Automatic Generation of Attacker Contracts in Solidity

Ignacio Ballesteros
Polytechnic University of Madrid, Spain

Clara Benac-Earle
Polytechnic University of Madrid, Spain

Luis Eduardo Bueso de Barrio
Polytechnic University of Madrid, Spain

Lars-Åke Fredlund
Polytechnic University of Madrid, Spain

Ángel Herranz
Polytechnic University of Madrid, Spain

Julio Mariño
Polytechnic University of Madrid, Spain

Abstract

Smart contracts on the Ethereum blockchain continue to suffer from well-published problems. A particular example is the well-known smart contract reentrancy vulnerability, which continues to be exploited. In this article, we present preliminary work on a method which, given a smart contract that may be vulnerable to such a reentrancy attack, proceeds to attempt to automatically derive an "attacker" contract which can be used to successfully attack the vulnerable contract. The method uses property-based testing to generate, semi-randomly, large numbers of potential attacker contracts, and then proceeds to check whether any of them is a successful attacker. The method is illustrated using a case study where an attack is derived for a vulnerable contract.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Dynamic analysis; Software and its engineering → Empirical software validation

Keywords and phrases Property-Based Testing, Smart Contracts, Reentrancy Attack

Digital Object Identifier 10.4230/OASIcs.FMBC.2022.3

Funding This work has been partly funded under the grant S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid and by the Spanish MCI/AEI under grant ref. PID2019-104735RB-C44.

1 Introduction

The support of Smart Contracts introduced a key development in the Ethereum [3] blockchain platform since the first blockchain, Bitcoin [10], was originally proposed for cryptocurrency transfers. Smart contracts provided the opportunity to study the properties and security of code executed in blockchain platforms.

In the last few years, a variety of tools and frameworks to analyze and find vulnerabilities in blockchain smart contracts have been developed based on static and dynamic analysis. These tools are based on popular program testing techniques such as fuzz testing [9, 15], symbolic execution, taint tracking, and static analysis.

A complementary testing technique is property-based testing (PBT) [4], a model-based testing technique. In PBT, tests are automatically generated from a model to cover a multitude of scenarios that a human tester may not have considered.

In [6], we presented Makina, a library and a domain specific language for writing PBT models for *stateful* programs. Models written in the new domain specific language are, using Elixir macros, rewritten into PBT state machines [1, 11]. Our main goal with Makina is to ease the task of developing correct and maintainable models, and to encourage model reuse. To meet these goals, Makina provides a declarative syntax for defining model states and model commands. In particular, Makina encourages the typing of specifications, and ensures through its rewrite rules that such type information can be used to effectively typecheck models. Moreover, to promote model reuse, the domain specific language provides constructs to permit models to be defined in terms of collections of previously defined models.

In (still unpublished) previous work, we have proposed a PBT model to test smart contracts. Such a model consists of a generic part, i.e., modelling concerns common to all smart contracts, and a specific part which is tailored to the specific behavior of each contract. As a case study, in this paper we extend the proposed PBT model and introduce automatic code generators to test a contract. The objective is obtaining a successful attacker contract on a contract vulnerable to the reentrancy attack.

A reentrancy attack involves two smart contracts: a victim contract and an untrusted attacker contract. A reentrancy attack occurs when a function from the victim contract makes a call to the attacker. In the Ethereum Virtual Machine (EVM), a reentrancy attack can happen also when a transfer to a contract is made. This transfer may end up in the execution attacker's code. The attacker takes advantage of this and tries to drain the victim's funds. One of the first (known) examples was the DAO attack which caused a loss of 60 million US dollars in June 2016.

The reentrancy attack is still an issue for Solidity smart contracts. Recent examples of reentrancy attacks are the 7.2 million dollar BurgerSwap hack (May 2021) caused by a fake token contract and a reentrancy exploit, and the 18.8 million dollar Cream Finance hack (August 2021) where the reentrancy vulnerability allowed the exploiter for a second borrow [12]. The reentrancy attack has been found in token standards as the ERC777 in the exploit of Uniswap[1]. This attack has been widely studied [2], classified[2] and is described in the official documentation of Solidity[3]. Some design patterns have been proposed to prevent this vulnerability, but the source of errors is tied to the language design.

