Solitor: Runtime Verification of Smart Contracts

On the Ethereum network

Lars Stegeman
l.stegeman@student.utwente.nl

November 20, 2018

Master Thesis
Master of Computer Science
Methods and tools for verification specialization

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods and Tools research group

Supervisors
prof.dr. J.C. van de Pol, University of Twente
dr. M.H. Everts, University of Twente

Abstract

The Ethereum blockchain is often called a decentralized world computer. On this blockchain smart contracts are deployed and executed. Smart contracts can control ether and data that is associated with that address. Changes can be made to the contract internals by executing transactions on the set of functions that the smart contract offers. This thesis explains the background of the Ethereum network and how smart contracts execute on the blockchain. This is important because smart contracts are committed to the blockchain. This means that the contract code is public and unchangeable. Ensuring the contract executed like intended is important because vulnerabilities can not be easily solved. A number of real world vulnerabilities have been detected and exploited on smart contracts. Many tools and solutions have been proposed to make it easier to develop secure smart contracts.

This thesis introduces the tool Solitor. Solitor is short for Solidity (runtime) monitor. It is a tool developed specifically for smart contracts on the Ethereum network. Solitor can parse and translate annotations in Solidity contracts to Solidity code which checks the annotation at runtime. Annotations can be used to check if certain properties hold during execution of the smart contract. These can either be contract invariants or pre and postconditions for methods. In general annotations are logical expressions that can reference contract variables and blockchain specific identifiers. To recognize the annotations the original Solidity grammar is extended and is similar to that of the Java Modelling Language (JML). Furthermore the thesis describes two case studies, where the tool is used to specify correct behaviour or detect a vulnerability.

Contents

1 Introduction

Ethereum is a decentralized platform that runs smart contracts. The platform is powered by a blockchain that is shared between all connecting parties. This blockchain contains all the transactions that these smart contracts use. The blockchain also stores the currency of Ethereum called Ether. Compared to Bitcoin it is more focussed to be a smart contract platform. On this platform applications will run without any trusted central party. This makes these applications unstoppable and censorship resistant. Each day new smart contracts are deployed to the Ethereum network. Smart contracts can be seen as decentralized application that can do computation and store/retrieve information from the blockchain. Users can communicate with smart contracts using transactions. These transactions are also stored in the blockchain which means they cannot be refused or reversed. Smart contracts are written in a language called Solidity. Solidity can be seen as a contract oriented programming language. It is high level and compiles to Ethereum Virtual Machine (EVM) bytecode. This is the actual code that is deployed to the blockchain and executes when a transaction is done. 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. Many different analysis tools have already been developed. Static analysis tools can be executed on many contracts and detect mistakes by analyzing known vulnerable patterns. Other tools which use formal verification need a specification to be able to guarantee a contract behaves 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. More examples of tools can be found in Section 4.

1.1 Goal

The goal of this research is to develop a tool that can do runtime verification for smart contracts. The annotations to check properties can be written at the level of Solidity. This will make it easy for Solidity developers to use the tool. Furthermore the specification does not have to be complete and proven correctly against all specifications like in the case of formal verification tools. This makes it easy to check for certain properties without having to specify all the behaviour. From the annotations Solidity code can automatically be generated. The generated code will be Solidity which can be executed on the blockchain like any normal contract. The tool will be made specifically for the language Solidity and for the Ethereum blockchain. 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.
- The annotations can be used by static analysis tools for other purposes. This can also work in combination with the current runtime verification. If a certain annotation can be proven statically, it does not have to be checked at runtime. On the other hand annotations that can not be proven statically can be checked at runtime.

1.2 Research Questions

A runtime verification tool for smart contracts has to be usable in the environment it will be used. The properties it can specify must be implementable in Solidity. The setting is very different from a general purpose programming language. For example the separation of storage and memory is different. Contracts have to be annotated with a certain syntax. This syntax has to be designed in such a way that it is understandable and usable. Furthermore the usability of the tool as a whole

should be tested on a case study of a smart contract. More concretely the following questions are answered in this thesis:

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?*

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?*.

1.3 Thesis Structure

This thesis will answer the above questions and introduce the tool Solitor. Before that some background information is given in Section 2. This introduces the setting in which these smart contracts are executed. It explains the workings of the blockchain in combination of the executed code. The network state and contract state is explained in detail. Next the language Solidity is introduced in section 3. This is the programming language that is used to develop smart contracts on the Ethereum network. It compiles to the EVM (Ethereum Virtual Machine) bytecode and is specifically designed for developing contracts. The language is introduced so that the design decisions for the tool can be understood. Section 5 discusses the tool in a high level overview. The next sections 6-8 discuss the different phases in the tool process. Some of the limitations within Solitor are discussed in Section 9. The tool is tested on two case studies. The first case study is a contract which models a subcurrency. This is called a token and many applications use such contract. The contract SimpleToken is a simplified version and a property is implemented and checked at runtime. The second case study is a contract which contains a vulnerability. The vulnerability is exposed using annotations. When executing the contract with annotations the vulnerability becomes visible and execution of the transaction is stopped. This can be seen in detail in Section 10. As said in the introduction many tools try to make smart contract development more secure. There are many approaches each focussing on a specific aspect of secure smart contracts. The different approaches and vulnerabilities they detect are discussed in Section 4. Lastly the conclusion of the thesis can be seen in Section 11. It briefly answers the questions asked in this introduction and discusses the results of Solitor.

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 [5]. 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 [6]. 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 thesis 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. The sender of a transaction has to specify the maximum amount of gas it wants to spend and the amount of ether per unit gas. This way the sender pays the network for executing the transaction. The gas cost of each EVM instruction is defined in the protocol and can not be changed. Instructions that are more intensive for the blockchain cost more gas. For example storing a value on the blockchain costs more gas then storing it in memory. 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 thesis 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 [7] 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 [8]. 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 and it offers the common types in traditional programming languages: 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

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

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 compiles to EVM bytecode which 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. 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 re-throwing/passing on the exception which is what the `transfer` method does.

4 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.

4.1 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.

4.1.1 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 [9], Securify [10] and Oyente [11]. 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.

4.1.2 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 [12], a formalization of the EVM in F* [13] and eth-isabelle [14] 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 [15] and Scilla [16] fall under this category.

4.2 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.

4.2.1 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.

4.2.2 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.

4.3 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.

4.3.1 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) [17]. 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`.

4.3.2 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 [18].

4.3.3 FSolidM

FSolidM [19] 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.

