UCLouvain

École polytechnique de Louvain

Smart Contract Modeling

Comparison of techniques and concrete applications

Authors: **Alexandre Halbardier, Martin Meerts**
Supervisor: **Axel Legay**
Readers: **Eduard Baranov, Lionel Metongnon, Ramin Sadre**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Abstract

A Smart Contract is an executable piece of code residing within a blockchain network. Users can interact with it and use it as an automatized third party, in order to ensure correct transactions. Once a Smart Contract has been deployed in the blockchain, its source code cannot technically be modified anymore. Unfortunately, users are able to interact with vulnerable contracts unconsciously and it may cost millions of dollars due to naive bugs leading to unexpected behaviours. A way to prevent those behaviours is by verifying Smart Contracts before their deployment. This master thesis approached this question by comparing three powerful techniques : Static analysis, Fuzzing and Model Checking. These techniques are able to verify several security properties on Smart Contracts. This work will focus on the comparison between those approaches, on the improvement of an existing analysis tool and finally on the detection of vulnerabilities in a real Smart Contract project named Wablieft.

Keywords— Ethereum, Blockchain, Smart Contract, Fuzzing, Model Checking, Static Analysis, Security, Vulnerabilities

Acknowledgements

First of all, we would like to thank our supervisor Axel Legay who allowed us to work on this very interesting master thesis topic. He advised us and put us on the right track to discover the different types of existing tools and properties to check. Our thanks also go to his two assistants, Lionel Metongnon and Eduard Baranov. Lionel helped us for the analysis of the Wablieft project contracts and checked the results with us. Eduard explained to us the timed-automata representation of Wablieft. This helped us to better understand its behaviour. Thanks to Ramin Sadre for accepting our invitation of being part of our jury. We would also like to thank the developers of the Securify2 tool: Petar Tsankov and Yannis Sachinoglou. They helped us to understand how the tool worked, and thanks to our exchanges we were able to make a contribution to improve their project. We also thank Christophe Bockish and Bugra Kaan Yildiz who spent some hours to explain their Java to Uppaal representation converter and Leonardo Alt, the main developer of Solidity SMT Checker who answered several of our questions.

More personally, we wanted to thank our parents and our girlfriends Marine and Pauline for supporting us during the writing of this master thesis. In addition, We also want to thank our friend Gautier for the proofreading of our thesis.

Contents

Chapter 1

Introduction

The concept of blockchain appeared in 1991 with a study that described a cryptographically secured chain of blocks storing time-stamped data [1]. Unfortunately, this technology didn't find any relevant usage since 2008, the year of Bitcoin's introduction. Bitcoin[2] system is a decentralized and peer-to-peer cash system based on a public ledger in which each time-stamped transaction of bitcoins (i.e the currency used within this blockchain) between users is stored. The first advantage of Bitcoin is its independence from centralized organizations, such as banks, governments or any authority. The second is the anonymization such that it is hard to re-identify the users. Last but not least, the immutability of blocks allows to keep track of each bitcoin generated since the beginning of the blockchain.

Due to the increased popularity of Bitcoin, many other blockchains emerged, such as Ethereum[3], Ripple[4], etc. Ethereum blockchain is the second largest blockchain right behind Bitcoin. The key difference between Bitcoin and Ethereum blockchains, behind cryptographical facts, is that Ethereum transactions support Smart Contracts. Smart Contracts are pieces of code written in a Turing-complete language, such as Solidity[5], running within the Ethereum Blockchain and acting as a virtual third-party for transactions between users, such as in fig 1.1.

For now, Smart Contracts are written by developers, and we all know that humans are failable. Since any appended block of the Ethereum Blockchain is immutable, a Smart Contract cannot be updated after its deployment. If a developer leaves errors within his code, the contract might be vulnerable against malicious users who could exploit those vulnerabilities and cause lot of damages, as you can see in fig 1.2.