To prevent a reentrancy attack in a Solidity smart contract one should (i) ensure all state changes happen before calling external contracts, i.e., update balances or code internally before calling external code, or (ii) use function modifiers that prevent reentrancy.

A way of testing that a contract is vulnerable to a reentrancy attack involves creating one attacker contract that exposes the problem. In this work, we use PBT to automatically generate such attacker contracts and test them against the given contract.

Additionally, many development and testing tools in the Ethereum platform offer ways of detecting contracts vulnerable to the reentrancy attack [8, 14, 16]. These tools are effective finding the vulnerability, but they are limited when providing an external test, for example in the form of an attacker contract, to independently check the vulnerability.

---

[1] https://blog.openzeppelin.com/exploiting-uniswap-from-reentrancy-to-actual-profit/
[2] https://swcregistry.io/docs/SWC-107
[3] https://docs.soliditylang.org/en/v0.8.11/security-considerations.html#re-entrancy

The rest of the paper is organized as follows. In Section 2 we present a smart contract which is vulnerable to the reentrancy attack and that will be used as a case study. The ideas behind the generation of attacker contracts are outlined in Section 3. In Section 4 we explain in detail how property based testing works and how it is used to test stateful systems. Experimental results are presented in Section 5. Some related work is discussed in Section 6. Finally, conclusions, limitations and further work are discussed in Section 7.

2 Case study

In this section, we present a running example based on the well known reentrancy attack. The goal is to test whether a contract is vulnerable to such an attack by creating an attacker contract, that tries the attack on the given contract. As we shall see, the attacker contract can be generalized and, thus, reused for testing other contracts against the reentrancy attack.

2.1 The victim contract

The following contract represents a wallet vulnerable to reentrancy attacks. In this wallet multiple accounts can deposit and withdraw *ether*. The funds are private and one can only operate on its own balance.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.15;
3
4  contract Wallet {
5    mapping(address => uint) private credit;
6
7    function donate() external payable {
8      require(msg.value > 0 wei);
9      credit[msg.sender] += msg.value;
10   }
11
12   function withdraw() external {
13     uint bal = credit[msg.sender];
14     require(bal > 0);
15
16     (bool sent, ) =
17       msg.sender.call{value: bal}("");
18     require(sent, "Failed to send Ether");
19
20     credit[msg.sender] = 0;
21   }
22 }
```

Listing 1 Wallet contract (victim)

The `Wallet` contract has a `private` attribute named `credit`, which is a table storing the balance for each account. The contract exposes two public methods: `donate` and `withdraw`, and they can be called by any account or any other contract.

The method `donate` is labeled as `payable`, indicating that when this method is called, it can be done with some *ether* value attached. The *ether* value can be seen in the variable `msg.value`. In this case, the `Wallet` contract requires it to be greater than 0. The variable `msg.sender` has the address of who (account or contract) called to this method. The `Wallet` contract proceeds to register and link the `ether` amount and the sender into the `credit` variable.

When the `withdraw` method is called, the `Wallet` contract transfers back the registered balance of the caller (`msg.sender`). The transfer is done using the `call` function with the corresponding *ether* attached. Finally, it updates the `msg.sender` balance to 0.

The reentrancy attack is based on the exploit of a particular behavior of EVM smart contracts, affecting to languages like Solidity. When a transfer is done to a contract, the receiver can execute some code during the transaction. In the case of a transfer between two contracts, this behavior is giving the control to the receiver during a function call.

Here, the goal of the reentrancy attack is to steal money from a victim contract by draining its funds. In the `Wallet` contract the reentrancy attack can be exploited because in the `withdraw` method, the `sender` balance is updated after the amount has been transferred (lines 20 and 17, respectively). During this transfer, the attacker calls again to `withdraw`. The balance has not been updated yet (line 20), so the conditions to make a new transfer are still met (lines 13 and 14). This loop of re-entrant calls could be executed until the `Wallet` transfers all of its funds.

In the following section, we explain the attacker contract and how it is able to exploit the attack on this `Wallet` contract.

2.2 The attacker contract

Solidity smart contracts can define a function to be executed when they receive a transfer. This function is named `receive`, and it is fundamental for the reentrancy attack. In this section we explain a contract exploiting the reentrancy attack in the `Wallet` contract of the previous section.

In the following Solidity code, a contract attacker is presented. The entry point is the `attack` function. This function makes a first payment to the victim contract and then calls the `withdraw` function which will transfer ether back to the attacker. This transfer will trigger the `receive` function and before the `receive` terminates, the attacker calls again the `withdraw` function. That is, the attacker contract consists of two phases interacting with the victim contract, first the trigger phase, and second, the `receive` phase to continue draining the victims funds.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.4.22 <0.9.0;
3
4  contract Attacker {
5    address private victim;
6
7    constructor(address victim_) {
8      victim = victim_;
9    }
10
11   receive() external payable {
12     uint victimBalance =
13       address(victim).balance;
14     bool has_funds =
15       victimBalance >= msg.value;
16
17     if (has_funds) {
18       try victim.withdraw() {} catch {}
19     }
20   }
21
22   function attack() external payable {
```

```
23      require(msg.value >= 1 wei);
24      uint init = address(this).balance;
25
26      victim.donate{value: msg.value}();
27      victim.withdraw();
28
29      uint end = address(this).balance;
30      emit SuccessfullAttack(end > init);
31    }
32  }
```

