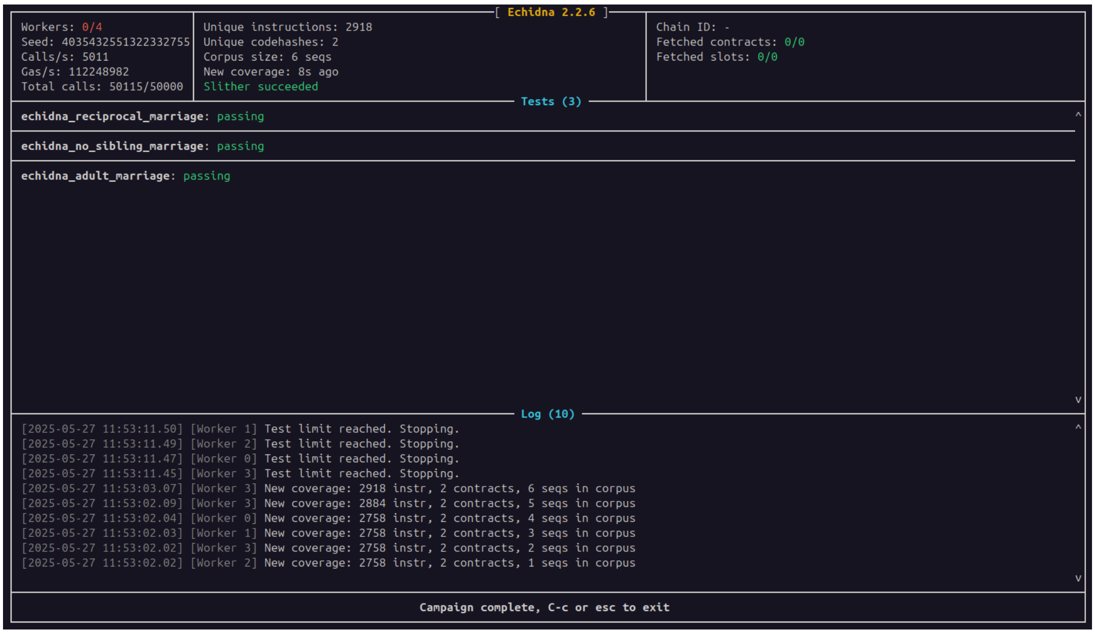
Their **correct functioning** is based on the following principles:

- when you **marry** someone, you first set `spouse` to your significant other's address, **then** the other party does the same with you ("accepts" it)
- when you **divorce** someone, you first set `spouse` to `address(0)` **then** the other part does the same with you ("accepts" it)

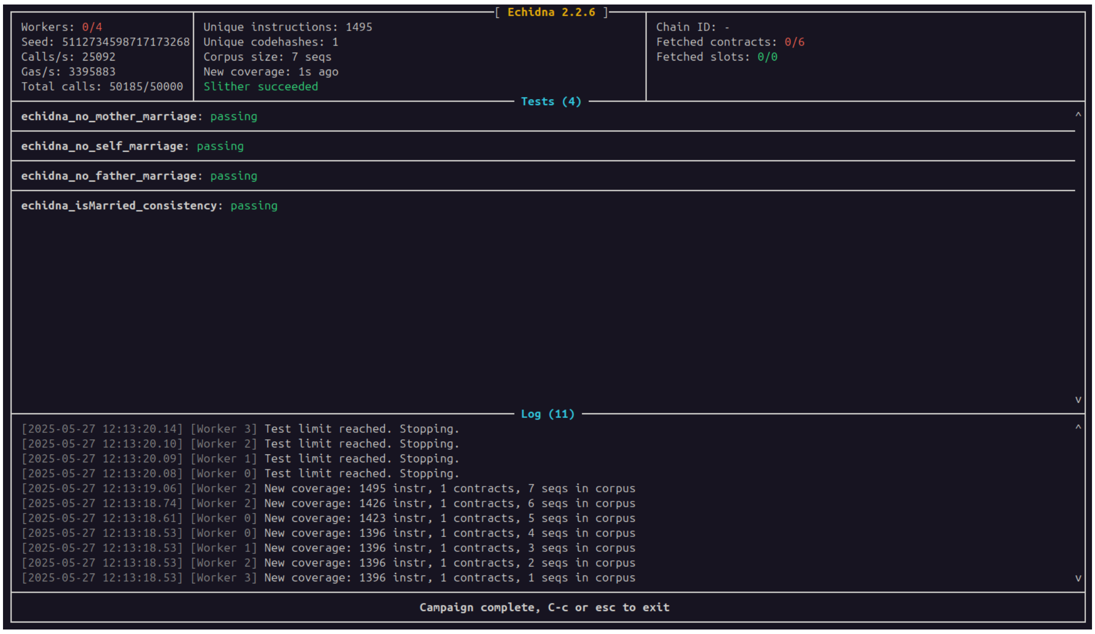
This **ensures** that marrying or divorcing the same person twice is **not possible** even by accident, and that onlookers cannot just set your `spouse` through `setSpouse()` as they wish. In order for this to function, it is **critical** that `acceptMarriage()` and `acceptDivorce()` are called **after** the local `spouse` assignment within the `marry()` and `divorce()` functions **respectively**.

This also makes possible to call `marry()` or `divorce()` once and the other party **accepts it automatically**, avoiding any intermediate unconsistent state with the `spouse` variable.