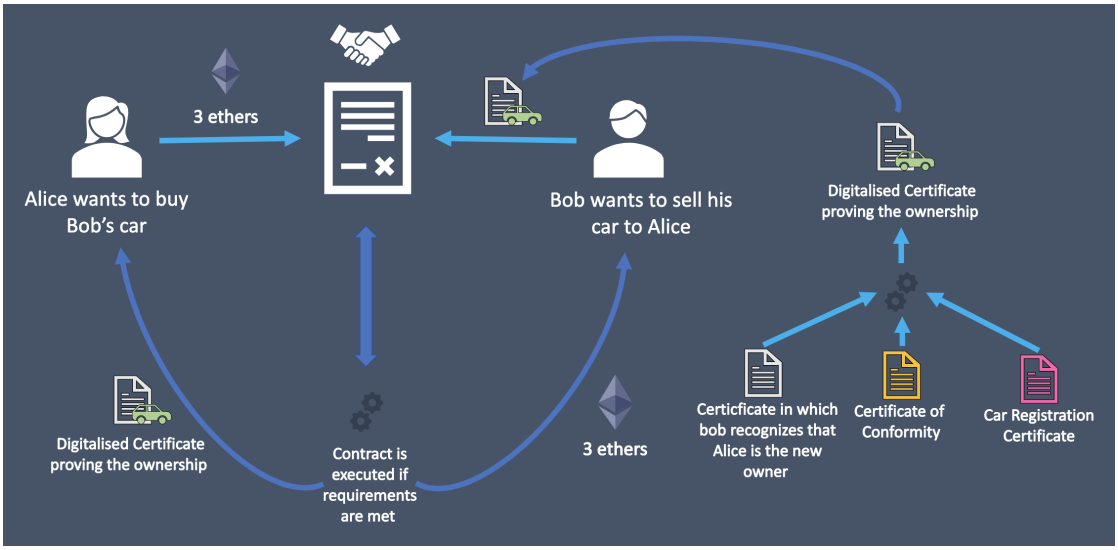

Figure 1.1: Alice buys Bob's car for 3 ethers. The car documents (Certificate of Conformity, Car Registration and the certificate in which Bob recognizes legally the transmission of the ownership) are aggregated within a certificate proving that the ownership of the car belongs to Alice. Alice sends 3 ethers to the Smart Contract while Bob sends the aggregated certificate too. If the requirements are met, the contract proceeds to the transaction. Alice becomes the legal owner of the car and Bob receives 3 ethers.

Indeed, since its introduction in 2015, the Ethereum blockchain suffered from many attacks which have caused hundreds of millions of dollars in losses. The most famous one is the 2016 DAO attack[6] allowing an hacker to steal approximately 160 million dollars worth in Ethers, the Ethereum cryptocurrency (with an actual worth of 16 billion dollars in may 2021). Fortunately, a consensus between the parties of the Ethereum blockchain managed to refund the real owners by performing an hard-fork on it, which basically consists in a rollback of each transaction that happened during and after the attack. Naturally, rollbacks are used the least possible since all of the correct transactions are also reverted. This shows the importance for developers to write correct Smart Contracts. A correct Smart Contract satisfies its specifications independently of any action applied on it. Nowadays, many verification techniques are available in order to verify Smart Contracts before their deployment.

Our approach was to compare three of those techniques with each other : **Model-Checking, Fuzzing and Static-Analysis**. Among a list of recent and open-source tools in each category, we selected one tool per technique. The