4.3.4 Quantitative Analysis of Smart Contracts

Chatterjee et al. [20] 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.

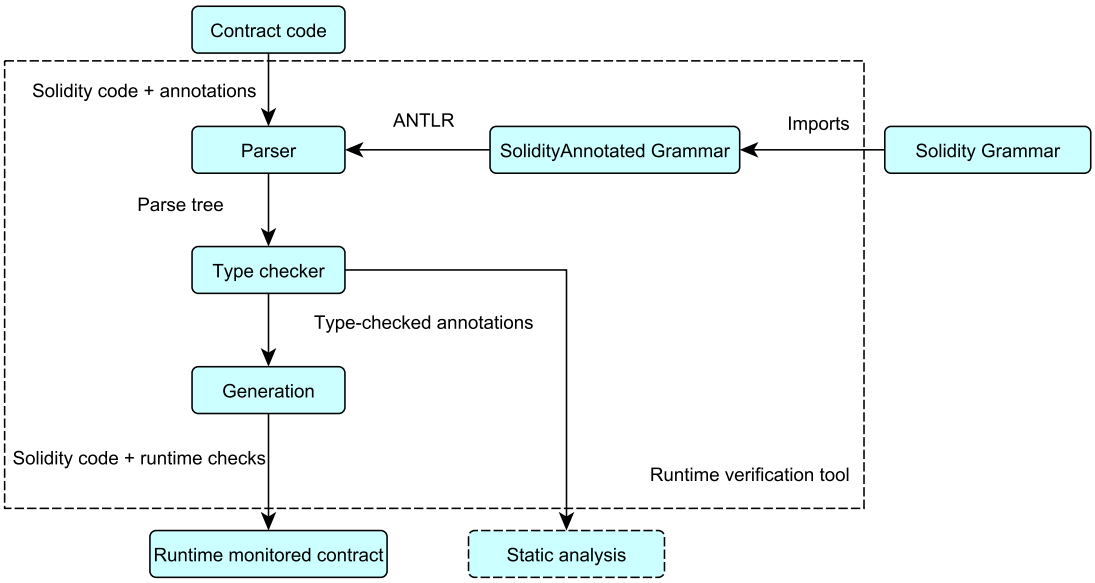

Figure 1: Overview of the tool Solitor

5 Solitor

The following sections introduce the tool Solitor. The tool can parse smart contracts written in Solidity which have extra annotations in them. These annotations will be translated to Solidity code which can be checked at runtime. This way assumptions about the contract state can be expressed and tested. Using this tool the security of smart contracts can be improved.

5.1 Overview

In Figure 1 the complete overview of the tool Solitor can be seen. Within the dashed square the implemented parts are visible. The arrows indicate the flow of the contract code throughout the program.

First contract code has to be annotated according to a specified grammar. Section 6 explains the grammar in more detail and gives some example annotations. The tool ANTLR [21] is used to generate code for the lexer and parser. The grammar has to be expressed in the language that is recognized by the ANTLR tool. The automatically generated parser is used to parse Solidity contract code and annotations into a parse tree. The parse tree makes it possible to walk the complete contract code and do analysis on specific parts of the contract. This parse tree is used in later stages of the tool.

The next step is type checking the annotations. This uses the parse tree to examine the annotations and check if they are valid. The type checking is done bottom up and works in two phases. The first phase collects all the relevant variables. This includes state variables and function definitions (function name, arguments and return values). The next phase uses this information to do the actual type checking of the annotations. This is explained in more detail in Section 7.

The result of the type checker phase are type-checked annotations. In practice these are parse tree objects in which the types correspond to the operators used and the identifiers that are used are also defined in the contract. This is used as input for the generation phase. The generation phase will operate on the information that is created during the type checker phase. For each annotation it will generate the code that is needed to check it during runtime. This happens in a single pass of the complete parse tree. Details on this phase can be found in Section 8.

The output of the type checker phase can also be used for static analysis tools. The benefit of using the tool to validate the annotations is that the result is a type checked parse tree that can be parsed and traversed in various ways to be useful for static verification methods.

6 Annotation Language

The first step is defining an annotation syntax, and formally write this down using a grammar. The parser generator that we use is ANTLR [21]. Using the grammar definition the lexer and parser will be automatically generated. The output of this phase is a parse tree that can be used in later stages of the tool. We use the parser generator ANTLR, mostly for two reasons. The first reason is that there already exists a actively maintained grammar definition for the complete Solidity language [22]. The second reason is the grammar inheritance capabilities of ANTLR. This is done by inheritance over the original grammar [2]. It functions much like object oriented inheritance. The main grammar inherits all rules, token specifications and named actions from the imported grammar. Rules in the main grammar override rules in the imported grammar. We will use this principle to extend the grammar of Solidity to recognize the special annotations that will later be used in the tool. In this case the imported grammar is the original Solidity grammar. The 'new' main grammar is defined further below and is called SolidityAnnotated. The advantage of this approach is that changes to the original Solidity grammar can easily be updated in the tool. This only holds for small changes to the language, if grammar rules change that the tool makes use of the SolidityAnnotated grammar also has to be updated.

6.1 Solidity Annotated

The original Solidity grammar has to be extended to recognize the annotations that will be defined. The annotations have certain requirements that can be summarized in the following way. Later each requirement is discussed in detail.

- Annotations can be specified at the top level of the contract.
- Annotations should be able to reference all variables used in the contract.
- Basic math operations can be used within annotations.
- Annotations can not have side effects.
- The type should be boolean at the highest level (that way they can be verified).
- There are three types of annotations: invariants and pre- or postconditions to a function.

The annotation syntax is heavily inspired from the JML annotation syntax [23]. But has a lot less built-in keywords since the setting is easier and the tool is less complex. Only top-level annotations are necessary because they are used for runtime generation. Inline annotations are usually used for loop-invariants or to help the verification engine in other annotation languages. Since Solidity is a contract-oriented language, the functions, variables and structs are all defined within the contract. All annotations should be able to make use of them. Variables are either defined in the contract as a global variable, or used as function parameters. The annotations themselves should contain logic to check a certain property that is defined by the annotation. These properties are built from basic math operations and variables and should result in a boolean at the highest level. The boolean is needed because in the runtime verification the annotation is actually checked when the contract code is executed. The three types of annotation that are defined are invariant, precondition and postcondition. This is sufficient since no other contract can make changes to the internals of the contract memory or storage. This means that all access from the contract is from the functions that are defined. This way having preconditions to check annotations before a certain function, and postconditions to check them after is enough for individual functions. Invariants are defined for contracts, they make sure a property holds at all times. The only time these could change is when a function is executed. In practice this means that for each invariant it has to be checked at the end of every function.

