Security in Software Applications - Homework

Elios Buzo - 1669351

May 2024

Abstract

This document sets out in detail an analysis of the security aspects of a given smart contract. Through the use of the Echidna tool, which implements fuzz testing methodology, we were able to examine the robustness of the software and its resistance to possible attempts, voluntary or not, to alter data integrity.

Contents

1 Introduction

1.1 Fuzz testing

Fuzz testing, also known as fuzzing, is an automated software testing technique that aims to unearth bugs, vulnerabilities and other problems within software that may escape traditional testing methods.

More specifically, the fuzzer injects a large set of inputs into the software, which may include random, malformed or simply unexpected inputs that the software does not know how to handle.

During this process the fuzzer observes any reaction that might be of interest to the developer, such as crashes, errors, blockages or excessive resource consumption, and reports all these problems through a log system to the developer.

This system is very effective in detecting problems such as buffer overflows and other memory-related problems such as memory leaks, as well as race conditions and deadlocks.

While it has a lot of positive aspects fuzzing also comes with some negative ones. First of all, since it generates a lot of input data it can reports some false positives, which can subtract precious time to developers. Second of all, always due to the amount of data produced, it can be very resource demanding, demand that increases with the complexity of the software.

Overall it is undoubtedly a valuable and effective technique for identifying vulnerabilities and improving the overall security and reliability of software systems.

1.2 Echidna

Due to their immutable nature, it is well known that smart contracts have to be heavily tested before considering any deployment in production. In fact, having a good software testing coverage with unit tests is essential, but it may not be enough, since we may miss several edge cases. In this case it may be beneficial to extend the testing strategy by using additional tools like fuzzers.

In order to test the degree of security and reliabilty of the smart contract code we used the Echidna tool. Among all smart contract testing software, Echidna has definitely gained a prominent position in terms of popularity and adoption. Specifically, Echidna belongs to a specific family of fuzzers, also called property-based fuzzers.

Compared to classic fuzzers, which attempt to crash software, these kind of fuzzers operated by trying to violate some user-defined properties, or *invariants*. Defined as Solidity functions, these are properties that the smart contract must always satisfy and that Echidna will try to falsify by generating random sequence of calls to these functions.

1.3 How to use Echidna

As we said, in order to test a property we need to define a Solidity function. There are two main ways in which we can define a property that Echidna offers:

- **Boolean properties**
- **Assertions**

The property testing mode is used by default and functions defined in this way have the following characteristics:

- they have no arguments
- they return a boolean
- they have its name starting with *echidna_*
- they pass if they return true and fail if they return false or revert

On the other hand the assertion testing mode will detect any assert violation within the function and they:

- do not require any particular name
- can have any inputs
- do not return anything

The property based mode is useful when a property can be easily computed from the use of state variables and there is no need for extra parameters. while the assertion mode is useful when invariants are better expressed using arguments or when can only be checked in the middle of a transaction.

Once we have decided which mode to use, all the functions need to be inserted inside a test contract which has to inherit the original contract that we want to test, in order to share the same state variables and functions.

```
1  contract Token {
2      mapping(address => uint256) public balances;
3
4      function airdrop() public {
5          balances[msg.sender] = 1000;
6      }
7
8      function consume() public {
9          require(balances[msg.sender] > 0);
10         balances[msg.sender] -= 1;
11     }
12
13     function backdoor() public {
14         balances[msg.sender] += 1;
15     }
16 }
```

```
contract TestToken is Token {
    constructor() public {}

    function echidna_balance_under_1000() public view returns (bool
        ) {
        return balances[msg.sender] <= 1000;
    }
}
```

In the previous two code snippets we can see an example of how Echidna can be used. Indeed a property that this contract example has to satisfy is that every user can hold a maximum of 1000 tokens. So in the test example we can see the function **echidna_check_balance()** that actually checks this. Echidna will automatically generate random transactions in order to test this property and will report any transactions that led the property to return false or throw an error.

Thanks to this example we have also seen another requirement of Echidna, that is the fact that the test contract constructor must have no arguments, so if some inizialization is needed it should be performed in another way, as we can see below in the test contract developed for **Taxpayer.sol**.

```
import "./Taxpayer.sol";

contract TestTaxpayer is Taxpayer(address(1), address(2)) {
    constructor() {}

    function echidna_test_true() public returns (bool) {
        return true;
    }
    ...
    ...
}
```

As we can see the **TestTaxpayer** is an extension of the analyzed contract but the required inizialization of the parents address of the taxpayer, which we set to a placeholder, has been done in the declaration space.

