Runtime Verification of Smart Contracts

On the Ethereum network

Lars Stegeman [s1346466]
l.stegeman@student.utwente.nl

May 28, 2018

Contents

1 Introduction

Each day new smart contracts are deployed to the Ethereum network. Some of these smart contracts control a large sum of ether. Since this ether has real world value and the source code for smart contracts is in the open many people are finding vulnerabilities within contracts. Several high profile security bugs were found and exploited [1, 2, 3, 4]. This sparked the interest in static

analysis tools and formal verification of smart contracts. Static analysis tools can be executed on many contracts and report valuable information on common mistakes. However unknown vulnerable patterns will not be detected since they have to be explicitly programmed to be able to analyse the specific pattern. Other tools which use formal verification need a specification to be able to guarantee a contract behave the correct way. These specifications are usually written in another language or defined at the EVM level. This makes it hard to understand what properties are proven and what that means for the contract. Smart contract developers will have a hard time understanding all these separate tools and their diverse syntax. This paper will describe a proposal for a concept tool which makes it possible to write specifications at the same level of Solidity code. These specifications will then be parsed and can be used for a number of different analyses. The benefits of this approach are:

- Explicitly writing a specification helps understanding the problem. The code usually describes how a contract should behave and do calculations. While the specification should describe what the contract does and what properties should be satisfied.
- Runtime exceptional state. While the contract is active on the main Ethereum network properties can be checked at runtime. If a certain property fails due to an untested case, the program can go into an exceptional state. In this state, functions can be deactivated or the contract can be completely cleared. Some special form of governance can be coded in this state which requires human intervention before the contract will continue.
- It helps with bug bounty programs. When new contracts are developed they usually first launch a bug bounty program. Vulnerabilities that discovered by users can be rewarded by the creators of the contracts before it goes live on the main network. In most of these programs the specification of what the contract should do is not given. This makes it difficult do decide what is intended behaviour and what counts as a found vulnerability. With runtime verification one could specify properties that the contract should satisfy. If one of these properties fail because of a vulnerability found by the community, it is guaranteed not to be intended. This makes these bug bounty programs more useful for both the creator and the participants.
- The output of the tool is Solidity code which means that it can serve as input to other formal verification tools. For example the KEVM framework [5], which formally verifies smart contracts at the EVM bytecode level.

The rest of this paper will be structured as follows. First the background information is given about the Ethereum blockchain and underlying principles. Then smart contract are introduced and explained at a high level overview. Next Solidity, the most used contract programming language, is discussed. Section 4 will have an example for a possible use of the concept tool. Then runtime verification is introduced and other related work is explained. The last section contains the research questions and a planning for the development of the tool.

2 Background

This section will discuss the background information that will be built upon further in the document. First we will briefly discuss the important parts of the Ethereum Blockchain. Then we will discuss the smart contracts in more detail.

2.1 The Ethereum blockchain

The Ethereum platform is built upon a distributed public ledger. On this ledger the cryptocurrency ether is stored. Ethereum has different denominations of the unit ether. The smallest value or base value is called `wei`, a single ether represents 1e18 wei. In contrast to Bitcoin, it is an account based system and not based on unspent transaction outputs (UTXO). There are two types of accounts, one is a default account in which a user controls the spending of funds through its private keys. These accounts are called "Externally owned Accounts". An account can be referenced by its address which is a hashed version of the public key. Each address has a balance and a nonce. The nonce is incremented each time the balance is updated with a transaction. The other option is a "Contract Account", which means that it is managed by code only. A contract account has more

data stored on the blockchain. These include storage hash and a code field. The code is set when the contract is constructed and initialized on the blockchain, and after that can never be changed. The code that is included in contracts is called Ethereum Byte Code. This bytecode is executed in a VM called the Ethereum Virtual Machine (EVM). Each contract has a persistent storage which is also maintained on the blockchain. Contract accounts only execute code when they are called from other contracts.
Transactions are created and sent to the network by creating a message and signing it with the private key of an "Externally Owned Contract". This contains information like the amount of ether and the receiver of the transaction. Additionally it can contain so called call data. This data is interpreted by the contract code and the correct function is executed. Transactions are the only entity that make changes to the storage. At an higher level overview we could see the Ethereum network as a large state machine in which changes to the state are controlled by transactions. Transactions are grouped in blocks and these blocks are distributed over the network and validated by each node.
The different types of state and environments are also described more formally in the Ethereum Yellow Paper [6]. The Yellow Paper states that there are three separate storages in each context.

- World state (σ): A mapping of Ethereum addresses to the accounts. Within each account the balance, contract storage, contract code and nonce are stored. For "Externally Owned Account" the contract code and storage are empty.
- Machine state (μ): State of the currently executing code from a transaction. This includes program counter, contract memory and virtual machine.
- Execution Environment (I): Variables related to this transaction. For example caller address, amount of ether send and call data.

Transactions can only be initiated from accounts. This means that the blockchain is global state computer which changes each time a transaction is executed. Transactions can be seen as function calls with additional information. This information includes the transaction sender, gas price and amount of ether.
Blocks serve the purpose to group transactions and give them order. Because the ordering is very important to the outcome of the transactions. The ordering is determined within a block and should be deterministic and all nodes should agree on the global state. This securing of blocks is done using a proof of work mechanism that is used by most cryptocurrencies. However each miner also has to validate each transaction by executing the corresponding EVM code and adjusting the global state. This is also done by each individual node to validate the block which includes all the transactions.