6.2 Grammar Definition

The following section explains what these requirements mean for the grammar definition. The original Solidity grammar is extended in such a way that annotations can only be defined on the top level. The relevant parts of the original Solidity grammar can be seen in the snippet below.

[2] This principle is explained in detail here `https://github.com/antlr/antlr4/blob/master/doc/grammars.md`

This does not include the full grammar specification but only the parts that are relevant for the annotation syntax.

```
grammar Solidity;

sourceUnit
  : (pragmaDirective | importDirective | contractDefinition)* EOF ;

contractDefinition
  : ( 'contract' | 'interface' | 'library' ) identifier
    ( 'is' inheritanceSpecifier (',' inheritanceSpecifier )* )?
    '{' contractPart* '}' ;

contractPart
    : stateVariableDeclaration
    | usingForDeclaration
    | structDefinition
    | constructorDefinition
    | modifierDefinition
    | functionDefinition
    | eventDefinition
    | enumDefinition ;
```

In the original grammar the definition of `contractPart` is what defines the declaration of variables and the definitions for structs and functions. This is where the extra annotations have to be added to the grammar. The snippet below shows the basic definition of an annotation. This is not the complete grammar: some of the tokens are omitted from this snippet, since they are not required to understand the grammar definition.

```
grammar SolidityAnnotated;
import Solidity;

@header {package generated;}

//Added annotationDefinition. This enables annotations to be on the
    top level only.
contractPart
  : stateVariableDeclaration
  | usingForDeclaration
  | structDefinition
  | constructorDefinition
  | modifierDefinition
  | functionDefinition
  | eventDefinition
  | enumDefinition
  | annotationDefinition ;

annotationDefinition
  : AnnotationStart AnnotationKind annotationExpression;

// Same as the expression rule except it does not include
   assignments, only comparisons
annotationExpression
  : '(' annotationExpression ')'
  | '!'annotationExpression
  | ('\\forall' | '\\exists') '(' identifier 'in' identifier ':'
     annotationExpression')'
  | annotationExpression integerOpInteger annotationExpression
  | annotationExpression integerOpBoolean annotationExpression
  | annotationExpression compareOp annotationExpression
```

```
  | annotationExpression booleanOp annotationExpression
  | primaryAnnotationExpression;

primaryAnnotationExpression
  : primaryExpression
  | primaryAnnotationExpression '.' identifier
  | primaryAnnotationExpression '[' primaryAnnotationExpression ']'
  | '\\old' '(' primaryAnnotationExpression ')';

//Annotation Tokens
AnnotationStart
  : '//@';

AnnotationKind
  : 'inv'| 'pre'| 'post';

// Added '->' for then.
booleanOp
  : '&&' | '||' | '->';

compareOp
  : '==' | '!=';

integerOpBoolean
  : ('>'|'>='|'<'|'<=');

integerOpInteger
  : '+' | '-';

// Remove '@' from first position of LINE_COMMENT token.
LINE_COMMENT
  : '//' ~[@] ~[\r\n]* -> channel(HIDDEN);
// Send whitespace to channel hidden.
WS
  : [ \t\r\n\u000C]+ -> channel(HIDDEN);
```

An AnnotationDefinition is composed of multiple components. It consists of `AnnotationStart`, `AnnotationKind` and `annotationExpression` components. The `AnnotationStart` token is used to signal that an annotation definition is coming next. This is defined as '//@' making it a line comment to other solidity compilers. This makes annotated solidity code still compilable by normal Solidity compilers. For the grammar to accept this notation the `LINE_COMMENT` token has to be adjusted to not accept '@' as a second character. Otherwise all annotation comments would be recognized as a `LINE_COMMENT` making it unusable.

There are three types of annotations that are defined by the token `AnnotationKind`. They can either be an invariant or a pre- or post-condition of a function. Invariants are defined per contract, and should hold at any point during the execution of the contract. Pre- or post-conditions are defined for a specific method. They are checked before and after execution of the method. Each annotation has an expression which has to be evaluated called `annotationExpression`. The expression parser rules are separated between `annotationExpression` and `primaryAnnotationExpression`. This is needed to keep the hierarchy in parsing and prevent using complex expressions within primary definitions. For example using the keyword '\old' before parenthesis. The annotation expressions use a different parser rules than the expression rules that are used within the original Solidity grammar. The `annotationExpression` does not allow syntax like `expression + '++'` and to distinguish these a new parser rule was introduced for annotations only.

The order in which the different subrules are defined in the `annotationExpression` is important. The order indicates the priority which the subrules are given. This means that parentheses bind stronger then any other rule, followed by the negation rule with the expression `!` and so on. The `annotationExpression` construction contains all the logical operators that can be used within annotations. In general they are of the form `expression - <operand> - expression`.

The expressions are defined recursively thus making it able to form longer expressions with multiple operands. The parser rules in `primaryAnnotationExpression` are used as leaves in the expression. `primaryExpression` reverts to different kinds of literals that are used in Solidity. The other rules deal with complex types of Solidity and the possibility to reference an old variable. `primaryExpression` and `identifier` are parser rules that are defined in the original Solidity grammar. The annotation expressions make use of these rules so that they do not have to be defined again. These rules do not include assignments and are without side effects. The `primaryExpression` parser rule includes all the literals that can be used within Solidity. The parser rule `identifier` is used for all kinds of identifiers such as function identifiers and variable identifiers. Function calls are not allowed within annotations, for more details see Section 9.

6.3 Examples

In this section a couple of annotation examples will be given for example contracts. First a contract snippet is shown and later the meaning of this annotation is explained.

```
uint256 nr1;
uint256 nr2;
//@ inv nr1 >= nr2
```

Defines an invariant that will be checked at the start and end of every function. `nr1` and `nr2` are global contract variables. `nr1` should always be bigger than `nr2`.

```
address owner;
//@ post \old(owner) == owner
function doSomething() public{
    // ...
}
```

Defines a post condition on the function `doSomething()`. Checks if the owner is not changed during execution of the function.

```
uint256[] a;
//@ inv \forall(x in a: a[x] > 0)
```

Defines an invariant that will check if all elements in array `a` are positive.

```
uint256 b;
//@ post (msg.sender == owner) -> (\old(b) != b)
function changeSomething() public{
    // ...
}
```

Postcondition for the function `changeSomething()`. If the sender of this transaction is equal to the owner (`msg.sender`), variable `b` must be different from the start of the function.

```
mapping(address => uint256) myMap;
address public adr;
//@ inv myMap[adr] == 5
```

Example of a mapping that maps `address` to `uint256`. The invariant checks the key `adr` in the map and checks if it is equal to 5.
A few example of expressions that do not parse correctly:

```
uint256 a = 5;
//@ inv getNumber() == a

function getNumber() returns(uint256){
    //...
}
```

Functions are not recognized in annotations. This is because some functions could have side effects. This can be checked in typechecker but is not implemented yet.

```
//@ post old(_a + _b) == _a + _b
function doSomething(uint256 _a, uint256 _b){
    //...
}
```

The construction `\old()` can only reference primary expression and not complex expressions.