2 Part 1

2.1 Properties analysis

The main property that the contract should always satisfy is the following: *If a person x is married to person y, then person y should of course be married and to person x.* In other words this is saying that the marriage must be a bidirectional relationship between the two people.

Besides this, we found other properties that would deserve to be analyzed, those are:

- *A person cannot marry itself*
- *A person cannot marry the address 0*

- *A person cannot marry one of its parent*
- *A person cannot marry if its underage (less than 18)*

```
contract Taxpayer {
    uint age;
    bool isMarried;
    /* Reference to spouse if person is married, address(0)
        otherwise */
    address spouse;
    address parent1;
    address parent2;
    ...
    constructor(address p1, address p2) {
        age = 0;
        isMarried = false;
        parent1 = p1;
        parent2 = p2;
        spouse = address(0);
        ...
    }
    ...
}
```

As shown by the code snippet, in this smart contract the marriage is represented by two variables: *isMarried* and *spouse*. The first one is a boolean that is set to true and the other one mantains the address of the spouse, if the person is married. By default, in the constructor they are initialized to false and address 0x0 since noone is married at birth.

```
function marry(address new_spouse) public {
    spouse = new_spouse;
    isMarried = true;
}

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

Then we can see how the marriage and divorce events are handled by the functions *marry* and *divorce*. So the first one just set the married status to true and the address of spouse, while the other one set the variable spouse of both the people to 0x0 and the isMarried variable of the current spouse to false.

2.2 Test functions and Code patching

In the following listing we can observe the Echidna functions used to test all the properties that we have previously introduced. We developed a function for each property in order to distinguish the source of the eventual error that would occur.

```
...
function echidna_test_marry_bidirectional() public returns (bool) {
    if (isMarried) {
        return Taxpayer(spouse).getIsMarried() && Taxpayer(spouse).
            getSpouse() == address(this);
    }

    return true;
}

function echidna_test_marry_address0() public returns (bool) {
    return (isMarried && spouse != address(0)) || (!isMarried &&
        spouse == address(0));
}

function echidna_test_marry_parent() public returns (bool) {
    return spouse != parent1 && spouse != parent2;
}

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

function echidna_test_marry_underage() public returns (bool) {
    if (isMarried) {
        return age >= 18;
    }

    return true;
}
...
```

Then, we can observe the output of an Echidna run and we have observed that almost every property has not been satisfied by the contract code.

For what concern the main property after an analysis of the *marry* function we came up with several observations that led to a deep update of the source code. So:

- First of all there is no guarantee that the marry function will be called also by the other spouse, leading to a situation where a person x is married to a person y but y is not married to x.

- Second of all even after solving the previous problem by forcing the marry function call of the other spouse, there is no check regarding whether the person is already married, leading to a situation where a person x is married to a person y and viceversa but then y can marry a person z without passing through the divorce phase first.

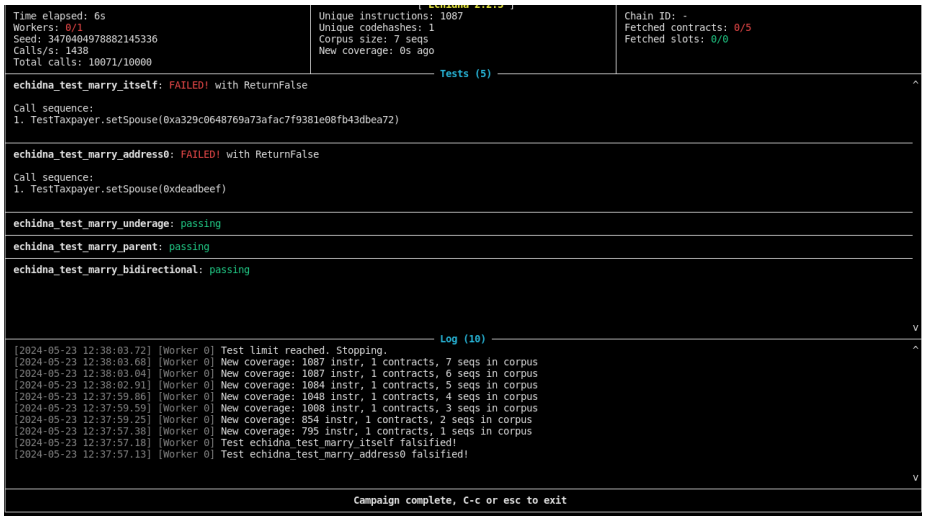
All those observations led to the updated version of *marry* function that we can observe below.

