SSA Report

Pietro Colaguori 1936709

January 2024

1 What are fuzzing and Echidna?

In this project I used the tool Echidna, a program written in Haskell used to perform fuzzing on Ethereum smart contracts, which are written in Solidity. Fuzzing is the practice of entering large amounts of unexpected inputs and recording what happens, in order to find vulnerabilities in the code.

A fuzzer, like Echidna, is a program that automatically injects semi-random data into a program to detect bugs. The data generation part is made of generators that combine static fuzzing vectors, i.e. using values that are known to be "dangerous", with completely random data. The vulnerability identification part relies on debugging tools.

In particular, Echidna enforces some properties which are conditions set by the developer that should always be true, for instance, a token contract may have a property dictating that the total token supply should never decrease. Properties in Solidity are articulated as functions that yield a boolean value, if they return *true* then the property has been valid for the whole run, otherwise, if they return *false*, the property has been violated at some point.

An alternative method for enforcing properties is by including *–test-mode assertion* when running *echidna-test* on some Ethereum smart contract. In this way Echidna focuses on validating assertion-type properties, which is beneficial if the primary objective is to enforce and confirm assertion-style properties defined in Solidity code.

The last method is the one I used in my project as it offers various benefits, such as:

- This mode offers a more targeted and focused approach to property-based testing, especially when dealing with specific conditions expressed as assertions.
- Developers can efficiently verify that critical invariants and conditions hold true during the execution of the contract.

I will now proceed by showing snippets of the code I wrote and of the modified code of the original contract, moreover I will provide screenshots of the output of Echidna, which I am running on Windows inside a Docker container.

2 My code and Echidna outputs

First of all, I have applied some changes to the original code of the contract *Taxpayer*: I changes the type of argument for the function *marry* from *address* to *Taxpayer* and I also added the following functions to the interface for comfort.

```
function getIsMarried() public view returns (bool) {
    return isMarried;
}
function getAge() public view returns (uint) {
    return age;
}
function setAge(uint a) public {
    age = a;
    if(a >= 65) {
        this.setTaxAllowance(ALLOWANCE_OAP);
    }
}
function isMinor() public view returns (bool) {
    return age <= 17;
}
function setIsMarried(bool x) public {
    isMarried = x;
}
```

I also modified the function *haveBirthday* so that if the new age of the contract is greater or equal than 65 then the tax allowance will be set to the appropriate one and not anymore the default one.

Note that to perform the testing I created a new contract *TaxpayerTest* that inherits from the original contract, in which I fix the parents set in the constructor.

The workflow I used is the following:

1. Write a function that takes in input some arguments that I want to be selected at random by Echidna, write one for each function to test.
2. Inside the testing function use the function of the original contract that we want to fuzz and verifiy, using *assert(bool condition)*, that some properties hold before and after the original function has been called.

3. Echidna will sample the arguments in the way previously described and feed them to the testing function, two things can now happen:

   - One or more assertions have been falsified, we check the sequence of calls used by Echidna to do so and we insert the proper *require* statements into the original function. Such statements will impose a condition on the arguments used inside the function and prevent the condition from being falsified again. Then, we repeat from step 3 to see if we have done so properly.
   - No assertion is falsified, this means that either the provided function is safe or we have already set the *require* statements needed.