2.2 Smart Contracts

Smart contracts are usually mentioned together with Ethereum. Other terms for smart contracts are "autonomous agents" or "executable code on the blockchain". It has many application domains according to the Ethereum White Paper [7]. Examples of usage cases include token systems, decentralized autonomous organizations (DAO), financial derivatives, identity/reputation systems and decentralized file storage. The idea is that these domains are perfect for the blockchain since they take away the untrusted third party. Smart contracts can only operate on data within the blockchain, this means that all information has to be included in the transactions that are send from "externally owned accounts". However in this paper we will look at the functional capabilities of smart contracts on the Ethereum network.
Smart Contracts on the Ethereum network consist of two parts. Each contract has a set of functions and a storage. The contract set of functions is defined by the contract code that is deployed with the contract creation. This contract code is EVM bytecode and is usually compiled from a higher level programming language. When the contract is created the storage is initially empty. Only the contract code can make changes and add data to the persistent storage, within this storage the state of the contract is maintained. As explained before each transaction also has a state. This is called `memory`, and is initially empty. It can also be used to store data and is much cheaper in terms of gas cost. But this data is not persistent through transactions, it is only persistent within the transaction. There are also so called "logs", this storage can only be used to store data and not retrieve. This storage is usually used to provide data for the external world because it can be

searched efficiently.
Since the EVM is a turing complete language any program can be expressed within the platform. To mitigate the possibility of a Denial-of-Service attack (with for example an infinite loop) the principle of gas is introduced in Ethereum. Gas is used to limit the amount of complex code that can be executed within a single transaction. Each transaction only has a specified limit called the gas limit. If an execution is terminated unexpectedly or runs out of gas the complete transaction is reverted. This includes storage changes made before the exception. When a transaction is successful left over gas will be returned to the sender. In the case of an exception all the remaining gas is consumed. Functions are only executed when they are called by external contracts. For example if a fund is to be released after a certain amount of time (block number higher then a certain amount). These funds will not be automatically transferred once the time threshold is reached, they will only be released when the function is called again.

2.3 Smart Contract bugs

Many smart contracts are deployed to the Ethereum main network every day. When a contract is created on the blockchain the contract code is stored on the blockchain forever. This cannot be changed afterwards. Because of this limitation bugs within smart contracts can be very costly. In the past many vulnerabilities have been detected causing a loss of several million Ether. This paper will not enumerate all of them since many other articles do a good job of summarizing all the found vulnerabilities. For a complete overview see [8] section 3, where each attack with its corresponding vulnerability is explained in detail.

3 Solidity

The most used language to develop contracts on Ethereum is Solidity [9]. Solidity comes with a compiler that compiles Solidity code into EVM bytecode. This bytecode is what is executed and put on the blockchain. Solidity has features like control flow, types and different storage constructions. Additionally it has some global variables that apply only to the blockchain setting. In this section we will further introduce the language in detail.

3.1 Syntax

The syntax that is used by Solidity is heavily inspired by Javascript. Solidity is in contrast to Javascript strongly typed. It offers the common types in normal programming languages like; booleans, integers, strings, fixed point numbers. Since each contract is executed on the blockchain, storage is extremely costly in terms of gas cost. This is why many different sizes for integers exist: `uint8`, `int8`, `uint16`, until `uint256` and `int256`.
Solidity offers a number of different options for more complex types. These complex types have an extra annotation that defines their storage location. This can either be `storage` or `memory`.

- Structs are a form to create new types in Solidity. Structs can contain any type including mappings except itself. For example a struct type A cannot contain a member of type A (no recursive definition).
- Arrays can be defined in memory or storage. Storage arrays can hold arbitrary types, memory arrays can not contain mappings. Storage arrays can be dynamically increased in size, however memory arrays are always fixed length.
- Mappings can only be defined in storage. They map a key of a certain type to a value of another type. They can be compared to hash tables in normal programming languages. However the key set of a mapping is not stored, this makes mappings not iterable.

The code snippet below shows how all these constructions can be used within a contract.

```
pragma solidity ^0.4.23;

contract C {
    // State variables are always stored in storage
    uint256 public number;
```

```
    uint[] x;
    mapping(address => uint256) myMap;
    // Definition of type myStruct
    struct myStruct{
        uint256 a;
        address b;
    }

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory

        // Declaring a mapping in memory is not allowed
        // mapping(address => uint256) memory temp_map;

        myStruct memory a; // declares a variable of type struct in memory
        myStruct b; // default of complex types is storage
        b.a = 100; // will assign 100 to the variable number!
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {}
}
```

3.2 Structure

In Solidity contracts are treated like objects in Object Oriented Programming languages. Contracts can contain state variables and functions and inheritance is supported between multiple contracts. A contract can have a constructor which will be called upon creation of the contract on the blockchain. In the code example below a simple contract is shown with the basic structure.

```
pragma solidity ^0.4.23;

contract SimpleStorage {
    uint public storedData; // State variable

    //Constructor will be called upon creation on blockchain.
    constructor(uint data){
        storedData = data;
    }

    function setData(uint data) public{
        storedData = data;
    }
    function() payable{
        //Unnamed function will be called if no function signature matches
    }
}
```

Solidity also has different visibility keywords. Their behaviour is a bit different from normal programming languages since it is executed on a blockchain setting. Visibility can be defined for functions and variables.