Listing 2 Attacker contract

The `attack` function emits an event `SuccessfulAttack(true)` with the result of the attack. We will use this event to test whether the attack was successful or not.

3 Automatic generation of attacker contracts

The goal is to automate the generation of a successful attacker contract. Our approach uses Property-based testing (PBT) to automatically generate random calls to the public functions obtained from the Application Binary Interface (ABI) of the victim contract with the goal of detecting a reentrancy vulnerability. If a vulnerability is found, the result is an attacker contract that can exploit the aforesaid vulnerability.

As we have seen in the previous section, in the case of the reentrancy attack, there are two phases in the interactions between the attacker and the victim: (i) the `trigger` phase that prepares the attack, and (ii) the `receive` phase where the attack takes place. These two phases correspond to two functions in the generated attacker contract: the `trigger` function and the `receive` function.

The trigger function contains a sequence of calls to the victim contract which ends with a call that *triggers* a transfer from the victim to the attacker. Then, this transfer invokes the `receive` function. In the `receive` function, there is a sequence of at least one call to reenter into the victim contract. Our approach is to automatically find such sequences, if they exist for the victim contract, using PBT.

The following code is the template of the attacker contract where the sequence of calls of the trigger and receive functions are generated using PBT. In the next section, we explain how they are automatically generated.

```
1  function trigger_sequence()
2    { /* To be generated using PBT */ }
3  function receive_sequence()
4    { /* To be generated using PBT */ }
5
6  function attack() external payable {
7    ...
8    trigger_sequence();
9    ...
10 }
11
12 receive() {
13   ...
14   if (has_funds) {
15     try receive_sequence() {} catch {}
16   }
17 }
```

Listing 3 Attacker contract template

For clarity, only the relevant code is displayed. In this code, line 8 corresponds to lines 26 and 27 of the attacker contract shown in Section 2.2. Note that this template contains the code that is common to many attacker contracts for the reentrancy attack, while the generated code is specific to each victim contract.

4 Property-based testing

Property-based testing (PBT) is a testing methodology which focuses on generating, automatically, test cases from a formal description of the behavior of the system under test. That is, PBT can be considered a form of model-based testing.

Formally, the model is an extended finite-state machine where one has to define the *state*, *commands*, *preconditions* and *postconditions*. PBT tools, for example, [1], will generate execution scenarios that will be run on the model and on the system under test to prove that the actual system behaves like the model. To generate the execution scenarios, *generators* are used.

4.1 Generators

A generator is capable of generating an infinite number of values of some data type, according to a probability distribution. PBT tools come equipped with a library of standard generators, for example, a generator for integer numbers.

In this work, we have defined a generator for non-negative integers to represent the ether used in payments. This generator is used as an argument for all transactions, for example in the calls to the *payable* functions or when deploying a contract.

A user of a PBT tool can also implement custom generators. In this work, we have defined a generator for attacker contracts.

4.1.1 Attacker contract generator

The attacker contract generator uses the template described in Section 3 and the ABI of the victim contract to generate a Solidity contract. The ABI is generated when a contract is compiled and contains the specification of the contract's functions. From the specification, one can extract the name, arguments and modifiers.

The attacker contract generator uses generators of values and function calls. These generators are fed with the information extracted from the ABI to obtain targeted function calls with valid values.