I will now show and comment the function used for fuzzing the *marry* function of the original smart contract. As input to the function I gave the *Taxpayer* object that is going to be married by the calling contract, the age of the two contracts and the "ex" spouses of the two contracts.
Inside the function I gave some information about the two contracts that will be useful to check later, in the assertions.
I have created many assertions to be able to capture the correct scenario.

```
function assert_fuzzMarry(Taxpayer new_spouse, uint
    myAge, uint spouseAge, address myOldSpouse,
    address spouseSpouse) public {
        this.setAge(myAge);
        this.setSpouse(myOldSpouse);
        if(myOldSpouse == address(0)) {
            this.setIsMarried(false); }
        else { this.setIsMarried(true); }
        new_spouse.setAge(spouseAge);
        new_spouse.setSpouse(spouseSpouse);
        if(spouseSpouse == address(0)) {
            new_spouse.setIsMarried(false); }
        else { new_spouse.setIsMarried(true); }

        if(address(new_spouse) != address(0)) {
            bool preMarried = this.getIsMarried();
            address preSpouse = this.getSpouse();
            bool preSpouseMarried =
                new_spouse.getIsMarried();
            address preSpouseSpouse =
                new_spouse.getSpouse();

            this.marry(new_spouse);

            bool postMarried = this.getIsMarried();
```

```
            address postSpouse = this.getSpouse();
            bool postSpouseMarried =
                new_spouse.getIsMarried();
            address postSpouseSpouse =
                new_spouse.getSpouse();

            assert(address(new_spouse) != address(0));
            assert(this.getAge() >= 18);
            assert(new_spouse.getAge() >= 18);
            assert(!preMarried);
            assert(preSpouse == address(0));
            assert(postMarried);
            assert(address(postSpouse) ==
                address(new_spouse));
            assert(!preSpouseMarried);
            assert(preSpouseSpouse == address(0));
            assert(postSpouseSpouse == address(this));
            assert(postSpouseMarried);
        }
    }
```

It is important that the assertions express the desired situation before and after the function *marry* is run. I'll briefly describe the assertions I used.

- *assert(this.getAge()* $\geq$ *18)* ensures that the marrying contract is not a minor.
- *assert(new_spouse.getAge()* $\geq$ *18)* ensures that the married contract is not a minor.
- *assert(!preMarried)* ensures that the marrying contract is not married when calling the function, while *assert(postMarried)* verifies that the contract is married after calling the function.
- *assert(address(postSpouse) == address(new_spouse))* and *assert(postSpouseSpouse == address(this))* ensure that the spouse is set correctly both for the calling contract and for the contract being married.
- Every assertion using *address(0)* checks if the address being used is not valid.

Running the function shown without requires in the *marry* function will yield an execution in which Echidna is able to falsify our assertions.
Above I provide some examples.

If we run our code like this, without inserting any *require*, obviously the assertions will be falsified, in particular the output produced by Echidna was

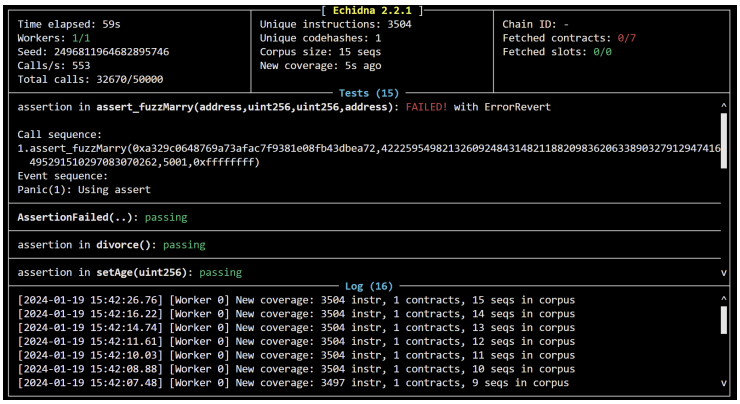

Figure 1: Output of Echidna without requires in marry (calling contract is already married).

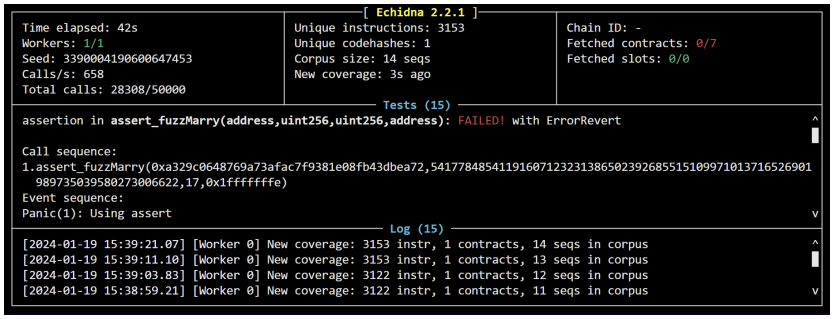

Figure 2: Output of Echidna, marry fails if spouse is a minor.

the following:

If the age of the spouse is less than 18, clearly the third assertion will be falsified, this is what Echidna did and what prompted me to insert a *require* inside *marry* to verify that the spouse is not a minor.