```
function marry(address new_spouse) public {
    Taxpayer spouse_tp = Taxpayer(new_spouse);

    require(new_spouse != address(0));
    require(new_spouse != parent1 && new_spouse != parent2);
    require(new_spouse != address(this));
    require(age >= 18);
    require(!isMarried && spouse == address(0));
    require(!spouse_tp.getIsMarried() && spouse_tp.getSpouse() ==
        address(0));

    spouse = new_spouse;
    isMarried = true;

    if (spouse_tp.getSpouse() != address(this)) {
        spouse_tp.marry(address(this));
    }
}
```

We can see that regarding all the secondary properties we adopted the strategy of using a *require* statement in order to enforce them and that was sufficient to solve the problem.

While in order to enforce the main property we had to still add two *require* statements but also force the call of the marry function associated with the spouse instance. In this way also the other person will execute the same code and the relationship will be correctly established. This approach unfortunately raised a problem: if a person calls the marry function of the spouse then it will do the same since the code is shared. So this would create a cycle and can cause the gas to run out.

In order to break this cycle we added a simple if statement that checks if the spouse is already married with the current person and only if not call the marry function of the spouse.



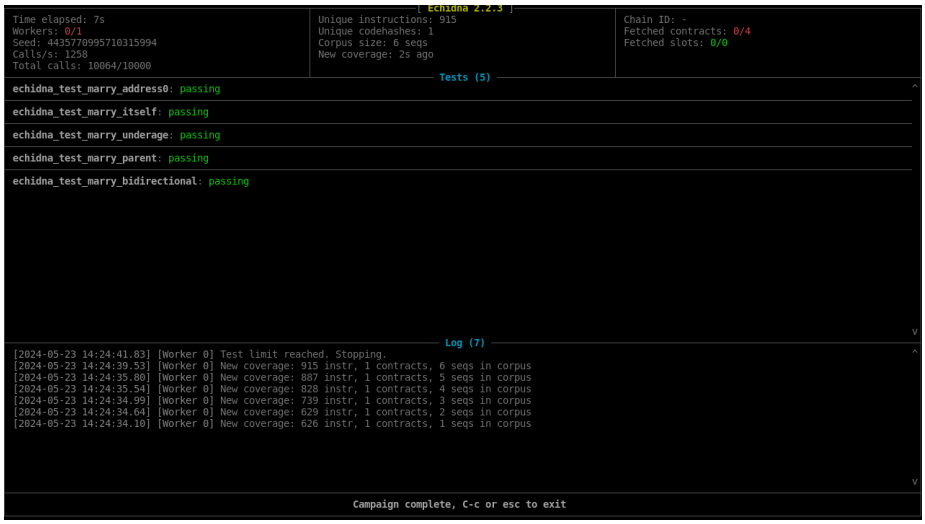
Even after all of those additions Echidna reported that a couple of property were still not satisfied. After a brief analysis we discovered that the problem was represented by the *setSpouse* function.

Indeed this function caused the spouse attribute to change indiscriminately without any control. So we decided to delegate all the marriage related operations only to the marry function and to remove the setSpouse one.

This last decision force us also to modify the *divorce* function since it relied on the setSpouse function. So we decided to make it coherent with the marry function and let it call the divorce function of the spouse in order to nullify the marriage relationship.

```
1  function divorce() public {
2      Taxpayer spouse_tp = Taxpayer(address(spouse));
3
4      require(isMarried && spouse != address(0));
5
6      spouse = address(0);
7      isMarried = false;
8
9      if (spouse_tp.getIsMarried()) {
10         spouse_tp.divorce();
11     }
12 }
```

Finally we can see the report of Echidna that states that all the properties are now satisfied.



3 Part 2

3.1 Properties analysis

The next properties have to be deducted from those statements:

- *Every person has an income tax allowance on which no tax is paid. There is a default tax allowance of 5000 per individual, and only income above this amount is taxed.*
- *Married persons can pool their tax allowance as long as the sum of their tax allowances remains the same. For example, the wife could have a tax allowance of 9000, and the husband a tax allowance of 1000.*

The main property that has to be satisfied here is about the fact that transfering allowance should be a transparent operation, so no amount should be created out of nowhere during this process.

Other secondary but trivial properties are that the amount of tax allowance to be transfered must be lower or at most equal to the current tax allowance and that only those that are married should be able to transfer or receive allowance.