To generate an attacker contract, two sequence of calls are generated and inserted in the template: the trigger and the receive sequence. These sequences contain calls to the victim contract. A call is generated from the ABI, based on its specification. When a call to a contract needs to be generated, it chooses one of the defined function in the ABI. If the function has any arguments or modifiers, for example: if it is `payable`, an *ether* amount is generated for the call. Figure 1 is a diagram of this described process.

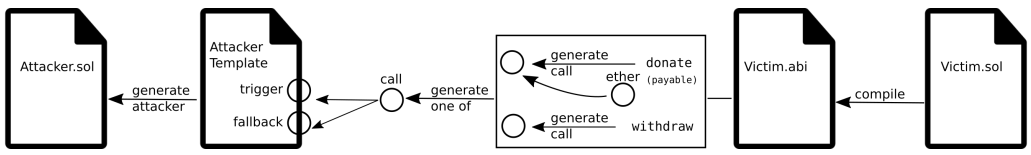

Figure 1 Attacker contract generator.

The following represents an example of generated code in the `Attacker` for the trigger and receive sequences in the `Wallet` example.

```
function trigger_sequence() payable {
  victim.donate{value: 3}();
  victim.withdraw();
  victim.donate{value: 1}();
}

function receive_sequence() payable {
  victim.withdraw();
  victim.donate{value: 1}();
  victim.withdraw();
  victim.donate{value: 2}();
  victim.withdraw();
  victim.withdraw();
}
```

Listing 4 Generated attacker contract sequences.

In this example, the generator has filled the two sequences: the `trigger_sequence` and the `receive_sequence`. This code fragment belongs to a generated contract based on the template in Listing 3. For example, the `trigger_sequence` includes 3 generated function calls: 2 `donate` calls and 1 `withdraw`. The function `donate` is `payable`, which means that it can receive `ether` attached, and the calls to this function include a generated `ether` value as argument.

4.1.2 Function call generator

The generation of function calls is driven by the specification found in the ABI. In the ABI, a function definition contains the name of the function, the number of arguments and their types and any other modifier. In the example of the `donate` function, Listing 5 represents the object definition extracted from the ABI.

```
{
  function: "donate",
  method_id: "ed88c68e",
  input_names: [],
  types: [],
  returns: [],
  type: :function,
  state_mutability: :payable,
  inputs_indexed: []
}
```

Listing 5 ABI definition of function `donate`.

Given the ABI definition of a function, the contract call generator outputs a new object with generated values. As the `donate` call is `payable`, an *ether* value is generated. Listing 6 contains the generated object for a single function call. Then, this generated call is encoded as Solidity code: `donate{value: 3}();`

```
{
  name: "donate",
  args: [],
  state_mutability: :payable,
  value: 3
}
```

Listing 6 ABI definition of function `donate`.

The contract call generator is able to generate values for each given argument of a function. The `donate` function does not have any additional argument, but given list of types for each argument, it generates a list of values for each corresponding argument. This generator is explained in the next subsection.

4.1.3 Value generator with context awareness

A valid Solidity value can be generated given a type. Given an `int`, a number in the range of the integers is generated. For `uint`, only non-negative integers are possible. This mapping between types and possible values is done for each Solidity type except `function`, `fixedNxM` and `ufixedNxM`.

There are limitations with random generated values. A relevant case is `address` values. A generated reference for an address will be syntactically valid, but it is unlikely to represent a populated address in the blockchain. This behavior is desired to test some cases, but the generation will be biased towards testing non-existent addresses. To fix this limitation, one could include in the address generator the possibility of generating addresses from a set of already known values.

A context in the generation of values carries extra information about values to be generated, for example: set of possible addresses, limits on the amount of *ether*.

4.2 Commands

In most cases, a test case is not a simple call to a function, but rather a sequence of calls to operations or methods interacting with a system. In our case, the blockchain and the smart contracts it contains. That is, a test case generator returns sequences of API calls, and the test property checks whether the execution of such a sequence of API calls is correct. To interact with the Ethereum blockchain we have defined commands that will be executed during the tests, checking the expected result after each call. In the following subsections, we describe the specific commands interacting with the blockchain that are relevant to understand how the test for reentrancy attack is executed.