These examples are small and will not compile because there is no contract code wrapping them. They only show the possibilities of the annotation language. Later on larger examples will be shown which include complete annotated contracts and are able to compile.

7 Annotation Type Checking

With the annotation language defined, the next step in the process is validating annotations. The annotations will be parsed and the type of each identifier will be checked. The types of these identifiers should match to the context, and the identifiers should be defined. This is important for the annotations since they will be transformed to Solidity code in later phases of the tool.

7.1 Design

Annotations have to be validated on certain aspects for them to be meaningful. These aspects have to be verified first for the annotations to be useful in the next generation phase. The parser ensures annotations are syntactically correct. However, there are more properties that have to be checked. The typecheck phase will consist of two passes that walk the complete parse tree. The first walk will collect all the variables and defined structures and store these in an information object. The second walk will type check each annotation individually. During this type checking the type of each identifier is looked up using the collected information from the first walk.

7.2 Implementation

During the first phase all the variables, structs and function definitions are stored in an object. This object is later used by the second phase to retrieve information.

```
public class ValidationInformation {
  ArrayList<SolidityVariable> identifiers;
  ArrayList<SolidityFunction> functions;
  ArrayList<SolidityStruct> structs;

  ...
}
```

- SolidityVariable is an object which has a name (the identifier) and a type. These model state variables in a contract.
- SolidityFunction is an object which represent a function and stores the name and arguments. The arguments are of type SolidityVariable.
- SolidityStruct is an object that represents struct definitions in a solidity contract. It stores the name and elements. Elements are again of type SolidityVariable.

As mentioned in Section 3 Solidity has many types. To make generation and typechecking easier the types are reduced to 8 base types (`uint256, uint128` etc are all regarded as `INTEGER`). These are all represented in the enumeration `SolidityType`, and all the internal representations of contract code make use of it. The internal representations must deal with nested constructions. For example consider the following solidity code:

```
struct A {
    B b;
}
struct B {
    uint256 nr;
}
A var1;
```

For this Solidity code the typechecker would create two `SolidityStruct` objects, `A` and `B`. Struct `B` contains a variable `nr` of type `INTEGER`, struct `A` contains a variable `b` of type `STRUCT` with reference to `B`. There also is a global variable `var1` with type `STRUCT` with reference to `A`.

The next phase will only parse the annotation part of the contract code. This means that the entire parse tree of the original Solidity contract code will be ignored. The actual typechecking happens in this phase. It works bottom up, getting the type of each identifier and verifies the types of each step. The top level of each annotation should result in the type `BOOLEAN`. An extra type `UNDEFINED` was added to the SolidityType object to deal with cases where the identifier

was not found and to produce a result without crashing the program. For the rest of the parse rules/operators the following type system is used:

- Base case: expression is a `primaryAnnotationExpression`. This could mean a identifier where the type is found through the `SolidityVariable` or a literal of some type. The type can just be passed on to the higher level.
- '!' `expression`: Type checker verifies the nested expression and validates this results in `BOOLEAN`. Result of this step is always `BOOLEAN`.
- \forall | exists(identifier in identifier: expression): There are multiple things that have to be verified. First the second `identifier` should be of type `MAPPING` or `ARRAY`. Secondly the nested expression is typechecked, this is within a special scope since expressions can make use of the first identifier. This expression should result in `BOOLEAN`. This result is also returned for the higher level expression.
- `expression` ('+' | '-') `expression`: Both subexpressions should return `INTEGER`. Result of the current expression is `INTEGER` as well.
- `expression` ('>'|'>='|'<'|'<=') `expression`: Both sub-expressions should return `INTEGER`. Result of the current expression is `BOOLEAN`.
- `expression` ('==' | '!=') `expression`: The types of the sub-expressions should match. The result of this is `BOOLEAN`.
- `expression` ('&&' | '||' | '->') `expression`: Both sub-expressions should return `BOOLEAN`. Result of this is `BOOLEAN` as well.

If any of the types do not correspond to the expected value a validation error is reported and logged.

In case complex types are used such as structs, the additional information is retrieved from the corresponding object. This information can be retrieved from the object that is referenced. For example consider an identifier `a.b`. This would mean that the type of `a` must be a struct and that in the definition of the struct the type of the identifier `b` must be retrieved. Additionally annotations can make use of function arguments and reference them. This is solved by looking up the `SolidityFunction` object that the annotation was declared above. This makes it possible to retrieve the types of function arguments and use them within the annotation.

7.3 Example

There are annotations which are valid according to parser rules, however they contain an error when type checking them. A type error can occur when types do not match the logical operator that is being used. An other possibility is that an identifier that is used is not defined in the contract. A small example contract is given below.

```
pragma solidity ^0.4.24;

contract TypeError {
    uint256 a = 5;
    //@ inv a == b

    address a1;
    address a2;
    //@ inv a1 + a2  >  5

}
```

When type checking the contract Solitor will output error messages. Firstly the identifier `b` is not defined, thus the type cannot be retrieved. The second error message is because the type defaults to `UNDEFINED` when the type is not found. This makes the comparison operator fail because the types do not match. The third error message is because both identifier reference to an address, and these can not be used with the add operator. Solitor will output the following error messages, when it is called with the contract above:

```
Line 5:17 - Identifier b in annotation not defined as variable
Line 5:12 - Expected type to match at a==b but is INTEGER, UNDEFINED
Line 9:12 - Expected types to be integers at a1+a2 but is ADDRESS, ADDRESS
```