```
1  contract Taxpayer {
2      ...
3      /* Constant default income tax allowance */
4      uint constant DEFAULT_ALLOWANCE = 5000;
5
6      /* Constant income tax allowance for Older Taxpayers over 65 */
7      uint constant ALLOWANCE_OAP = 7000;
8
9      /* Income tax allowance */
10     uint tax_allowance;
11
```

```
12      uint income;
13
14      constructor(address p1, address p2) {
15          ...
16          income = 0;
17          tax_allowance = DEFAULT_ALLOWANCE;
18      }
19      ...
20      /* Transfer part of tax allowance to own spouse */
21      function transferAllowance(uint change) public {
22          tax_allowance = tax_allowance - change;
23          Taxpayer sp = Taxpayer(address(spouse));
24          sp.setTaxAllowance(sp.getTaxAllowance() + change);
25      }
26      ...
27      function setTaxAllowance(uint ta) public {
28          tax_allowance = ta;
29      }
30
31      function getTaxAllowance() public view returns (uint) {
32          return tax_allowance;
33      }
34  }
```

3.2 Test functions and Code patching

As we can see from the listing above when a new instance of the contract is created the related person has a tax allowance equal to the default value, which is 5000. Then the function *transferAllowance* takes the responsibility of the tranfering the allowance from a person to its spouse.

In order to test the properties we faced a problem. The tax allowance is initially set to the default value but in the tests we cannot rely only on checking that the sum of the two values is 10000 since with the function *setTaxAllowance* the allowance can be modified in order to respect some particular real life situation of the related person.

A first idea was to add a new state variable that had to hold the sum of the two allowances if the person is married and so in the test function we had to only check that the sum was equal to this value at any moment. But this approach has a defect because this variable would be used only for the testing phase and then would only result in a waste of gas.

Until now we interact only with the current state in which the smart contract is and not with the transition that took the contract from one state to the other. And so in this way we have no access to the value that the sum has in a previous state.

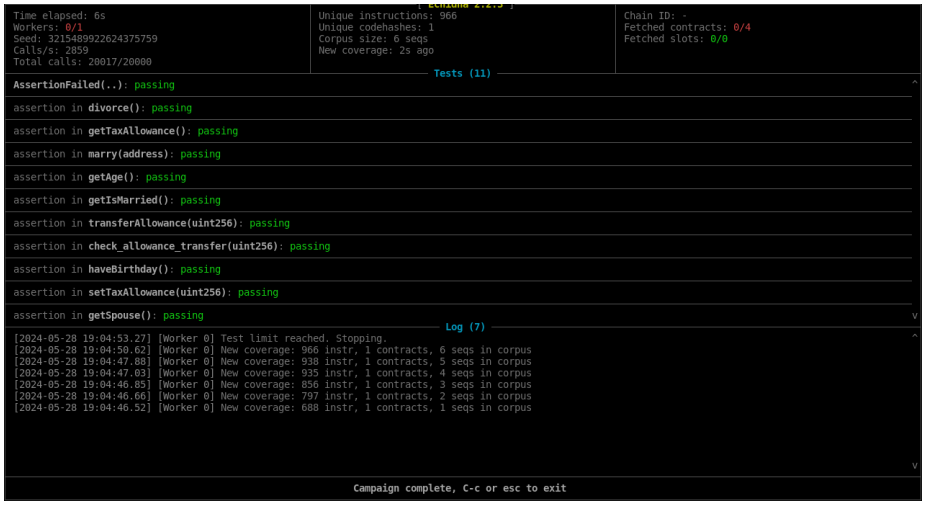
So the solution that we came up with was to force a transition inside the test function by calling *transferAllowance* by ourselves and give it an input of our choice.

Initially we opted for calling it with an hardcoded input, like for example *transferAllowance(1000)*, but then the tests would be based only on a single value and not a variety of values that Echidna instead can provide.