4.2.1 Register Attacker

The Register command is responsible for compiling the generated attacker. The input of this command is the source code of an attacker contract, generated by the attacker contract generator, described in Section 4.1.1 using the known victim's ABI. The command compiles the generated attacker source code and registers the contract. This registration is required to later deploy instances of the attacker with the *Deploy* command. After the execution of this command we will have a registered attacker.

4.2.2 Deploy

This command will deploy a given contract with the arguments of the contract's constructor. We use this command with the victim contract and with the attacker. Once the contract is registered with the contract manager library, the command deploy can be executed. The result of this command is a transaction hash returned by the blockchain, ensuring that the request was received but might not be processed yet. The address where the contract was deployed is known once the transaction is completed.

The `constructor` of the attacker contract requires the address of the victim as an argument (see line 7 of Listing 2). If the victim contract has already been deployed, the tool is able to select the address as the argument for the `constructor`.

4.2.3 Attack

The Attack command triggers the execution of the attack. It calls to the method `attack` in the attacker contract. The condition to be able to execute this command is having an attacker deployed and its address known. We expect a transaction hash returned, to later check the transaction completion status. As the `attack` function is `payable`, this command has the possibility to call the method with an *ether* amount. The PBT tool generates this value with an *ether* value generator. The `attack` function in the smart contract emits an event with the result of the test. This Attack command also specifies that we expect that event with a result of *"not succeeding"*. This event is checked later with the *Get Events* commands.

4.2.4 Get Events

Given a transaction hash and if the transaction is complete, this command fetches and decodes any known events emitted. We know what events we expect from the transaction because any transaction command can add the expected result. For example, the *Attack* command included the result event as the one expected. At this point, we can have all the information to test the success of the attack

4.3 Shrinking

As a final ingredient in PBT implementations, there is an attempt to derive an easy-to-understand counter example through a procedure called *shrinking* which systematically tries to simplify counter examples, in order to ease the (manual) analysis required to discover the cause of the detected error. Shrinking is linked to generators. For instance, shrinking uses 0 as a simpler value for the int generator.

In our approach, the attacker contract is automatically obtained from shrinking, as it is explained in the following section.

5 Results

To test the Wallet contract presented in Section 2, we have used *Makina* [5], an Elixir DSL for writing stateful PBT models compatible with Quviq's Erlang QuickCheck [1] and PropEr/PropCheck [11]. To replicate the blockchain ecosystem, we use `Ganache`[4] as a local development and testing environment.

[4] `https://trufflesuite.com/ganache/`

The objective of the test is finding a successful attacker contract. To do this, the test deploys a private blockchain environment in which the commands are executed. The test fails when the expected result after the execution of the command is different from the behavior of the blockchain system. That is, we are comparing the results of executing commands in the model of the blockchain with those in the real system. Concretely, if the `attack` function successfully emits an event `SuccessfulAttack(true)`, then the test has found an attacker exploiting the reentrancy vulnerability in the victim contract.

In the case of the Wallet example, when the test is run, we obtain the following output.

```
Failed!

expected: SuccessfulAttack(false)
obtained: SuccessfulAttack(true)
```

That is, a test has found a successful attack event after the execution of an attack. The result of the test includes a counterexample, which can be used to reproduce the test case to manually check the error and study the source of the vulnerability. The provided counterexample includes: (i) an attacker contract that is capable of exploiting the reentrancy attack vulnerability, and (ii) a sequence of interactions with the blockchain to make the attack possible.

PBT tools try to shrink the counterexample to produce a simpler and smaller attacker contract, and to reduce the number of interactions with the blockchain to reproduce the attack. Note that the shrinking process does not guarantee that the reduced counterexample is the minimal case. Therefore, different runs of the test usually lead to different counterexamples.

In the following, we present a reduced counterexample of the generated attacker contract sequences i.e., after shrinking. The original counterexample had 22 calls to the victim contract in the trigger sequence, 10 to `withdraw` and 12 to `donate`, and 32 calls in the attack sequence, 13 to `withdraw` and 19 to `donate`.

```
1 function trigger_seq() public payable {
2   target.donate{value: 1}();
3   target.withdraw();
4 }
5
6 function receive_seq() public payable {
7   target.withdraw();
8 }
```

Listing 7 Generated and reduced attacker contract sequences.

The reduced counterexample contains the same calls that were manually written in the first Attacker contract shown as an attacker example (Listing 2).