8 Generation of runtime monitoring code

After type checking annotations, the generation phase starts. In this phase the annotations are transformed to Solidity code and added to the contract. The functions of the contract will not change, but extra code is added which only checks certain properties. Since the interface does not change, the front-end can communicate to the runtime monitored contract like it is the original contract. Most of the constructions used in annotations can be directly translated to Solidity code. However mappings cause problems because the key set is unknown. To solve this, extra code is added to the contract to store this information. This is discussed in detail in Section 8.3

8.1 Design

The important requirement for this phase is that the interface of the contract does not change. That is, the publicly callable functions and their arguments should not change. Therefore we will wrap the original function in a new function that calls the original function. The functional behaviour of the contract should remain the same and the added code only performs extra checks. For each added annotation three steps have to be performed:

1. Generate a function for each annotation that checks the expression. This function should have the correct number of arguments that are used within the annotation. Arguments are variables that are not reachable from the global scope and used in the expression. These are `old` variables and function variables. The only variables that are available within a function with no arguments are the globally defined variables. These do not have to be given as an argument since then you have two variables defined with the same name. Other variables have to be given as an argument to the function. This includes function arguments, these are arguments that the original function has where this annotation is defined for. This is empty by construction for invariants since they do not reference a function. Additionally pre- and postconditions can use the \old() construction on a variable. With this expression the value before the function execution is referenced. The same value also has to be given as an argument to the annotation function.

2. For each original function of the contract: Create a wrapper function with the old name which calls the original function body.

3. Add all annotations that should be checked to the wrapper function. All variables that should be stored before the function call should be stored in memory before executing the function body. This means that all variables that are reference in an annotation with the keyword \old() have to be stored in a variable with name 'variable + _old'.

To illustrate the steps consider the small example below. Some details are abstracted away since they do not add information to the example. The expression "invariant_expression" is not valid syntax, but can be replaced with any arbitrary expression. The same holds for "post_expression". Furthermore the parameters of the method `annotation1` are replaced with "argument" since it is unknown which parameters should be included. In the implementation section it will be explained which argument variables are copied and which are not.

```
//@ inv 'invariant_expression'

//@ post 'post_expression'
function testFunction() public{
    //...
}
```

At the end of the generation phase this will be the generated code:

```
//@ inv 'invariant_expression'
function annotation0(){
  //check 'invariant_expression'
}

//@ post 'post_expression'
function annotation1('argument'){
```

```
  // check 'post_expression'
}

function testFunction() public{
  //stored variables in memory, i.e. copy all variables in 'argument'
  annotation0();
  testFunction_body();
  annotation1('argument');
  annotation0();
}

function testFunction_body() private{
    //...
}
```

8.2 Implementation

During the typechecking phase the object `AnnotationInformation` is created for each annotation it parses. This object contains the following information:

```
public class AnnotationInformation{
  ParserRuleContext node;
  private String name;
  private Map<SolidityVariable, Boolean> variables;
  private String type;
  private String function;

  ...
}
```

- Reference to the node in the parser tree.
- A unique name for the annotation. This name is generated and is 'annotation' + a number.
- A list of arguments used in the method. The boolean indicates if it should be included as an argument for the method. As explained in the Design section. These are all arguments that cannot be reached from the global scope.
- Type of the annotation. Invariant, Pre or Post-condition. This indicates where it should be added to the wrapper function.
- Function the annotation is declared above. Null in case this is an invariant.

When parsing the tree the correct `AnnotationInformation` object is retrieved using the parse tree node. The function declaration is constructed using the name and arguments from this object. The body of this function can be constructed from the `expression` that is within the parse tree object. In most of the cases the code can directly be printed but in some cases the expression has to be altered:

- The expressions `\forall` and `\exists` have to be replaced with a loop, that checks the expression for all the elements of the collection.
- The usage of `\old` has to be replaced with the variable that was created and initialized before the function execution. This means rewriting the variable `\old(identifier)` to `identifier_old`. In case of structs this is a bit more works since `\old(a.b)` should be replaced with `a_old.b`.
- Boolean operator `a->b` which can be used in annotations is not valid Solidity code, and has to be replaced with `!a||b`

Since the annotations have been type-checked the generated code will always be valid. The result of an annotation is always a boolean value. The actual testing of this value is done using an `assert` statement. If the value is false it means that the transaction will be reverted and all gas will be consumed. The actual test of the boolean value could be changed, to for example the triggering of an `event`. The difference in approach is important for the result of an transaction when the annotation is not satisfied. Assert will revert the entire transaction. A possible vulnerability that triggers the assertion will not go through as the assertion stops the execution. The vulnerability can not be exploited. In the case of changing this to an `event` the vulnerability can be exploited, but it will be registered because of the emitted event. Other behaviour can be specified here, for example calling a function that halts all the functionality of a contract except for a special function that only the admin of the contract can execute.

When the parser encounters a function it will check if annotations have to be added. This means searching through all `AnnotationInformation` objects and see if any of them reference the current function or are null (invariant). The original function will be transformed to a private function and renamed to "functionName"+ _body. The new wrapper function will call this function. The wrapper function does two other things that are important that is saving the current state of variables that are later to be used in annotations. It will only do this for variables that are referenced within `\old(identifier)`. The second thing it does is add function calls to annotations before and after executing the original body. Annotations of type invariant are added before and after, preconditions only before and postconditions only after.

The actual printing of the contract code works with a `TokenStreamRewriter`[3], which is part of the ANTLR framework. The problem with printing the contract code is that the parse tree skips all whitespace. This means that the whitespace is lost in the parsing process. Including the whitespace in the parser rules would complicate the grammar because they have to be added to every rule. The ANTLR book also has an example on a similar case like this [4]. The idea is that the lexer gives all tokens a number in order, but the tokens are split between two channels. The parse tree only parses tokens on the first channel and ignores the others. The `TokenStreamRewriter` has the information of both channels. The token positions are still known and this way the `TokenStreamRewriter` can replace/insert text based on parser rule nodes positions. The tool uses the rewriter to insert the annotation blocks and function body in the correct places while preserving whitespace of the original contract code.

8.3 Mappings

The above section holds for all constructions except for `mappings`. A mapping cannot be stored in memory and is not iterable since the key set is not known. This gives problems 1) when storing the variable for `\old()` and 2) checking a expression for all elements in a mapping.

This is solved by using an iterable mapping [5]. This works by wrapping the mapping definition within a struct. This struct contains the mapping and an array. The array stores the indexes which are used. Since the indexes are in an array these can be iterated. The key and value can be retrieved using the index. The value can also be retrieved using the key. In order for this to work extra code has to be added to the contract.