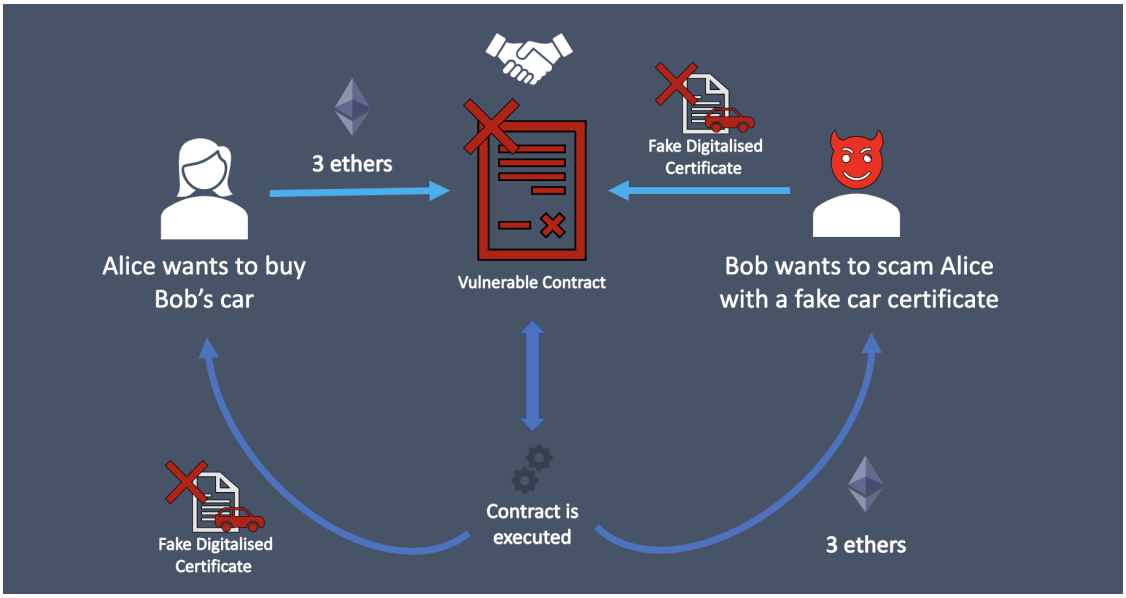

Figure 1.2: Bob is a malicious hacker exploiting the vulnerability he found in the contract.

comparison would tell us whether they were complementary, their limitations and their field of actions. At first, we picked several known vulnerabilities and described the properties they were relying on. Then we selected Smart Contracts containing each of these vulnerabilities in order to figure out whether the three tools discovered them. After the testing phase, we finally described the advantages and disadvantages of each tool and explained their differences. During this master thesis, we had the opportunity to improve a widely used open-source tool named Securify2 which performs static analysis on Smart Contracts. In addition, we were given Wablieft, a UCLouvain project instantiated in Solidity which defines a shared marketplace for medical services, improving the medical organization. Testing those three techniques on Wablieft helped us in our comparison since we had more contracts to treat. In addition, we were able to discover several vulnerabilities within the project and report them to the developers in order to help them improve it.

Chapter 2

Background

2.1 Blockchain

Blockchain[7] is a technology that consists in a digital public ledger without a central authority (i.e. decentralized). Users could perform transactions and record them in this public ledger such that once a transaction has been approved by a consensus mechanism played by some nodes of the network, it cannot be modified anymore. Moreover, all transactions are transparent and each user is anonymized. From a technical point of view, this ledger is based on a growing list of data blocks, each block is cryptographically linked to the previous one. This implies that a data block could not be changed without changing all subsequent blocks.

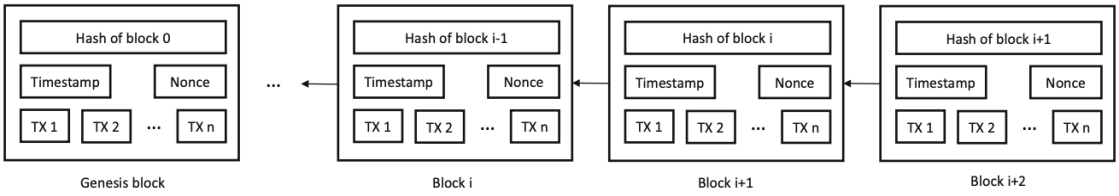

Figure 2.1: Generic Blockchain's data representation[8]

Its storage and its communication system are peer-to-peer, decentralized and the data-history of this ledger cannot technically be lost because several copies are stored on different nodes. The data stored within the blocks composing this ledger is usually an aggregate of transactions (e.g transactions for Bitcoin's blockchain) and metadatas about the blocks such as timestamp or a hash value corresponding to the previous block.

The most known blockchain is the Bitcoin's Blockchain and is mostly used for financial transactions. But other blockchains have more applications such as

performing elections[9], distributed-learning for healthcare's domain[10], IoT[11],etc.

2.2 Ethereum Blockchain and Smart Contracts

Ethereum Blockchain is similar to Bitcoin blockchain[12] in many points : each is based on its own digital coin, is decentralized and uses a distributed public ledger to store time-stamped records of all transactions between users. But technically, they differ in many ways. This document won't cover all of these differences but will focus on this one : Ethereum Blockchain can contain executable codes called "Smart Contracts"[13] which reside within the Ethereum blockchain. Smart Contracts are composed of functions callable by real users. They have their own storage located at their assigned address allowing them to store important data. Each contract has its own Ethereum wallet storing Ethers, the currency of Ethereum.

Ethereum accounts are controlled by real users, which also have a dedicated address and wallet. They can interact with Smart Contracts through transactions. But first, what is an Ethereum transaction[14] and which parties are involved ? A transaction is an action initiated by a real user towards a target address which may be a Smart Contract, another user or a specific address (0x0) used only for contract-deployments.