- `external`: External can only be used by functions and means that they can not be called from internal functions. They can be called from other contracts.
- `public`: Public can be used for functions and state variables. For functions it means that it can be called both internal and external. For state variables it means that a getter function is automatically generated.

- `internal`: Internal functions and state variables can only be accessed internally from within the current contract and derived contracts.
- `private`: Private functions and state variables are only visible to the contract they are defined in.

The extra keywords are used because different functionality can be desired by contracts. Also note that private variables can be read outside of the EVM by inspecting the storage of the smart contract [1]
Solidity also gives the possibility to define function modifiers. These are usually used to check a condition before execution of a function. Modifiers can be inherited from other contracts and reused in functions on that contract. As explained in the previous section the Ethereum blockchain has another type of storage called "logs". Logs are read only and can be written to using `Events`. Events have to be defined in the contract itself and can be inherited, events can have specified parameters to emit the correct information. Below is a Solidity code snippet showing the basic behaviour of both constructions.

```
pragma solidity ^0.4.23;

contract myContract {
    uint public data;

    //Event declaration
    event dataIncreased(address sender, uint amount);

    //Modifier declaration
    modifier onlyPositive(uint number){
        require(number >  0);
        _;
    }

    //Before function call check modifier onlyPositive
    function increment(uint number) onlyPositive(number) public{
        data +=number;
        //Emit event dataIncreased
        emit dataIncreased(msg.sender,number);
    }
}
```

The function `increment` has a modifier that will be executed when the function is called. The modifier `onlyPositive` checks the number and requires the number to be greater then zero. The "_;" indicates the rest of the body of the function. This way function modifiers can be used to add code before and after the normal function body. If the assumption fails the require will throw an exception and the transaction will stop executing. This means that all state changes made during the transactions are reverted and the transaction is marked as failed. There are two types of constructions that can be used to detect undesired behaviour one is `require()` the other is `assert()`. Both function will throw an exception when the statement is false, but assert will consume all remaining gas while require will not consume any more gas. This means that in practice `require` is used to check and validate user input, and `assert` is used to test invariants and internal error checking. Both functions will create an exception that will bubble up to through the call structure. At this point exceptions can not be caught.

3.3 Blockchain specific variables

What makes Solidity special in terms of programming languages is that it is executed on the blockchain. All code is executed because of the transactions that are being sent to the network. These transactions can be seen as rich function calls with extra information. This extra information is available in special constructed variables which are globally accessible during execution of the contract.
There are two objects that contain information about the blockchain these are: `block` and `msg`. The block object contains variables like `block.number`, `block.timestamp`, `block.difficulty` and `block.coinbase` (current block miner address). The information in block is the block where the current transaction is mined in. The object msg contains information about the current transaction.

[1] For example with the web3.js interface with the call `web3.eth.getStorageAt(addressHexString, position)`

These are found in variables like: `msg.gas` (remaining gas), `msg.value` (value sent in wei) and `msg.sender` (address of the sender). The `address` object is used for communication between contracts. This makes it possible to execute code of multiple contracts within a single transaction. The keyword `this` refers to the address object of the current contract. This also contains the balance of the contract under the variable `<address>.balance`. There are five different flavors of calling other contracts.

- `<address>.transfer(uint256 amount)`: forwards given amount in wei to address, throws on failure. The function sends 2300 gas with the transfer.
- `<address>.send(uint256 amount) returns (bool)`: same behaviour as transfer but returns false on failure.
- `<address>.call(...) returns (bool)`: forwards all gas to function call. Returns false on failure.
- `<address>.delegatecall(...) returns (bool)`: same behaviour as call but storage and state variables of original contract are used. This makes it possible to create library functionality within the blockchain. The library contract can contain functions that do not require access to state variables. That means that they must rely on their input. Or the library contract has to have to exactly the same state variables declared in order to be used in functions of the library contract.
- `<address>.callcode(...) returns (bool)`: older version of delegatecall. Usage is discouraged and will be removed in the future.

All these transfer functions can be sent to "Externally Owned Contracts", but also on "Contract Accounts". This means that arbitrary code can be executed when invoking one of these methods. To limit the amount of code that can be executed by a remote function call it is important to specify the amount of gas to be sent with the transfer. Exceptions can not be caught within contracts, they bubble up through the call tree. Exceptions can be caught when using the `send` function because then this will return false instead of rethrowing/passing on the exception which is what the `transfer` method does.

4 Example

In this section a small example will be introduced. This example helps understand the problem this research tries to solve. The tool that will parse the specification is not implemented yet, that is why in this example the specification is translated by hand. First we will introduce the contract and what is it supposed to do. Next we will add the specifications to the contract that we wish to check for during runtime. After that we will look at the code of the runtime monitored contract, which is made by translating the specified properties to Solidity code and adding it to the existing contract. Lastly we will discuss what this means for the contract in terms of gas execution and additional changes that had to be made to the contract.