From the arguments selected we can see that the age set for the spouse is 17 which causes the falsification of the assertion. Note that the version shown in the outputs didn't have the last argument, which I added later.

Proceeding with the test I was able to add all the *requires* needed inside the original *marry* function so that Echidna was unable to "break" it. Below is the updated version of *marry* with all the require statements needed.

```
1 function marry(Taxpayer spouse_contract) public {
```

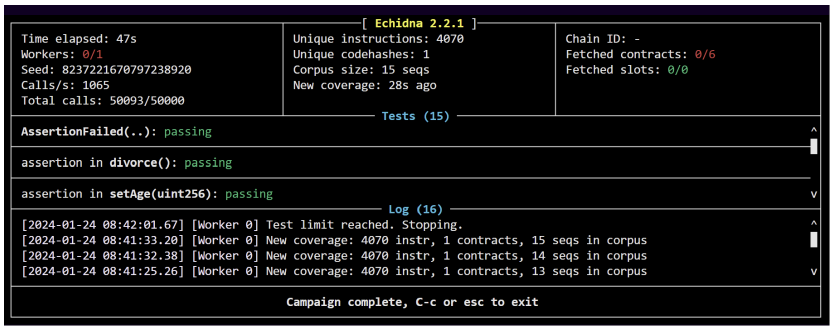

Figure 3: Marry passes all the tests.

```
    require(this.getAge() >= 18, "You are a minor");
    require(!spouse_contract.isMinor(), "You cannot
        marry a minor");
    require(address(spouse_contract) != address(0),
        "Invalid spouse address [0x0]");
    require(isMarried == false , "You are already
        married, divorce first [1]");
    require(this.getSpouse() == address(0), "You are
        already married, divorse first [2]");
    require(!spouse_contract.getIsMarried(), "New
        spouse is already married");
    spouse = address(spouse_contract);
    spouse_contract.setSpouse(address(this));
    require(spouse_contract.getSpouse() ==
        address(this), "You married someone else's
        spouse");
    isMarried = true;
    require(spouse_contract.getIsMarried(), "You
        married someone who didn't marry you");
}
```

I proceeded in a similar way for the *divorce* function, noting that since it takes no arguments I will have to perform fuzzing on the arguments of the function *marry*. The testing function I created is the one shown below, which initially led to assertion failures as the person calling the divorce function must not be married.
Note that majority of the security of *divorce* is implied by the security of *marry*, but for the principle of redundancy I decided to still be as exhaustive as possible with my assertions and requirements.

```
    function assert_fuzzDivorce(Taxpayer
        currentSpouse) public {
        this.setSpouse(address(currentSpouse));
        if(this.getSpouse() != address(0)) {
            this.setIsMarried(true); }
        bool preMarried = this.getIsMarried();
        address preSpouse = this.getSpouse();

        this.divorce();

        bool postMarried = this.getIsMarried();
        address postSpouse = this.getSpouse();
        bool postSpouseMarried =
            currentSpouse.getIsMarried();
        address postSpouseSpouse =
            currentSpouse.getSpouse();

        assert(preMarried);
        assert(preSpouse == address(currentSpouse));
        assert(!postMarried);
        assert(postSpouse == address(0));
        assert(!postSpouseMarried);
        assert(postSpouseSpouse == address(0));
    }
```

Here the assertions we set up aim at ensuring that before calling the function the contract is married to someone, using *assert(preMarried); assert(preSpouse == address(currentSpouse);* and that after calling the function the contract is not married anymore, by using *assert(!postMarried); assert(postSpouse == address(0);*. The two final assertions check the same conditions on the contract of the initial spouse.