- Import the library at the beginning of the contract.

```
import "./itMapsLib.sol";
```

- Use library function when doing operations on the struct.

```
using itMaps for itMaps.itMapUintUint;
using itMaps for itMaps.itMapAddressUint;
using itMaps for itMaps.itMapUintAddress;
```

- Replace declaration of mapping with iterable mapping.

```
//mapping(address => uint256) a;
itMaps.itMapAddressUint a;
```

[3] `https://www.antlr.org/api/Java/org/antlr/v4/runtime/TokenStreamRewriter.html`

[4] Source code can be found here: `https://pragprog.com/titles/tpantlr2/source_code`. The example is in lexmagic/ShiftVarComments.java

[5] `https://github.com/szerintedmi/solidity-itMapsLib`

- Create mapping_old for annotation purposes. Only when the mapping is used within a \old() expression.

```
itMaps.itMapAddressUint a_old;
```

- Replace each mapping reference with `.get(...)` or `.set(...)` depending on the context.

```
//uint256 b = a[0x0];
uint256 b = a.get(0x0);
//a[0x1] = 1;
a.set(0x1, 1);
```

Adding these constructions to the contract is costly. Firstly because the keyset must be stored in the storage of the contract which costs gas. Secondly because a second iterable mapping must be used to store the old values of the mapping.

The most important change is that the contract storage changes because of this construction. A change in storage is necessary because the keyset must be stored in order for this to work. Another solution would be to store a separate keyset array within the contract and not use an iterable mapping construction. The keyset has to be updated on every access of the original mapping. This keyset would be used to construct the old mapping which can then be done in memory. However using this approach a lot of custom Solidity code has to be generated which is not implemented in Solitor. These limitations are discussed more in detail in the next section.

9 Limitations

There are some limitations of Solitor that are present in the current version of the implementation. Some of the limitations are in the annotation language and some of them are because of the generated Solidity code. The annotation language and parser only work on a single scope level. This means each declared variable can only be used once. However in Solidity contract inheritance is supported. This makes it possible to define small contracts and inherit from these contracts. This makes development of smart contracts modular and scalable. The tool currently does not support inheritance. This means that the tool only parses the main contract and does not look in the inheritance tree. Annotations can only make use of definitions that are declared within the current contract. To solve this problem the tool would have to be extended with a scope mechanism. Different variables are declared in different contracts and thus have a different scope. Annotations should be able to reference inherited properties and make use of the variables. This requires some changes to the parser structure, but not the annotation language. Because the syntax to reference to variables of inherited contracts is the same as accessing a variable from a `struct`. The parser should look up all the inheritance statements and parse all contracts that are referenced by `import` statements. These contracts can be declared in separate files.

Since mappings are not iterable in Solidity these are replaced by iterable mappings as explained in Section 8.3. However iterable mappings are only replaced for a few types of mappings. These are mappings which maps `uint` $\Rightarrow$ `address`, `address` $\Rightarrow$ `uint`, `uint` $\Rightarrow$ `uint`, all other combinations are not supported. This is because the library that is used does not yet implement other mapping combinations. Also nested mappings are excluded from the generation. The iterable mapping solution that is used in the current version of the tool uses a global variable to store the \old state of the mapping. Using another function within a function which uses an annotation would overwrite the global variable, making it unusable to check the annotation expression. This limitation is only due to the generation of Solidity code for the annotation. The annotation syntax can handle these nested mappings and other declared mappings.

10 Case study

In this section we present two case studies of smart contracts. These smart contracts will be annotated and the tool will generate the contract code with extra checks added to the code. The generated contracts are then tested using the Truffle framework [24]. This framework makes it possible to generate transactions and execute them on a local temporary blockchain. A few transactions will be used to detect the vulnerability or show that contract behaves correctly. The first example is a minimal token implementation. The second example is a contract with a vulnerability which we will show can be detected with the correct annotations.

10.1 SimpleToken

In this example we will use the contract SimpleToken. The contract code can be found on the Ethereum Foundation website[6]. 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 contract's internal storage and is stored on the blockchain. It indicates the amount of tokens each address holds token and changes every 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 send 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 totalSupply value is assigned and cannot be changed after initialization.

Note that this contract is not ERC20 compliant[7]. 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. 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 ^0.4.23;

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]);
```

[6] https://www.ethereum.org/token
[7] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

```
    // Subtract from the sender
    balanceOf[msg.sender] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
  }
}
```

10.1.1 Annotation

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 post-conditions or invariants. These properties are important to the functionality of the contract. The first 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 rest of the balances remains the same. This property should be checked after the execution of the `transfer` function. Using the correct syntax the annotation will look:

```
//@ post
 (balanceOf[_to] == (\old(balanceOf[_to]) + _value)
 && balanceOf[msg.sender] == (\old(balanceOf[msg.sender]) - _value)
 && \forall(x in balanceOf: (x != _to && x != msg.sender) ->
    balanceOf[x] == \old(balanceOf[x])))
 || (msg.sender == _to  && \forall(x in balanceOf: balanceOf[x] ==
    \old(balanceOf[x])))
```

Going over it line by line:

1. Indicate that this is an annotation and not a comment. And this is of type post condition.
2. Balance of `_to` gets incremented by `_value`
3. Balance of `msg.sender` gets decremented by `_value`
4. Rest of the balances does not change.
5. This makes the annotation also valid for the special case where `_to` is equal to `msg.sender`. In this case none of the balances change.

10.1.2 Generated Code

When the tool gets executed with the above contract as input including the annotation, the tool will parse the annotation and output generated solidity code with extra checks. Note that the functionality of this contract is exactly the same as the previous code but with extra checks. This means that this approach is only feasible for development purposes. Since the gas cost of executing the previous contract is much lower then that of the runtime monitored contract. Also the more addresses that get added to the iterable map the more gas the transaction will consume. The complete generated code can be seen below.