4.1 Simple Token

In this example we will use the contract SimpleToken. The contract code can be found on the Ethereum Foundation website [2]. It models a minimum viable token, to keep the state of the contract a `mapping` is used that maps `address` to `uint`. This mapping is kept in the contracts' internal storage and is stored on the blockchain. It indicates which amount each address holds of this token and changes each time the `transfer` function is called. The `transfer` function requires two parameters an address (`_to`) and an uint(`_value`) which specifies the amount to be sent. The from address is determined from the global variable `msg` which is present in each transaction. The require statements in the transfer function will check if the sender has enough balance to sent the amount specified in `_value`, and test for overflows in the balance of the receiver. If one of the two fails an exception will be thrown and the changes this transaction made will be reverted.
When the contract is created the constructor will be called. In this constructor the initialSupply is given as a parameter. All the initial supply is given to the contract creator (`msg.sender`). The

[2] `https://www.ethereum.org/token`

totalSupply value is assigned and cannot be changed after initialization.
Note that this contract is not ERC20 compliant. ERC20 is the interface that most tokens use to implement the desired functionality. This interface is defined in order for all wallets and exchanges to be able to handle different tokens [3]. The main difference is that this contract does not have an `approve` mapping which lets users approve a certain transfer of tokens. Also this SimpleToken does not allow `minting` or `burning` of tokens, in other words the total supply is fixed. Below we can see the Solidity source code of the contract SimpleToken.

```
pragma solidity ^0.4.20;

contract SimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
  /* TotalSupply is fixed for this token, and does not change. */
  /* It is assigned in the constructor */
  uint256 totalSupply;

  /* Initializes contract with initial supply tokens to the creator of the contract
       */
  function SimpleToken(uint256 initialSupply) public {
    // Give the creator all initial tokens
    balanceOf[msg.sender] = initialSupply;
    totalSupply = initialSupply;
  }

  /* Send coins */
  function transfer(address _to, uint256 _value) public {
    // Check if the sender has enough
    require(balanceOf[msg.sender] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value >= balanceOf[_to]);
    // Subtract from the sender
    balanceOf[msg.sender] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
  }
}
```

4.2 Specification

The above section describes the implementation of the token contract. However there is also a specification given in words as to what the contract should do. A few properties of this specification can be declared explicitly using pre and postconditions or invariants. These properties are important to the functionality of the contract. Properties could be translated to corresponding Solidity code and added to the contract. This code is executed within each transaction and thus the properties are checked and validated at runtime.
The first property is that the value of `totalSupply` is always equal to the sum of all the individual balances in the mapping `balanceOf`. Another property is that when a transfer function is executed the balance of the `_to` address is incremented with the `_value`. And the balance of the sender is decreased with the same value. The first property can be seen as an invariant of the contract and should hold before and after execution of a function. The second property should be checked after the execution of the transfer function and can be represented as a postcondition on that function. The exact syntax on how to declare these properties is not defined yet. This example will use syntax that is close to the syntax that is used by JML.

```
pragma solidity ^0.4.20;

contract SimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
  /* TotalSupply is fixed for this token, and does not change. */
  /* It is assigned in the constructor */
  uint256 totalSupply;


  /* @invariant totalSupply ==  sum = (forall x in balanceOf [ sum +=balanceOf[x]])
```

[3] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

```
  /* Send coins */
  /* @ensures
      balanceOf[_to] == \old(balanceOf[_to]) + _value &&
      balanceOf[msg.sender] == \old(balanceOf[msg.sender]) - _value &&
      forall x : x != _to || x != msg.sender : balanceOf[x] == \old(balanceOf[x])
          &&
      _to != msg.sender
      ||
      _to == msg.sender
  */
  function transfer(address _to, uint256 _value) public {
  }
}
```

