



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

```

Workers: 0/4 Unique instructions: 1594 Chain ID: -
Seed: 1264321784531439052 Unique codehashes: 2 Fetched contracts: 0/1
Calls/s: 9429 Corpus size: 5 seqs Fetched slots: 0/0
Gas/s: 87227553 New coverage: 3s ago
Total calls: 37718/50000 Slither succeeded

Tests (3)
echidna_adult_marriage: FAILED! with ReturnFalse
Current action: shrinking 1/5000 (worker 3)
Call sequence:
1. UnitTest.p1Marry() Time delay: 322357 seconds Block delay: 38100

Traces:
call Person::getSpouse() (/home/alex/unl/ssa/proj3/UnitTest.sol:47)
└─ (Person)
call Person::getAge() (/home/alex/unl/ssa/proj3/UnitTest.sol:48)
└─ (0)

echidna_reciprocal_marriage: FAILED!
*no transactions made*

echidna_no_sibling_marriage: passing

Log (16)
[2025-05-27 10:31:59.02] [Worker 1] Test limit reached. Stopping.
[2025-05-27 10:31:59.00] [Worker 0] Test limit reached. Stopping.
[2025-05-27 10:31:58.99] [Worker 2] Test limit reached. Stopping.
[2025-05-27 10:31:55.70] [Worker 1] New coverage: 1594 instr, 2 contracts, 5 seqs in corpus
[2025-05-27 10:31:55.64] [Worker 1] New coverage: 1591 instr, 2 contracts, 4 seqs in corpus
[2025-05-27 10:31:55.57] [Worker 0] New coverage: 1557 instr, 2 contracts, 3 seqs in corpus
[2025-05-27 10:31:55.57] [Worker 3] Crashed:
Prelude.init: empty list
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1646:3 in base:GHC.List

Campaign complete, C-c or esc to exit

```

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

```

Workers: 0/4 Unique instructions: 622 Chain ID: -
Seed: 7690188206539480454 Unique codehashes: 1 Fetched contracts: 0/6
Calls/s: 404 Corpus size: 4 seqs Fetched slots: 0/0
Gas/s: 471702 New coverage: 0s ago
Total calls: 404/50000 Slither succeeded

Tests (4)
echidna_isMarried_consistency: FAILED! with ReturnFalse
Call sequence:
1. Person.setSpouse(0xdeadbeef)

echidna_no_father_marriage: FAILED! with ReturnFalse
Call sequence:
1. Person.setSpouse(0x2)

echidna_no_self_marriage: FAILED! with ReturnFalse
Call sequence:
1. Person.marry(0xa329c0648769a73afac7f9381e08fb43dbee72)

echidna_no_mother_marriage: FAILED! with ReturnFalse

Log (12)
[2025-05-27 10:39:59.89] [Worker 3] Test limit reached. Stopping.
[2025-05-27 10:39:59.87] [Worker 1] Test limit reached. Stopping.
[2025-05-27 10:39:59.86] [Worker 0] Test limit reached. Stopping.
[2025-05-27 10:39:59.86] [Worker 2] Test limit reached. Stopping.
[2025-05-27 10:39:59.33] [Worker 3] New coverage: 622 instr, 1 contracts, 4 seqs in corpus
[2025-05-27 10:39:59.32] [Worker 2] New coverage: 622 instr, 1 contracts, 3 seqs in corpus
[2025-05-27 10:39:59.32] [Worker 0] New coverage: 622 instr, 1 contracts, 2 seqs in corpus
[2025-05-27 10:39:59.32] [Worker 1] New coverage: 622 instr, 1 contracts, 1 seqs in corpus
[2025-05-27 10:39:59.31] [Worker 3] Test echidna_no_self_marriage falsified!
[2025-05-27 10:39:59.31] [Worker 2] Test echidna_no_mother_marriage falsified!
[2025-05-27 10:39:59.31] [Worker 1] Test echidna_no_father_marriage falsified!

Campaign complete, C-c or esc to exit

```

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** accidentally 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 inconsistent state with the spouse variable.

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

The screenshot displays the Echidna 2.2.6 interface. At the top, it shows campaign statistics: 0/4 workers, seed 4035432551322332755, 5011 calls/s, 112248982 gas/s, and 50115/50000 total calls. It also lists unique instructions (2918), codehashes (2), corpus size (6 seqs), and new coverage (8s ago). The Chain ID is -, and it shows 0/0 fetched contracts and slots. Below this, three tests are listed as 'passing': `echidna_reciprocal_marriage`, `echidna_no_sibling_marriage`, and `echidna_adult_marriage`. A 'Log (10)' section shows a series of messages from four workers (0-3) indicating that test limits were reached and they were stopping. The final message states 'Campaign complete, C-c or esc to exit'.