```
pragma solidity ^0.4.23;

import "./itMapsLib.sol";

contract SimpleToken {
  /* This creates an array with all balances */
  using itMaps for itMaps.itMapUintUint;
  using itMaps for itMaps.itMapAddressUint;
  using itMaps for itMaps.itMapUintAddress;
  itMaps.itMapAddressUint balanceOf;
  itMaps.itMapAddressUint balanceOf_old;
  /* 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.insert(msg.sender,initialSupply);
    totalSupply = initialSupply;
  }

  /* Send coins */
  //@ post (balanceOf[_to] == (\old(balanceOf[_to]) + _value) &&
     balanceOf[msg.sender] == (\old(balanceOf[msg.sender]) - _value)
     && \forall(x in balanceOf: (x != _to && x != msg.sender) ->
     balanceOf[x] == \old(balanceOf[x]))) || (msg.sender == _to && \
     forall(y in balanceOf: balanceOf[y] == \old(balanceOf[y])))
  function annotation0(address _to, uint256 _value) view private{
    bool expression0= true;
    for(uint256 i0=0; i0<balanceOf.size() &&expression0;i0++){
      var x= balanceOf.getKeyByIndex(i0);
      expression0=!(x!=_to&&x!=msg.sender)||balanceOf.get(x)==
    balanceOf_old.get(x);
    }
    bool expression1= true;
    for(uint256 i1=0; i1<balanceOf.size() &&expression1;i1++){
      var y= balanceOf.getKeyByIndex(i1);
      expression1=balanceOf.get(y)==balanceOf_old.get(y);
    }
    assert((balanceOf.get(_to)==(balanceOf_old.get(_to)+_value)&&
    balanceOf.get(msg.sender)==(balanceOf_old.get(msg.sender)-_value)&&
    expression0)||(msg.sender==_to&&expression1));
  }

  function transfer(address _to, uint256 _value) public {
    balanceOf_old.destroy();
    for(uint256 mapcopy=0; mapcopy < balanceOf.size(); mapcopy++){
      balanceOf_old.insert(balanceOf.getKeyByIndex(mapcopy),balanceOf.
    getValueByIndex(mapcopy));
    }

    transfer_body(_to, _value);
    annotation0(_to, _value);
  }
  function transfer_body(address _to, uint256 _value) private {
    // Check if the sender has enough
    require(balanceOf.get(msg.sender) >= _value);
    // Check for overflows
    require(balanceOf.get(_to) + _value >= balanceOf.get(_to));
    // Subtract from the sender
    balanceOf.insert(msg.sender,balanceOf.get(msg.sender)-_value);
    // Add the same to the recipient
    balanceOf.insert(_to,balanceOf.get(_to)+_value);
  }
}
```

10.1.3 Testing the contract

To execute the transactions on the contract you need the truffle test suite. The test cases are located in test_SimpleToken_generated.js. The test transactions do the following in this order:

1. Create contract, constructor is called with 1000 initial supply and gets assigned to creator account.
2. Send 100 tokens from account 1 to account 2.
3. Send 10 tokens from account 1 to account 3.
4. Send 100 tokens from account 2 to account 1.
5. Send 100 tokens from account 1 to account 1 (transfer to self).
6. Send 1000 tokens from account 1 to account 3. This should fail since the balance of account 1 is lower then 1000.

The test cases can be executed with the command `truffle test`. Since this uses the runtime monitored contract code, each time the annotations are checked as well. This ensures that the rest of the balances does not change when a transfer is done between account 1 and 2. The test are executed and all succeed.

10.2 Vulnerable Contract

The next example is a simple contract with a vulnerability. The vulnerability will not be easily visible when reading the contract, but with extra annotations added this is visible. This contract is not actually in use but is based on a CryptoRoulette contract [8]. Most of the contract code is omitted for this example. The idea is that the contract keeps a list of messages, for each message the sender and the message are saved. Only the address `admin` has extra privileges to possibly delete messages or delete the contract (these functions are not in the snippet).

```
pragma solidity ^0.4.23;

contract LogContract {

  address public admin;
  uint256 public nrOfMessages;
  Message[] public messages;

  struct Message {
    address sender;
    string msg;
  }

  constructor() public{
    admin = msg.sender;
  }
  //@ post admin == \old(admin) && (nrOfMessages == (\old(nrOfMessages)
    + 1))
  function logMessage(string _msg) public{
    Message message;
    message.sender = msg.sender;
    message.msg = _msg;
    messages.push(message);
    nrOfMessages++;
  }

}
```

[8] https://github.com/misterch0c/Solidlity-Vulnerable/blob/master/honeypots/CryptoRoulette.sol

10.2.1 Annotation

The annotation that is added to the function `logMessage` checks the basic behaviour of the function. The address `admin` should not be changed, and the number of messages should be increased by one. This should be checked after the function is executed. The `admin` address is set once by the constructor of the contract. No other function changes the values of this address variable, thus it makes sense to add this as an annotation.

```
//@ post admin == \old(admin) && (nrOfMessages == (\old(
    nrOfMessages) + 1))
```

10.2.2 Generated Code

The code that is generated stores the original variables in memory before calling the function body. After the function is executed the old variables will be compared to the current state variables. Every call to the function `logMessage` will report an error since each time the state variables `admin` and `nrOfMessages` are changed. These are changed because the struct `message` within the function body defaults to `storage`.

```
pragma solidity ^0.4.23;

contract LogContract {

  address public admin;
  uint256 public nrOfMessages;
  Message[] public messages;

  struct Message {
    address sender;
    string msg;
  }

  constructor(){
    admin = msg.sender;
  }
  //@ post admin == \old(admin) && (nrOfMessages == (\old(nrOfMessages)
    + 1))
  function annotation0(address admin_old, uint256 nrOfMessages_old)
   view private{
    assert(admin==admin_old&&(nrOfMessages==(nrOfMessages_old+1)));
  }

  function logMessage(string _msg) public{
    address admin_old = admin;
    uint256 nrOfMessages_old = nrOfMessages;
    logMessage_body(_msg);
    annotation0(admin_old, nrOfMessages_old);
  }
  function logMessage_body(string _msg) private{
    Message message;
    message.sender = msg.sender;
    message.msg = _msg;
    messages.push(message);
    nrOfMessages++;
  }