The second part of the counterexample is the sequence of commands interacting with the blockchain. Initially, the sequence of commands had 113 steps. This does not necessary mean that all those commands are executed, on the contrary, it means that an error was found at some point while executing those commands. The shrinking is applied here to find a smaller counterexample. The PBT tool manages to shrink from 113 commands to a sequence of 15 commands. From this, the relevant commands that are required to prepare the blockchain into a state where the attack can be made are the following:

```
victim_address = deploy(Victim, value: 0)
register_attacker(Attacker, source_code)
deploy(Attacker, victim_address, value: 0)

call(victim, :donate, value: 1)
attack_hash = call(attacker, :attack, value: 2)
get_events(attack_hash)
```

Listing 8 Generated and reduced commands sequences.

The first three commands deploy the contracts required for the test. The `value` attribute represents the amount of initial *ether* that is transfer to the contract. The register command compiles the source code generated with the attacker contract generator and registers into the tool with the name `Attacker`. The deployment of the attacker command uses the address of the victim (`var_victim`) as argument to the constructor of the `Attacker`.

The next three lines contain calls generated interacting with the contracts. Notice how there is a command calling the method `donate` of the victim contract (`var_victim`) with an amount of `value` of 1 *ether*. This donation is fundamental to perform the attack, as the attacker contract tries to steal preexisting funds. These funds come from a known account that we have omitted for simplicity.

The time spent to obtain this result was 2 minutes and 30 seconds until finding a successful attacker contract. Then, it took 1 minute to shrink the counterexample. More than a hundred tests where generated, each one of them with a fresh Ganache environment. The tests ran on a laptop with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz and 16GB of RAM.

6 Related Work

There has been a great interest on detection and prevention of reentrancy attacks after the DAO incident. The DAO attack itself and reentrancy vulnerabilities are documented in [2], which presents a whole catalog of vulnerabilities for Ethereum contracts. A survey of tools for analyzing Ethereum smart contracts is presented in [7]. Specifically for reentrancy vulnerabilities, [13] proposes to take advantage of the similar code structures found in these attacks. Structural analysis is performed in order to detect suspicious code patterns. After that, a dynamic analysis is performed in order to rule out false positives.

Several testing tools and frameworks for smart contracts have also been proposed, some of them combining techniques such as *fuzzing* or *genetic algorithms*. Dapp tools[5] is a kit including `hevm`[6], a fuzzing tool which also uses symbolic execution to falsify assertions in contracts. Foundry is an alternative collection of development tools, providing a testing framework, `forge`[7], with fuzzing capabilities. Some testing tools worth mentioning are CONTRACTFUZZER [8] and ECHIDNA [14]. Each of these tools require different information from the users – models, invariants, etc. – and also provide diverse outputs – some of them produce complete test suites, other just diagnostics on existing suites, etc. Our approach is original in the sense that is mostly automatic and provides a reusable, somehow minimal attacker contract.

Our work shares with CONTRACTFUZZER [8] a similar approach reading the Application Binary Inteface (ABI) to extract all the functions calls to a contract. This is a key component for the automation of the tools. The CONTRACTFUZZER tool has an `Agent` contract, which

[5] `https://dapp.tools/`
[6] `https://github.com/dapphub/dapptools/tree/master/src/hevm`
[7] `https://github.com/foundry-rs/foundry/tree/master/forge`

plays a similar role as our template `Attacker` contract when checking for reentrancy attacks. The existence of this contract is required by the nature of the attack, because it depends on the interaction of two contracts. The `Agent` auxiliary contract is designed as a link to execute calls to any contract. Our template is oriented to code generation, with the objective of building a self-contained contract. While CONTRACTFUZZER is oriented to be an analysis tool, we present a testing approach to be integrated with the development of smart contracts. We aim to provide useful and reduced counterexamples to understand the origin of the error and how to reproduce it, not only the diagnosis of the error.

We pursue similar objective as the ECHIDNA [14] tool. We both want to provide a tool where a minimal interaction with the user is possible. This requires analysis of the ABI for the automatic generation of calls. ECHIDNA focuses on detecting assertion violations and custom properties written in Solidity. To test them, ECHIDNA generates a sequence of calls to a contract to reach those conditions, shrinking them when an error is found. The main difference with our tool is that we can automatically test interactions between contracts.