```
function check_allowance_transfer(uint change) public {
    uint old_allowance;
    uint new_allowance;

    if (isMarried) {
      Taxpayer spouse = Taxpayer(getSpouse());
      old_allowance = tax_allowance + spouse.getTaxAllowance();
            transferAllowance(change);
            new_allowance = tax_allowance + spouse.getTaxAllowance
                ();
    } else {
            old_allowance = tax_allowance;
            transferAllowance(change);
            new_allowance = tax_allowance;
        }

    assert(old_allowance == new_allowance);
  }
```

In order to solve this problem, we decided to use the *assertion* testing mode. In this way we can define the test function with an input, in this case the *change*, that we will inject in the *transferAllowance* call. Echidna will generate different values for this parameter and so we can test the contract function in different situations.

As we can see by the listing, we simulated a call to the transferAllowance function, either if the person is married or not in order to observe the result, and then with an assert statement we checked that the transfer has been done correctly.



By observing the result of the Echidna test campaign we can see that the main property, i.e. that the tax allowance is transfered correctly, is already satisfied by the contract. But still we have a subtle problem: if the value that Echidna generates for the *change* is higher than the current value of *taxAllowance* there will be an integer overflow.

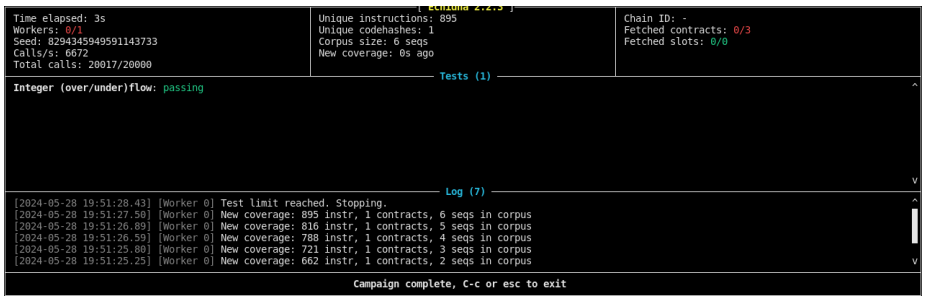
Starting from the version 0.8.0 Solidity started to handle integer overflow situations by causing a *revert*, which cancels every changes made by that transaction until that moment. Echidna reports a revert only if this is caused by an *assert* which is not the case. In order to overcome this limitation, Echidna provides another testing mode, which is called *overflow*, that is specialized in spotting integer overlflows.

And as we predicted there is no control over the value that the function *transferAllowance* get and so this can cause an integer overflow. So, in order to solve this problems we patched the function in the following way.

```
1 function transferAllowance(uint change) public {
2     require(isMarried);
3     require(change <= tax_allowance);
4
5     tax_allowance -= change;
6     Taxpayer sp = Taxpayer(address(spouse));
7     sp.setTaxAllowance(sp.getTaxAllowance() + change);
8 }
```

Now we simply check that the person is married and the change is not higher the current tax allowance in order to execute the transfer.

Additionally we added into the divorce function a statement that reset the tax allowance to its default value.



With all these changes the contract now behaves as expected.

4 Part 3

4.1 Properties analysis

The next properties have to be deducted from this statements:

- *The new government introduced a measure that people aged 65 and over have a higher tax allowance, of 7000. The rules for pooling tax allowances remain the same.*

At a first sight it seems that we need little adjustments to make the previous changes work also for this stage.

Originally the idea was only to distinguish between the cases when we have to deal with the tax allowance for people under 65 and people over 65. So the only changes were to set the right tax allowance when someone celebrates its 65th birthday and when a married person divorce to reset its tax allowance value to the right value distinguishing between over 65 and under 65.

But as we already noted if a person marries, transfers/receives tax allowance and then divorce the original value is lost and we don't know to what value we have to reset it.

```
...
/* Constant default income tax allowance */
uint constant DEFAULT_ALLOWANCE = 5000;

/* Constant income tax allowance for Older Taxpayers over 65 */
uint constant ALLOWANCE_OAP = 7000;

/* Income tax allowance */
uint single_tax_allowance;
uint married_tax_allowance;

uint income;
...
```

So we came up with the idea of decouple the tax allowance related to a person when is single and when is married. In this way we don't loose the previous value and we only have to check if a person is married and then use the appropriate variable every time we have to execute an operation.

```
...
constructor(address p1, address p2) {
    ...
    income = 0;
    single_tax_allowance = DEFAULT_ALLOWANCE;
    married_tax_allowance = 0;
}
...
```

```
function marry(address new_spouse) public {
    ...
    spouse = new_spouse;
    isMarried = true;
    married_tax_allowance = single_tax_allowance;
```

```
    ...
}
```

So now when a new instance is being created we set the *single_tax_allowance* to the default value and the *married_tax_allowance* to zero. Then if a person marries we set the *married_tax_allowance* equal to *single_tax_allowance* in order to have starting point when trying to transfer allowance.