}
```

10.2.3 Testing the contract

To execute the transactions on the contract you need the truffle test suite. The test cases are located in test_VulnerableContract_generated.js. The test transactions do the following in this order:

1. Create contract, admin set to the creator of the contract.
2. Send transaction to log function with message "test".

The test cases can be executed with the command `truffle test`. Since this uses the runtime monitored contract code, each time the annotations are checked as well. This ensures that when executing the function `log` the address of the admin does not change, and that the number of stored messages gets increased by one. However we know that this contract contains a vulnerability and the admin address will change because of the function call. This causes the transaction to revert because the assertion in the annotation will fail. This error is caught by the test, and thus this test succeeds.

11 Conclusion

Security is an important aspect for developing smart contracts. Solitor helps the programmer write more secure Solidity code for the EVM. Many tools try to improve security of smart contract each developed for a specific vulnerability. Solitor works by defining annotations at the level of Solidity code. This makes it easy to understand for developers. The usability of the tool is shown in two case studies of small smart contracts. Using the annotations a vulnerability was exposed and thus can not be exploited. The environment of smart contract code is different from that of normal executable code. Vulnerabilities cannot be solved because the code is committed to the blockchain. The generated code that Solitor adds not only costs extra computational time, it also costs extra gas. This means that transactions will cost more for runtime monitored smart contracts. Solitor is thus best used in beta versions of the smart contract. It could be deployed on a test net in order to detect vulnerabilities. Later in the main net version these vulnerabilities that are detected could be patched and then deployed.

The annotation language was inspired from the syntax of JML (Java Modeling Language). JML is designed specifically for Java and has a lot more keywords built in. However these keywords do not apply to the blockchain setting. Settings that do apply to the blockchain like global transactions variables have been added to the tool in order for the parser to recognize these variables. There are two types of annotations, invariants must always hold and are defined for the complete contract. Pre- and postconditions are defined for contract methods. These are checked before and after execution of the function.

The annotation expression is checked at runtime. Solidity code is generated according to the expression that is given in the annotation. Expressions can also be specified for a collection (mapping or array in Solidity), in that case the generation will generate the loops to check this expression for all elements in the collection. Mappings are not iterable by default in Solidity, since the key set is not known. To solve this and be able to check expressions at runtime iterable mappings are used. These iterable mappings store the key in a separate array and wrap the construction in a struct.

Solitor is publicly available on Github for every developer to use. For instructions see Appendix A. Annotations are comments to a normal Solidity compiler, so they still can be compiled to EVM bytecode. The interface of a runtime monitored smart contract does not change. This makes Solitor easy to use and replace the normal contract with the runtime monitored contract.

11.1 Future work

There are a few aspects of future work that could help improve the tool:

- The tool should be able to handle contract inheritance.
- Implement a generic iterable construction, that can be stored in memory.
- The tool should have a mechanism for exposing the parse tree for other programs after validation.
- The tool could be used in so called bounty programs
- The tool could be used in combination with a fuzzer, to generate better test cases.

The first point is really important for this tool to be useful for larger decentralized applications. Because most of these applications use a structure of smaller contracts which the main contract inherits from. For example see the secure smart contract repository of OpenZeppelin [9]. To implement this functionality a different level of scope has to be added. This is not that difficult since Solitor uses a `SolidityVariable` to store each variable. Each variable should be kept in a map which only contains the variables of that scope. Annotations should be able to make use of variables declared from inherited contracts. Smaller contract could also have annotations themselves and should also be considered when parsing the main contract. The point that requires the most work is actually finding the contracts and parsing them. It would mean that the parser needs to do a prerun of the inheritance structure and find all corresponding contract code. This could also be present in imported contracts, which can be at different locations.

[9] `https://github.com/OpenZeppelin/openzeppelin-solidity`

Currently only mappings of specific types are able to be used in generated code. A generic iterable construction would solve this issue. To implement this each mapping has to be parsed and checked which types it used. For each key (this can recursive) an array must be created which stores them. This way the keyset of each mapping is known. Each addition and deletion from this mapping should be changed in order for the key array to keep up with the mapping, the same way it does for the iterable mappings that are used now. When the mapping is referenced in an annotation the complete mapping should be stored in an array of structs. The elements of the structs have to be determined from the elements of the mapping. This means that each mapping would require an additional struct definition to store the data. The benefit of this approach is that for an array of structs it is possible to store it in memory, which is not possible in the current implementation. This involves a lot of extra parsing (of each mapping) and generation (struct definitions) and is not implemented in the current version of the tool.

One of the benefits of this approach is that annotations could be used by other analysis tools as well. Currently there is no way to use the annotations that are parsed by this tool in other programs. This, however, could be easily implemented by having an extra command line option to only parse the annotations and not go in the generation phase.

A recent trend with smart contract deployment is a bounty program. The smart contract is deployed on a testnet and when a vulnerability is found the user is rewarded with some Ether. With the correct annotations, Solitor can be used to define the bounty criteria and automatically generate such a contract. For example when a certain property should always hold during contract execution, this property can be defined as an invariant. Solitor will automatically generate a check for this property, and insert it before and after each function. When such property fails to satisfy the transaction will be reverted because the property is wrapped in an `assert` statement. However, this could also be changed to return a certain amount worth of ether when such property fails.

Fuzzing or fuzz-testing is a technique used in software development to test pieces of software. For smart contracts this can also be used to test smart contracts against various inputs. This technique could be used in combination with Solitor. The main issue with runtime monitoring is that it requires a lot of test input to test if the contract functions correctly. For Solidity there exists a fuzzer called Echidna [25] that could be helpful in combination with annotation.

A Tool Usage

The code of the tool is publicly available on the repository `https://github.com/LarsStegeman/EthereumRuntimeMonitoring`. Instructions on how to use the tool on other contracts can be seen in the README.

A.1 Getting Started

A.1.1 Prerequisites

- Java 1.8
- Maven 3.5.3
- ANTLR 4 (Maven will get this automatically when installing)

A.1.2 Installing

To install this tool perform the following steps
Clone this repository

```
git clone https://github.com/LarsStegeman/EthereumRuntimeMonitoring
```

Build the tool using maven

```
mvn package
```

Run the tool with basic example

```
mvn exec:java
```

This should output the file `basicAnnotations_generated.sol`.

A.2 Using the tool

A.2.1 Grammar examples

For example annotations see Section 6. The grammar for the annotation language can be found in the file SolidityAnnotation.g4. More example annotations can be found in the test directory.

A.2.2 Run the tool on other contracts

When the tool is built, a jar file is also generated to use the tool on other contracts. The jar file is located in the `target` directory. Use the tool on other contracts using the command

```
java -jar .\target\Ethereum-RuntimeVerification-1.0.jar     <my-annotated-contract.sol>
```

This will generate a file `my-annotated-contract_generated.sol`

A.2.3 Parameters

For debugging and testing the tool has two parameters which can be switched on. They can be found in Parameters.java. The boolean `DEBUG` enables extra print statements to be printed while parsing. The boolean `STOPONERROR` makes it so that the tool does not stop on the first error it encounters.

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

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

[21] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[22] "Solidity grammar for ANTLR 4," https://github.com/solidityj/solidity-antlr4.

[23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman *et al.*, "Jml reference manual," 2008.

[24] "Truffle framework," https://truffleframework.com/.

[25] "Echidna," https://github.com/trailofbits/echidna.