The same happens for `Person.sol` specific invariants

```

Workers: 0/4      Unique instructions: 1495      [ Echidna 2.2.6 ]      Chain ID: -
Seed: 5112734598717173268      Unique codehashes: 1      Fetched contracts: 0/6
Calls/s: 25092      Corpus size: 7 seqs      Fetched slots: 0/0
Gas/s: 3395883      New coverage: 1s ago
Total calls: 50185/50000      Slither succeeded

Tests (4)
echidna_no_mother_marriage: passing
echidna_no_self_marriage: passing
echidna_no_father_marriage: passing
echidna_isMarried_consistency: passing

Log (11)
[2025-05-27 12:13:20.14] [Worker 3] Test limit reached. Stopping.
[2025-05-27 12:13:20.10] [Worker 2] Test limit reached. Stopping.
[2025-05-27 12:13:20.09] [Worker 1] Test limit reached. Stopping.
[2025-05-27 12:13:20.08] [Worker 0] Test limit reached. Stopping.
[2025-05-27 12:13:19.06] [Worker 2] New coverage: 1495 instr, 1 contracts, 7 seqs in corpus
[2025-05-27 12:13:18.74] [Worker 2] New coverage: 1426 instr, 1 contracts, 6 seqs in corpus
[2025-05-27 12:13:18.61] [Worker 0] New coverage: 1423 instr, 1 contracts, 5 seqs in corpus
[2025-05-27 12:13:18.53] [Worker 0] New coverage: 1396 instr, 1 contracts, 4 seqs in corpus
[2025-05-27 12:13:18.53] [Worker 1] New coverage: 1396 instr, 1 contracts, 3 seqs in corpus
[2025-05-27 12:13:18.53] [Worker 2] New coverage: 1396 instr, 1 contracts, 2 seqs in corpus
[2025-05-27 12:13:18.53] [Worker 3] New coverage: 1396 instr, 1 contracts, 1 seqs in corpus

Campaign complete, C-c or esc to exit

```

## 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**

[ Echidna 2.2.6 ]		
Workers: 0/4 Seed: 8059943561753344322 Calls/s: 25077 Gas/s: 3489545 Total calls: 50154/50000	Unique instructions: 1528 Unique codehashes: 1 Corpus size: 6 seqs New coverage: 1s ago Slither succeeded	Chain ID: - Fetched contracts: 0/7 Fetched slots: 0/0
Tests (7)		
echidna_no_mother_marriage: passing		
echidna_no_self_marriage: passing		
echidna_married_subsidy: passing		
echidna_subsidy_bounds: passing		
echidna_no_father_marriage: passing		
echidna_isMarried_consistency: passing		
echidna_unmarried_subsidy: passing		
Log (10)		
[2025-05-27 15:46:12.35] [Worker 0] Test limit reached. Stopping.		
[2025-05-27 15:46:12.35] [Worker 2] Test limit reached. Stopping.		
[2025-05-27 15:46:12.34] [Worker 1] Test limit reached. Stopping.		
[2025-05-27 15:46:12.34] [Worker 3] Test limit reached. Stopping.		
[2025-05-27 15:46:10.61] [Worker 0] New coverage: 1528 instr, 1 contracts, 6 seqs in corpus		
[2025-05-27 15:46:10.56] [Worker 2] New coverage: 1525 instr, 1 contracts, 5 seqs in corpus		
[2025-05-27 15:46:10.50] [Worker 2] New coverage: 1498 instr, 1 contracts, 4 seqs in corpus		
[2025-05-27 15:46:10.49] [Worker 3] New coverage: 1498 instr, 1 contracts, 3 seqs in corpus		
[2025-05-27 15:46:10.49] [Worker 0] New coverage: 1498 instr, 1 contracts, 2 seqs in corpus		
[2025-05-27 15:46:10.49] [Worker 1] New coverage: 1498 instr, 1 contracts, 1 seqs in corpus		
Campaign complete, C-c or esc to exit		

## 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 glimpse inside the insidiousness of web3 smart contract development, fixing and testing, even more so if **payable** functions were involved.