A quick rundown of Echidna on `UnitTest.sol` reveals that **every** invariant is now respected



The same happens for `Person.sol` specific invariants



Second part

This second part required to **modify allowances** based on marriage and age, based off of the following three constraints:

1. *Every person receives a default subsidy of 500 until age 65.*
2. *After the age of 65, the default subsidy increases to 600 if unmarried.*
3. *Married persons receive each the default subsidy reduced by 30%.*

To clearly **distinguish** between the two subsidies, a **`constant`** variable **for each** was introduced

```
uint constant DEFAULT_SUBSIDY_YOUNGER_65 = 500;
uint constant DEFAULT_SUBSIDY_OLDER_65 = 600;
```

The following **`if`** allows newly created people to get assigned the correct subsidy **right away**

```
// instead of state_subsidy = DEAFULT_SUBSIDY;
if(age > 65) state_subsidy = DEFAULT_SUBSIDY_OLDER_65;
else state_subsidy = DEFAULT_SUBSIDY_YOUNGER_65;
```

Married people will see their subsidy reduced by 30% (i.e. multiplied by $0.7 = \frac{7}{10}$) thanks to this line added at the very end of `marry()` and `acceptMarriage()`

```
// decrease by 30%
state_subsidy = state_subsidy * 7 / 10;
```

People who divorce will have their default subsidy **restored** due to the following `if` added at the very end of `divorce()` and `acceptDivorce()`

```
// increase back
if(age > 65) state_subsidy = DEFAULT_SUBSIDY_OLDER_65;
else state_subsidy = DEFAULT_SUBSIDY_YOUNGER_65;
```

The following check at the end of `haveBirthday()` is needed, but only for people crossing their **66th birthday** to avoid reassigning the correct subsidy at every birthday

```
if(age == 66) {
  if(!isMarried) state_subsidy = DEFAULT_SUBSIDY_OLDER_65;
  else state_subsidy = DEFAULT_SUBSIDY_OLDER_65 * 7 / 10;
}
```

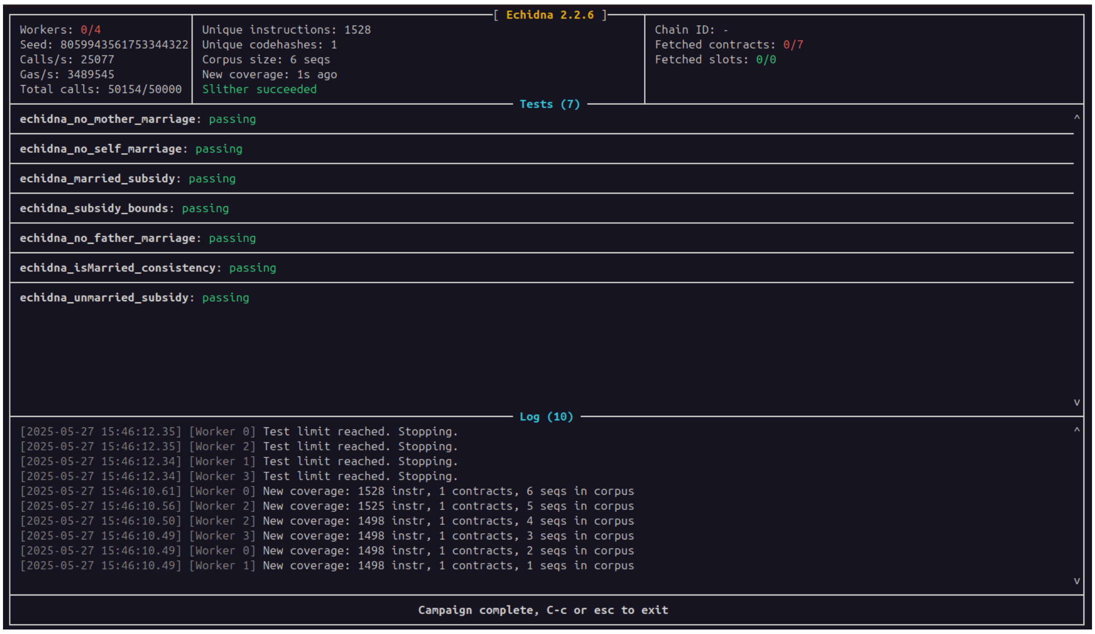
After the 66th birthday mark the `marry()`, `acceptMarriage()` and `divorce()`, `acceptDivorce()` functions are able to **correctly handle** each subsidy change from then on.

The following invariants were added to check for above constraints:

```
function echidna_unmarried_subsidy() public view returns (bool) {
  if(age > 65 && !isMarried) {
    return state_subsidy == DEFAULT_SUBSIDY_OLDER_65;
  }
  else if(age <= 65 && !isMarried) {
    return state_subsidy == DEFAULT_SUBSIDY_YOUNGER_65;
  }
  return true;
}

function echidna_married_subsidy() public view returns (bool) {
  if(age > 65 && isMarried) {
    return state_subsidy == DEFAULT_SUBSIDY_OLDER_65 * 7 / 10;
  }
  else if(age <= 65 && isMarried) {
    return state_subsidy == DEFAULT_SUBSIDY_YOUNGER_65 * 7 / 10;
  }
  return true;
}

function echidna_subsidy_bounds() public view returns (bool) {
  return state_subsidy == DEFAULT_SUBSIDY_YOUNGER_65 ||
```

```
        state_subsidy == DEFAULT_SUBSIDY_OLDER_65 ||
        state_subsidy == DEFAULT_SUBSIDY_YOUNGER_65 * 7 / 10 ||
        state_subsidy == DEFAULT_SUBSIDY_OLDER_65 * 7 / 10;
}
```

In conclusion, these small changes result in Echidna returning **all green**



Conclusions

Echidna is a **powerful and easy** tool for checking invalid or forgotten requirements when planning to release a smart contract in the Ethereum blockchain. Its use is **pretty much** essential when it comes to real-world applications, given its simplicity and **ease of use**.

This project was very **insightful** in providing a glimse inside the insidiousness of web3 smart contract development, fixing and testing, even more so if `payable` functions were involved.