```
function divorce() public {
    ...
    spouse = address(0);
    isMarried = false;
    married_tax_allowance = 0;
    ...
}
```

Then if a married person divorces we only reset the *married_tax_allowance* since we're going to use *single_tax_allowance* to calculate the amount of tax to pay.

```
function transferAllowance(uint change) public {
    require(isMarried);
    require(change <= married_tax_allowance);

    married_tax_allowance -= change;
    Taxpayer sp = Taxpayer(address(spouse));
    sp.setTaxAllowance(sp.getTaxAllowance() + change);
}
```

Now of course when someone wants to transfer its allowance to its spouse we consider the *married_tax_allowance* variable with the same behaviour as before.

```
function haveBirthday() public {
    age++;

    if (age == 65 && single_tax_allowance == DEFAULT_ALLOWANCE) {
        single_tax_allowance = ALLOWANCE_OAP;

        if (isMarried) {
            married_tax_allowance += (ALLOWANCE_OAP -
                DEFAULT_ALLOWANCE);
        }
    }
}
```

And now we can analyze the main point of this phase, that is, the people that are over 65 years old. We needed to modify the *haveBirthday* function in order to catch the case when someone celebrates its 65th birthday. In this case we are going to set the default allowance for people over 65 only if *single_tax_allowance* has not been modified from the default value.

This is because if we find a value different from the default one we don't have to overwrite it or we will loose this value that has influenced also *married_tax_allowance*. If we pass this check then we have to see if the person is married, because we need to update also the *married_tax_allowance* in order to reflect the new *single_tax_allowance* value.

```
function setTaxAllowance(uint ta) public {
    if (isMarried) {
        if (ta >= single_tax_allowance) {
            married_tax_allowance += (ta - single_tax_allowance);
        } else {
            married_tax_allowance -= (single_tax_allowance - ta);
        }
    }

    single_tax_allowance = ta;
}

function getTaxAllowance() public view returns (uint) {
    if (isMarried) {
        return married_tax_allowance;
    }

    return single_tax_allowance;
}
```

The last things that we had to modify were the *setTaxAllowance* and *getTaxAllowance*. For what concern the first we needed to update the *married_tax_allowance* before updating the *single_tax_allowance* in order to not loose its value. Here we had a problem due to the nature of the variables which are all unsigned integer. If the new tax allowance is lower than the actual value we need to subtract the difference from the married tax allowance. But it wouldn't be sufficient to compute the difference and than let the compiler handle the situation when the difference is below zero because *ta* and *single_tax_allowance* are both unsigned integer and so the result can't go below zero. So we simply had to distinguish both the situations as we can se in the code snippet above.

Lastly the second function simply returns the right tax allowance based on the marital situation of the person.

4.2 Test functions and Code patching

The functions on which we had to pay attention were *haveBirthday* and *setTaxAllowance* because they modify actual values of the contract.

Regarding the first one we concluded that we didn't need to write a test for it. The function doesn't take any external parameter and the operation that can cause some problems, i.e. the subtraction between the allowance, should be harmless since the two default allowances are constant value and their subtraction is always positive.

On the other hand the second function had to be tested since the function expects a parameter that in line 6 can potentially cause an integer overflow. Indeed the subtraction *single_tax_allowance* - *ta* can be bigger than *married_tax_allowance* and this must not be allowed.

```
function check_allowance_set(uint ta) public {
    if (isMarried) {
        uint old_married_tax_allowance = married_tax_allowance;
        setTaxAllowance(ta);
        uint new_married_tax_allowance = married_tax_allowance;
```

```
6
7           if (ta >= single_tax_allowance) {
8               assert(new_married_tax_allowance ==
                    old_married_tax_allowance + (ta -
                    single_tax_allowance));
9           } else {
10              assert(new_married_tax_allowance ==
                    old_married_tax_allowance - (single_tax_allowance -
                     ta));
11          }
12      } else {
13          setTaxAllowance(ta);
14      }
15
16      assert(single_tax_allowance == ta);
17  }
```

As we can see by the code snippet above we tested that the *married_tax_allowance* is set correctly by calling the function *setTaxAllowance* and then distinguish the case when *ta* is higher or lower than the *single_tax_allowance* in order to make the proper check.

Even if Echidna was not able to falsify the assertion seen in *check_allowance_set* we had to add an additional check about the possible integer overflow that we mentioned before as we can see in the code below on line 6.