4.3 Implementation of RuntimeMonitoredSimpleToken

The specification together with the solidity code should be compiled to a new smart contract. The compiled smart contract will have the same behaviour but with extra assertions added to the code. The tool should be able to parse specifications and add the correct code to the corresponding functions. For this example the code will be made by hand because the tool still needs to be developed. The contract solidity code for the Runtime-Monitored SimpleToken can be seen in the snippet below.
For this example if an assertion in the specification is false, the call will return an error. Later more complex behaviour can be added when this situation occurs. The way the contract works from the outside should not change, because several front-end implementations could depend on it. This means that the added behaviour should be inside the functions that are in the original contract. To accomplish this the transfer function body is moved to a separate private function `transfer_body()`. The original transfer function will do several things. First it stores the current state in the `memory` storage of the contract. This is needed because specifications can use the `\old` keyword to reference to variables before the function execution. Next it checks the invariant and possible preconditions. After that the function body is executed. Lastly the postcondition and invariant are checked, the state before execution is passed as a parameter to the functions.

```
pragma solidity ^0.4.20;

contract RuntimeMonitoredSimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
  /* TotalSupply is fixed for this token. */
  /* It is assigned in the constructor */
  uint256 totalSupply;


  /* ------------------------------------------------------------ */
  /* Array to keep a list of in use addresses */
  address[] public addressesInUse;
  /* Struct to save the state of the balanceOf mapping */
  struct BalanceOfStruct{
    address _address;
    uint256 balance;
  }
  /* ------------------------------------------------------------ */


  /* Initializes contract with initial supply tokens to the creator of the contract
      */
  function RuntimeMonitoredSimpleToken(uint256 initialSupply) public {
    // Give the creator all initial tokens
    balanceOf[msg.sender] = initialSupply;
    /* ------------------------------------------------------------ */
    addressesInUse.push(msg.sender);
    /* ------------------------------------------------------------ */
    totalSupply = initialSupply;
  }

  /* Send coins */
  function transfer(address _to, uint256 _value) public {
    //Save state of current variables needed for validation of properties
```

```
    BalanceOfStruct[] memory  old_balanceOf = new BalanceOfStruct[](addressesInUse.
        length);
    address[] memory old_addressesInUse = new address[](addressesInUse.length);
    for(uint x; x < addressesInUse.length; x++){
      old_addressesInUse[x] = addressesInUse[x];
      old_balanceOf[x] = BalanceOfStruct(addressesInUse[x], balanceOf[
          addressesInUse[x]]);
    }

    // check invariant
    invariant();
    // check requires

    // execute body
    transfer_body(_to,_value);
    // check ensures
    transfer_ensures(_to, _value, old_addressesInUse, old_balanceOf);
    // check invariant
    invariant();
  }

  function transfer_ensures(address _to, uint256 _value, address[]
      _old_addressesInUse, BalanceOfStruct[] _old_balanceOf ) private{
    uint index_to;
    uint index_from;
    for(uint x; x< addressesInUse.length; x++){
      if(addressesInUse[x] == _to){
        index_to=x;
      }
      if(addressesInUse[x] == msg.sender){
        index_from = x;
      }
    }
    uint old_balance_from = _old_balanceOf[index_from].balance;
    uint old_balance_to ;
    if(index_to >= _old_addressesInUse.length){
        old_balance_to = 0;
    }else{
        old_balance_to = _old_balanceOf[index_to].balance;
    }

    bool exp1 = (balanceOf[_to] == (old_balance_to + _value));
    bool exp2 = (balanceOf[msg.sender] == (old_balance_from - _value));
    bool exp3 = true;
    for(x=0; x < _old_addressesInUse.length && exp3; x++){
      if(x != index_to || x!= index_from){
        exp3 == (balanceOf[addressesInUse[x]] == _old_balanceOf[x].balance);
      }
    }
    bool exp4 = msg.sender == _to;
    assert( (exp1 && exp2 && exp3 && !exp4) || exp4);
  }

  function invariant() private{
    uint256 sum;
    for(uint x; x < addressesInUse.length; x++){
      sum += balanceOf[addressesInUse[x]];
    }
    assert(totalSupply == sum);
  }

  /* transfer function original body */
  function transfer_body(address _to, uint256 _value) private {
    // Check if the sender has enough
    require(balanceOf[msg.sender] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value >= balanceOf[_to]);
    // Subtract from the sender
    balanceOf[msg.sender] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
    /* ------------------------------------------------------------ */
    bool inUse = false;
```

```
    for(uint x; x < addressesInUse.length && !inUse; x++){
      if(addressesInUse[x] == _to){
          inUse = true;
      }
    }
    if(!inUse){
        addressesInUse.push(_to);
    }
    /* ------------------------------------------------------------ */
  }
}
```

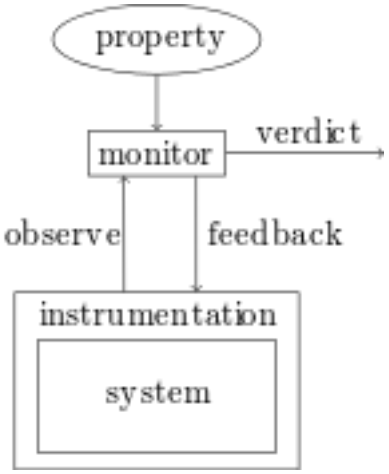
4.4 Discussion

There are a few choices that had to be made during the implementation of the runtime contract. The first observation is that a `mapping` variable is not iterable. This means that properties that use quantifiers and reference mappings cannot be tested at runtime. Mappings in Solidity do not store a keyset and just store the information at the corresponding hash of the key. This limitation can be avoided if we store the used keyset in an additional array. This array must be kept in the storage since it must be persistent between transactions. The SimpleToken contract has only one mapping and the storage array `addressesInUse` keeps track of the keys. A more general approach would be to define a new storage construction called IterableMapping [4].
Another limitation is that mappings cannot be declared in `memory`. Thus to store the variable `\old(someMapping{ key => value}` there is need for either a additional mapping in `storage` or the mapping can be represented by a struct array (which can be stored in `memory`). The last approach is used in the example, but additional analysis is needed to determine which method is the most gas efficient. Gas cost is significantly higher for `storage` in comparison to `memory`, but additional actions are required to have the information converted to a struct variable.
Lastly because of additional checks the gas cost for each function is not constant anymore. This effect is worse compared to the increase of gas cost in normal contracts. For example if the SimpleToken contract had 1000 addresses in use the cost of each function call would not increase in the original contract. But in the runtime-monitored contract the gas costs would increase since the array `addressesInUse` has length 1000 which causes extra iterations for validation. The additional gas cost are not analysed for this SimpleToken contract, but in the final tool tests with multiple transactions should be analysed for their gas cost.