Without any requirements Echidna will break the function as we are not even checking if the contract that calls the divorce function is actually married. Below I show the outputs of Echidna as well as the final version of *divorce*.

```
function divorce() public {
   require(isMarried, "You are not married, no one to
       divorce");
   Taxpayer sp = Taxpayer(address(spouse));
   require(sp.getSpouse() == address(this), "You are
       divorcing someone that is not married to you");
   sp.setSpouse(address(0));
```

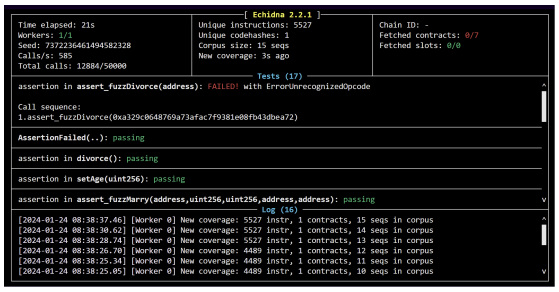

Figure 4: Echidna falsifies the assertions.

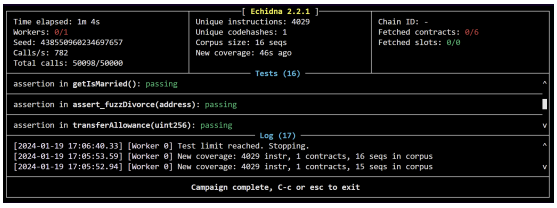

Figure 5: Echidna is not able to falsfiy the assertions.

```
    require(sp.getSpouse() == address(0), "Something
        went wrong, spouse did not divorce you");
    spouse = address(0);
    require(this.getSpouse() == address(0), "Something
        went wrong, you did not divorce correctly");
    isMarried = false;
    require(!this.getIsMarried(), "Your status did not
        update correctly");
    require(!sp.getIsMarried(), "Your spouse did not
        update the status");
}
```

Lastly, we need to check that the *transferAllowance* function is correctly implemented, with respect also to the age of the contracts that are married. To do so I let Echidna pick some arguments (shown below), including the *uint change* and I can easily verify the need to check that the value transferred is not greater than the tax allowance of the transferring contract. Echidna will in fact pick a great value for *change* and falsify the assertion.

```
    function assert_fuzzTransferTaxAllowance(Taxpayer
        currentSpouse, uint change, uint spouseAge,
        uint myAge) public {
        this.setAge(myAge);
        currentSpouse.setAge(spouseAge);
        if(spouseAge >= 65) {
            currentSpouse.setTaxAllowance(ALLOWANCE_OAP);
            }
        else {
            currentSpouse.setTaxAllowance(DEFAULT_ALLOWANCE);
            }
        this.marry(currentSpouse);

```

```
        bool bothOver65 = (this.getAge() >= 65 &&
            currentSpouse.getAge() >= 65);
        bool oneOver65 = ((this.getAge() >= 65 &&
            currentSpouse.getAge() < 65) ||
                          (currentSpouse.getAge() >=
                              65 && this.getAge() <
                              65));
        bool noneOver65 = (this.getAge() < 65 &&
            currentSpouse.getAge() < 65);

        uint myInitial = this.getTaxAllowance();
        uint spouseInitial =
            currentSpouse.getTaxAllowance();
        this.transferAllowance(change);
        uint myFinal = this.getTaxAllowance();
        uint spouseFinal =
            currentSpouse.getTaxAllowance();

        assert(change <= this.getTaxAllowance());
        assert(myInitial + spouseInitial == myFinal +
            spouseFinal);
    }
```

The two final assertions ensure that the caller actually has enough tax allowance to complete the transfer (*assert(change ≥ this.getTaxAllowance())*) and that the transfer is valid (*assert(myInitial + spouseInitial == myFinal + spouseFinal)*), meaning that the sum of the tax allowances remains the same.

The outputs of Echidna are shown below.

Figure 6: Echidna falsifies the assertions.

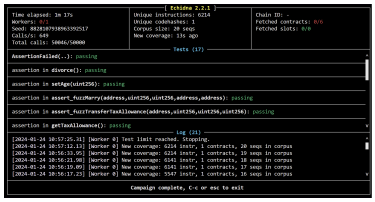

Figure 7: Echidna is not able to falsfiy the assertions.

From Figure 7 we can see how the change passed by Echidna is much bigger than the tax allowance, causing *assert(change ≥ this.getTaxAllowance())* to trigger.