Our proposal fits in the space between some functionalities of CONTRACTFUZZER and ECHIDNA, where we are able to automatically generate calls to a given contract but also provide meaningful counterexamples. Another advantage of our approach is that PBT is a general testing technique, that is, PBT testers do not need to learn a specific tool for testing smart contracts.

7 Conclusions and future work

In this paper, we present an approach to automatically detect vulnerabilities in Solidity smart contracts. Given a potential victim, the result of our approach is an attacker contract that exposes a vulnerability. We illustrate the approach in the case of the well known reentrancy vulnerability but our method that can be applied to other vulnerabilities in smart contracts.

Our method consists of identifying the common structure of an attacker. This structure is prepared as a template for automatically generating attackers given a victim contract. This generation is made using PBT tools, in which we have defined a custom generator for contracts of this type of attack. Using a model of a blockhain, we check the execution correctness by comparing it with the behavior of a running blockchain system.

The method proposed is able to identify the vulnerability in a given vulnerable contract. We provide as a result an instance of an attacker contract, along with all the steps required to replicate the attack.

The scope of this paper is limited to the identification of the reentrancy vulnerability in the `Wallet` example. The future work will include doing a study on the effectiveness of this method with a set of real world contracts. With this study, we will be able to compare the success of this method with similar tools detecting the reentrancy attack vulnerability.

The method explained it is not only aimed to identify the reentrancy attack vulnerability. In future work, we plan to apply this method on a broader range of attacks and study the effectiveness to detect other properties of smart contracts.

In this work, we have illustrated the approach with a simple example of a contract, the `Wallet` example. The public interface of the `Wallet` contract is fairly simple, and we did not have to elaborate complex interactions in the generated code. We plan to extend the context aware generator to, for example, use the return value from generated calls as arguments in the following calls. By doing that, we will be able to study the capability of our approach to generate more complex contracts and attacks.

The reentrancy attack is present in a wide variety of contracts. Token standards as the ERC777, mentioned in Section 1, can be susceptible to reentrancy attacks. As future work, the presented method could be studied to be adapted to this common interface contracts. The definition of specific PBT commands following the properties of the standard could be checked with generated contracts trying to break these conditions. The reentrancy condition arises as another property to be checked for these contracts. The general effectiveness of the vulnerability identification is future work with a corpus of contracts to test.

We have presented a testing approach that tests a pair of contracts: a victim and an attacker. Some interesting behaviors and bugs in our victim contracts may arise when multiple contracts operates and interact between them. In the future work, we plan to test models of more than two contracts interacting.

An additional detail is the refinement of the success condition in a reentrancy attack. Using the condition of a detected repeated call hook in the attacker contract along with detecting profits in an exploit.

References

1 Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with Quviq QuickCheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. doi:10.1145/1159789.1159792.

2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust – Volume 10204*, pages 164–186, Berlin, Heidelberg, 2017. Springer-Verlag. doi:10.1007/978-3-662-54455-6_8.

3 Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. *white paper*, 2013. URL: http://ethereum.org/ethereum.html.

4 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. doi:10.1145/351240.351266.

5 Luis Eduardo Bueso de Barrio, Lars-Ake Fredlund, Ángel Herranz, Clara Benac Earle, and Julio Mariño. Makina: A new quickcheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, Erlang 2021, pages 41–53, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471871.3472964.

6 Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, Clara Benac Earle, and Julio Mariño. Makina: a new quickcheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, pages 41–53, 2021.

7 Monika di Angelo and Gernot Salzer. A survey of tools for analyzing Ethereum smart contracts. *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019.

8 Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, September 2018. doi:10.1145/3238147.3238177.

9 Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990. doi:10.1145/96267.96279.

10 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *white paper*, May 2009. URL: https://bitcoin.org/bitcoin.pdf.

11 Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2034654.2034663.

12 Kamil Polak. https://hackernoon.com/hack-solidity-reentrancy-attack, January 2022.

13 Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in Ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, February 2020. doi:10.1109/iwbose50093.2020.9050260.

14 J. Smith. Echidna, a smart fuzzer for Ethereum, 2018.

15 Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 2nd edition, 2018.

16 Valentin Wüstholz and Maria Christakis. *Harvey: A Greybox Fuzzer for Smart Contracts*, pages 1398–1409. Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3368089.3417064.