5 Runtime verification

Runtime verification tools are already developed and in use for general purpose programming languages like Java [10]. In these the tool is usually used to detect deadlocks and race conditions. Because these are hard to reason about and using runtime verification the program can reason about a single execution path. It can also serve as an additional testing suite, where runtime specifications are used as a unit test when executing the code. This solves the problem of having test functionality while the system under test has to be under a certain condition.
The basic principle of runtime verification is described in the picture below. This was introduced by Havelund et al. in *A Tutorial on Runtime Verification.*

[4] `https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol`



The property is specified using some formal language. This monitor is then used to observe the system and test the specification at specific moments during execution. These points during executions can be defined within the monitor. For example when a variable is written or before a function is called. The system must sent events to the monitor so that it knows when to act and test the specification. The monitor can then report feedback to the system. The major drawback of using runtime verification is the overhead it creates. This overhead is dependent on the type of property that is checked and how many times during execution.
Runtime verification on smart contracts is different from general purpose languages since race conditions and deadlocks can not occur in the EVM because there is no multi threading. However other properties of runtime verification could be useful. For example monitors can observe the behaviour of a smart contract and when a property fails to satisfy the monitor could revert the transaction completely. This could help developers write more secure smart contracts. The EVM has some limitations because smart contracts are executed by every node. For example the extra computations cost extra gas, which makes the same functionality cost more Ether. Some of these limitations could be solved by checking some properties offline. Each transaction is public, which means that a specific 'Runtime Verification' node could check the properties where the normal node just execute the code without executing the monitor.

6 Related Work

There is a lot of work related to this topic. Ethereum is not the only blockchain platform that supports the deployment of smart contracts, but this section will focus on the development and research for the Ethereum blockchain specifically. There are papers discussing the verification of smart contracts. They can be further categorized as static analysis or formal verification. Additionally other contract languages have been proposed to help writing secure smart contracts. The last subsection discusses some other related work.

6.1 Smart Contract Verification

Due to the recent exploits that were found on the Ethereum blockchain this research area has seen a lot of attention. Especially in the field of formal verification. There are many proposals of verification tools that will help to write secure smart contracts. The security of smart contracts is important because if the bytecode of a contract is committed to the blockchain it cannot be changed afterwards. This means that testing and verification of the code before committing it to the network is important. The efforts can be categorized in two groups; static analysis and formal verification. The first class are tools that analyse the EVM code or a higher level code and check for patterns. Patterns that are known to be vulnerable get reported. The code is not actually executed, only symbolically. The second group is formal verification. These tools work by giving a specification for a given program. The tool then proves that the program is correct for all possible inputs with respect to the given specification. Some tools fully automate this process, some work with a proof assistant. Note that the Solidity code is usually translated to EVM or some intermediate language in which the proofs can be more easily automated.

6.1.1 Static Analysis Tools

There are many tools that are defined in this area. Most of the tools have the same functionality. You can analyse contracts using the Solidity Code or EVM bytecode. These contracts can be analysed locally or from an online provider (Ethereum mainnet or one of the test nets). Examples of such tools are Mythril [11], Securify [12] and Oyente [13]. The Oyente tool also offers the possibility to analyse all the contracts on the whole blockchain. Their tool is not only available on Github but also has a paper which describes the choices made fo the analysis tool. The tools under this category do not test for errors in business logic. For example if a function returns too much ether on a specific input, this will not be detected by static analysis tools.

6.1.2 Formal Verification Tools

To verify a contract a specification has to be written. Specification gives meaning to what the contract should do. However because Solidity is not fit for this most tools are defined at the EVM bytecode level, or introduce an intermediate contract language. These programs are then proven correct considering all possible inputs with respect to the given specification. KEVM [5], a formalization of the EVM in F* [14] and eth-isabelle [15] are very similar. All three tools are able to execute a large set of the official ethereum test suite and are able to proof specifications correct for certain contracts. Other approaches use an intermediate language over which properties can be proven correct. Lolisa [16] and Scilla [17] fall under this category.

6.2 Smart Contract Languages

Smart contracts are usually written in a high level language that compiles to EVM (Ethereum Virtual Machine) bytecode. Currently the best known and most used language is Solidity (as described in detail in section 3). But there are other options available that also compile to EVM bytecode. They differ in their syntax and influences by other languages.

6.2.1 Bamboo

Bamboo is a morphing smart contract language. State transitions are a core part of the language design. This makes the state transitions in smart contracts explicit. This way it avoids re-entrancy by default. Each function is declared within a state and executing a function causes a state transition. This way there should be less surprises in the execution of smart contracts. The project is located in a repository located at `https://github.com/pirapira/bamboo`. As an example the smart contract for a crowd funding is used. The crowd funding usually has several stages in which different things can happen. In Solidity these stages are usually modeled using boolean variables and enforced using `modifiers`. With this approach it is hard to keep track which functions are enabled at which state. In Bamboo this is not the case since functions are declared within a state and functions modify the signature of the smart contract.

6.2.2 Vyper

Vyper is a new and experimental smart contract programming language. It is maintained by the Ethereum Foundation at `https://github.com/ethereum/vyper`. The idea is to limit certain functions and aspects that are possible in Solidity to make writing smart contracts less error prone. It also tries to make smart contracts more human readable to make it simpler to see what will happen when a function is called. For example `modifiers`, inline assembly and class inheritance is not allowed in Vyper as opposed to Solidity.

6.3 Other related work

A number of other proposals have been published which try to make smart contracts more secure. They do not belong to a certain category but are related to the current work. Some projects only have source code available and do not have documentation or a paper.