```
1   function setTaxAllowance(uint ta) public {
2       if (isMarried) {
3           if (ta >= single_tax_allowance) {
4               married_tax_allowance += (ta - single_tax_allowance);
5           } else {
6               require((single_tax_allowance - ta) <=
                    married_tax_allowance);
7               married_tax_allowance -= (single_tax_allowance - ta);
8           }
9       }
10
11      single_tax_allowance = ta;
12  }
```

5 Extras

5.1 Properties analysis

In order to push further the potential of this contract we analized possible extensions and then related properties that maybe have to be satisfied.

Indeed we observed that the *income* parameter is set to 0 in the constructor and then never used. So we added getter and setter functions as well as a function that computes the amount of taxes that a person has to pay.

```
function setIncome(uint new_income) public {
    income = new_income;
}

function getIncome() public view returns (uint) {
    return income;
}

function getTaxesToPay(uint percentage) public view returns (uint)
    {
    uint taxesAmount = 0;

    if (income > getTaxAllowance()) {
        uint taxableIncome = income - getTaxAllowance();
        taxesAmount = (taxableIncome / 100) * percentage;
    }

    return taxesAmount;
}
```

As we can see by the code snippet the *getTaxesToPay* function compute the amount of income that can be taxed and then use the input parameter to compute the amount of taxes that has to be paid.

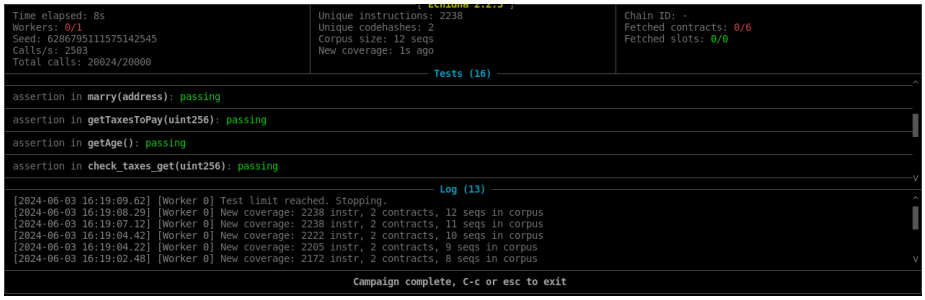
We can already observe that a possible property is that taxes doesn't not have to surpass the amount of taxable income, or it would be a nonsense.

```
function check_taxes_get(uint percentage) public view {
    if (income > getTaxAllowance()) {
        uint taxes = getTaxesToPay(percentage);
        assert(taxes <= (income - getTaxAllowance()));
    }
}
```

So we wrote an Echidna function that check this situation and as we expected it is not satisfied. This is because there is no control over the value of the percentage, that can take values over 100, which it has to be avoided.

So it was enough to add a check over the value of *percentage*.

```
function getTaxesToPay(uint percentage) public view returns (uint)
    {
    require(percentage <= 100);
    uint taxesAmount = 0;

    if (income > getTaxAllowance()) {
        uint taxableIncome = income - getTaxAllowance();
        taxesAmount = (taxableIncome / 100) * percentage;
    }

    return taxesAmount;
}
```



6 Conclusions and Future works

In this report we went over what a testing technique fuzzing is, how it is a very useful technique especially with regard to smart contract development, and how Echidna implements this approach to testing. We then took a smart contract as an example and performed a testing phase to verify that certain properties are satisfied and possibly update the code if they are not. In this way we illustrated the various ways in which Echidna can be used.

Despite this there is still room to improve the quality and security of the code. During the writing of this report it became clear the need for an access control system. This is because at present anyone can call the smart contract functions with unpredictable consequences for the system. With an access control system, on the other hand, it would be possible to define roles associated with certain operations, and only users who possess those roles would be able to

perform those certain operations. For example, a role associated with government employees could be provided for critical operations such as *marry*, *divorce*, *transferAllowance*, and all setter functions. A in depth analysis of this system and the drafting of a possible hierarchy of roles would bring the software to a higher level of reliability and security.

Beyond that, the software is still extensible in order to make it as close as possible to reality, triggering an iterative cycle of implementing new features and testing them that we are all familiar with.