6.3.1 ContractLARVA

ContractLARVA can be found on github at `https://github.com/gordonpace/contractLarva`. Following the instructions on the README you can write a specification and a contract in Solidity. The compiler will combine these two and output a new Solidity contract with the runtime verification checks in place. Properties have to be specified using *dynamic event automata* (DEA) [18]. The tool is based on a similar tool called LARVA for Java.
For example consider the following Solidity contract. In this contract we would like to monitor the variable `number`, it should always be positive.

```
pragma solidity ^0.4.23;

contract myContract {

    uint public number;

    function setNumber(uint amount) public{
        number = amount;
    }
}
```

The monitor has to be defined in DEA syntax.

```
monitor myContract{
    DEA testMonitor {
        states {
            State: initial;

        }
        transitions {
            State -[number@( number > 0)]-> State;
        }
    }
}
```

The specification and contract are combined into a new contract with the added behaviour. The output of the tool can be seen below.

```
pragma solidity ^0.4.23;
contract LARVA_myContract {
    modifier LARVA_DEA_1_handle_after_assignment_number {
        _;
        if ((LARVA_STATE_1 == 0) && (number > 0)) {
            LARVA_STATE_1 = 0;
        } else {
        }
    }
    int8 LARVA_STATE_1 = 0;
    function LARVA_set_number_pre (uint _number)
        LARVA_DEA_1_handle_after_assignment_number public returns (uint) {
        LARVA_previous_number = number;
        number = _number;
        return LARVA_previous_number;
    }
    function LARVA_set_number_post (uint _number)
        LARVA_DEA_1_handle_after_assignment_number public returns (uint) {
        LARVA_previous_number = number;
        number = _number;
        return number;
    }
    uint private LARVA_previous_number;
    function LARVA_myContract () public {
    }
    function LARVA_reparation () private {
    }
    function LARVA_satisfaction () private {
    }
    enum LARVA_STATUS {NOT_STARTED, READY, RUNNING, STOPPED}
    LARVA_STATUS private LARVA_Status = LARVA_STATUS.NOT_STARTED;
    function LARVA_EnableContract () private {
        LARVA_Status = (LARVA_Status == LARVA_STATUS.NOT_STARTED)?LARVA_STATUS.
            READY:LARVA_STATUS.RUNNING;
    }
```

```
    function LARVA_DisableContract () private {
        LARVA_Status = (LARVA_Status == LARVA_STATUS.READY)?LARVA_STATUS.
            NOT_STARTED:LARVA_STATUS.STOPPED;
    }
    modifier LARVA_ContractIsEnabled {
        require(LARVA_Status == LARVA_STATUS.RUNNING);
        _;
    }
    modifier LARVA_Constructor {
        require(LARVA_Status == LARVA_STATUS.READY);
        LARVA_Status = LARVA_STATUS.RUNNING;
        _;
    }
    uint private number;
    function setNumber (uint amount) LARVA_ContractIsEnabled public {
        LARVA_set_number_post(amount);
    }

}
```

The above example is a contract that can be deployed to a local testnet. However all calls to the function `setNumber` will fail because the code is not initialized correctly. The `LARVA_Status` is never set to running thus the modifier `LARVA_ContractIsEnabled` will throw an exception. This approach has several limitations, for example monitors can only be added with states. Even if the contract does not represent a state machine. The states are represented as `int8` in the generated contract code, which cost extra gas. States have to be initialized in the beginning. This means that the generated contract has to have a constructor and potentially call the original constructor. This changes the contract interface and thus could limit the testing of the contract because other applications could depend on it. To test a certain specification on previous values the variable is stored to a storage location. This causes a lot of extra gas cost where should be possible to store in in `memory`. In the previous example see the variable `LARVA_previous_number`.

6.3.2 Ethereuem-runtime-verification

This project is located at `https://github.com/shaunazzopardi/ethereuem-runtime-verification`. No documentation is available for this project. It mentions the LARVA project in the description in that it differs from LARVA because this runtime-verification tool can dynamically add properties to an already deployed smart contract.

6.3.3 The Hydra Project

The Hydra Framework is a project for smart contracts on the Ethereum network. It tries to make smart contracts more secure by making multiple implementations of the same contract. They call this *N-of-N-version programming*. The different implementations are controlled by a meta contract which forwards the incoming calls to all the implementations. If the implementations do not agree on a single answer, the meta contract will be able to react on this. When such a vulnerability is found a bounty is given to the person who exploited the vulnerability. They call this principle *the exploit gap*, this means that a hacker should claim the bounty instead of exploiting the vulnerability. More information can be found in their paper [19].

6.3.4 FSolidM

FSolidM [20] is a fully functional tool which helps developing secure smart contracts. It provides a GUI to specify contracts using finite state machines (FSM). These FSMs are then translated to secure solidity contract code. This tool helps creating secure smart contracts since the semantics of the FSM is well defined. The tool comes with a code generator for generating Solidity code, and also the possibility to define plugins. These plugins can be used to define certain patterns that implement common design patterns or include security constraints.

6.3.5 Quantitative Analysis of Smart Contracts

Chatterjee et al. [21] analyse the utility (expected payout) for smart contracts. It does so by using game theory and incentives to analyse a stateful game. It uses a simplified contract language and translates these contracts to state-based games. These games can then be analysed by the tool for

their expected payout. The functions in the games are assumed to be executed at distinct timeslots. This is however not the case for Ethereum since one can always write a specific contract to call all functions within the same transaction. Also calls to other contracts are not considered while this is where most of the complexity and vulnerabilities are discovered in real world contracts.

7 Final Project

This document serves as a starting point for the Final Project. This paper describes the requirements and related work of a concept tool that is to be developed in the Final Project. The tool will make it possible to write specifications at the same level of Solidity code. These specifications will then be parsed and can be used for a number of different analyses. For example they can be added as extra checks in the code, which is called runtime verification. Writing the specification at the level of Solidity code helps understanding the problem. It can also be used to generate code that other tools can use for formal verification as mentioned in Section 6.
The result of this are a research questions that will be answered during the development of the tool in the Final Project. The project can be seen in three different steps that have to be taken.

1. **Property specification/definition.** The first step is to decide and analyse which properties should be able to be checked and specified. Properties should make sense and should be able to be checked within Solidity. This raises the question: *What properties should the tool be able to identify and specify?*. Specifically the *syntax* has to be defined. And a *parser* has to be written to decide if properties are according to the defined syntax.

2. **Tool development.** The next step is defining the output of the tool. In other words: *What can be generated from the specification and smart contract source code?* For example some properties can be checked using static analysis and some have to be implemented in Solidity. The properties that are inserted into the Solidity code are checked at runtime. Another possibility is to generate code for different formal verification tools which are described in Section 6.

3. **Tool usage on smart contract.** The last step is to test the tool on real world smart contract. And see if it can detect vulnerabilities that would otherwise have not been found. *How can the tool be used to detect vulnerabilities in smart contracts?*.

7.1 Tasks and Planning

To answer each question different steps have to be taken. For the planning see the diagram below.

1. Property specification

    1.1. Decide on technique/tool. There are multiple options to implement a parser. One of the solutions is to extend the Solidity compiler which is open source on Github [5]. Another solution is to use the ANTLR based grammar which is kept up to date independently [6].

    1.2. Extend Grammar. The grammar has to be extended to accept the specifications that should be inserted into the contracts. It should be able to parse the extended annotations.

    1.3. Design. Design the architecture for the parser and validator. The parser can be generated from the grammar automatically. But the specification has to be validated. For example the identifiers that are used must be present in the contract and only for all constructs can be used on arrays.

    1.4. Implementation. Implement the property specification and syntax validation in the technique that was chosen in task 1.

2. Tool output

[5] https://github.com/ethereum/solidity
[6] https://github.com/solidityj/solidity-antlr4

2.1. Design. Output of the previous step is a parsed and valid specification. Next step is generate the runtime verification code. First we need to decide how to implement this functionality.

2.2. Implementation.

3. Tool Analysis

3.1. Examples. To test the tool examples of smart contracts are needed. These should then be annotated with additional properties and then be passed on through the tool.

3.2. Gas Analysis. One thing to analyse is the extra gas usage the extra checks use and compare it to the gas cost of the base contract.

3.3. Other analysis tools. Other analysis can be done with the specifications. A few of them are mentioned during the introduction, but they could be used to generate code that can be used for a formal verification tool. This means that the solution must be easily extensible.

4. Project

4.1. Report. The report should be updated throughout the project. If a certain task is completed the report should be updated as well.

4.2. Final Presentation.

TASK NUMBER	TASK TITLE	WEEK	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
1	Property specification																							
1.1	Decide on technique/tool																							
1.2	Extend grammar																							
1.3	Design																							
1.4	Implementation																							
2	Tool output																							
2.1	Design																							
2.2	Implementation																							
3	Tool analysis																							
3.1	Example																							
3.2	Gas Analysis																							
3.3	Other Analysis Tools																							
4	Project																							
4.1	Report																							
4.2	Final Presentation																							

References

[1] S. Palladino, "The Parity wallet hack explained," https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

[2] A. Hertig, "Ethereum client bug freezes user funds as fallout remains uncertain," https://www.coindesk.com/ethereum-client-bug-freezes-user-funds-fallout-remains-uncertain/.

[3] "Ether.camps hkg token has a bug and needs to be reissued," https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued.

[4] P. Daian, "Analysis of the DAO exploit," http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[5] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "KEVM: A complete semantics of the Ethereum virtual machine," Tech. Rep., 2017.

[6] G. Wood, "Ethereum yellow paper," http://yellowpaper.io, 2014 (Accessed in 2018).

[7] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, 2013.

[8] A. Dika, "Ethereum smart contracts: Security vulnerabilities and security tools," Master's thesis, NTNU, 2017.

[9] "Solidity documentation," http://solidity.readthedocs.io/en/v0.4.23.

[10] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of jml accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, 2005.

[11] "Mythril," https://github.com/ConsenSys/mythril.

[12] "Securify," https://securify.ch/.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[14] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[15] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.

[16] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the solidity programming language," *arXiv preprint arXiv:1803.09885*, 2018.

[17] I. Sergey, A. Kumar, and A. Hobor, "Scilla: a smart contract intermediate-level language," *arXiv preprint arXiv:1801.00687*, 2018.

[18] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *Formal Methods for Industrial Critical Systems (FMICS)*, ser. Lecture Notes in Computer Science, vol. 5596, L'Aquila, Italy, 2008, pp. 135–149.

[19] L. Breidenbach, P. Daian, F. Tramer, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts."

[20] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 270–277.

[21] K. Chatterjee, A. K. Goharshady, and Y. Velner, "Quantitative analysis of smart contracts," *arXiv preprint arXiv:1801.03367*, 2018.