After the transaction has been initiated, it is broadcasted through the network and then validated by a Proof of Work Consensus mechanism carried out by the network nodes, which is based on the computational power of the miners, but tends to evolve in a proof of stakes consensus protocol[15].
There are 3 types of nodes[16] composing the Ethereum Blockchain : Light nodes, Full nodes and Archive nodes. An archive node is used to store all historical data of the Ethereum Blockchain but is not involved in the blocks validation process. The two other kinds are usually called Miners nodes because they participate in blocks validation. The only difference between Light nodes and Full nodes is the data they store. A Full node will store the full state of the blockchain while the Light nodes will only contain headers and then request the data they need over the network.

Concerning the code of a Smart Contract, it's an executable code written in high-level languages which is Turing-Complete. This code is run on the Ethereum Virtual Machine (EVM[17]) when a transaction towards its address is initiated. EVM is a virtual and sandboxed 256-bits machine, residing within nodes of the Ethereum network.

To illustrate how Ethereum transactions work, here are two examples of how they are performed and validated by nodes:

- From User to User : Alice wants to send 1 ether to Bob. Alice's account will be debited and Bob's account will be credited. The transaction, which does not involve any Smart Contract, will be mined and both account balances will be updated within the blockchain state.
- From User to Smart Contract : Let's imagine a Smart Contract VendingMachine remotely connected to a vending machine owned by Bob. Alice wants to get a snack from it. At first, she calls the function getSnack() of contract VendingMachine, created by a developer and deployed within the Ethereum Blockchain.

  This action will send 1 ether to the contract. After receiving the ether from Alice's address, the executable code of the contract is run on the EVM of the nodes mining the transaction. This code will update the state of the machine's contract wallet and which will unlock the desired snack for Alice. Bob can then withdraw ether from the vending machine contract. You can find a technical example of a transaction involving a contract on figure 2.2.

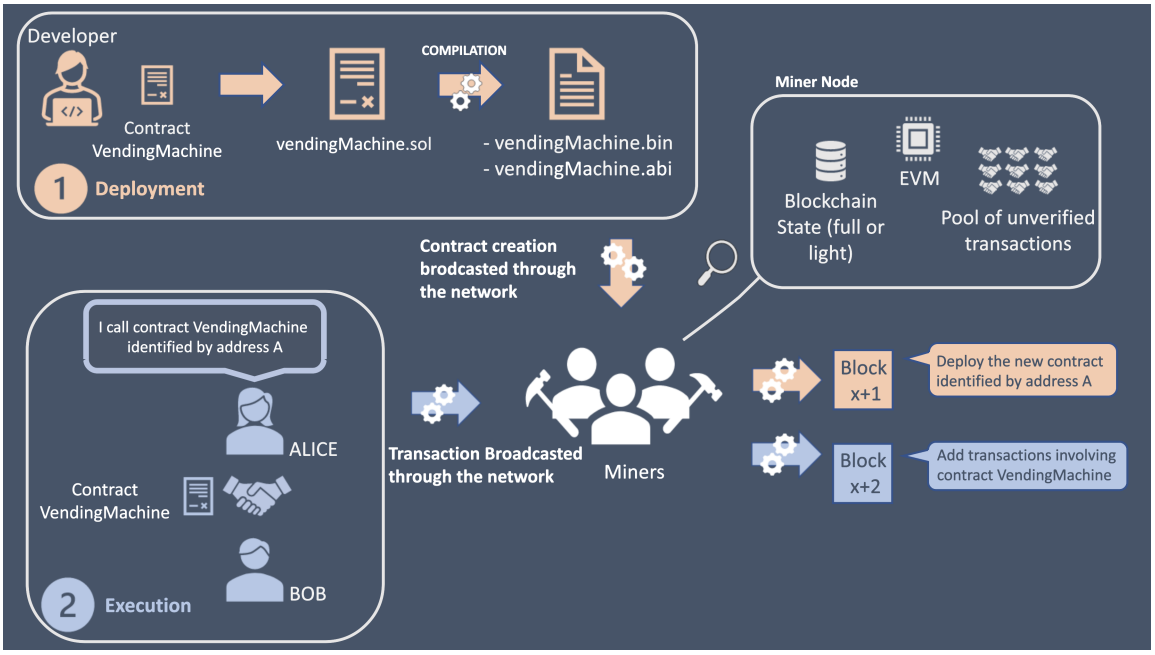

Figure 2.2: Ethereum transactions with Smart Contract

In this figure, a developer creates the contract VendingMachine which allows Bob to sell snacks with his vending machine. VendingMachine.sol is compiled into

VendingMachine.abi and VendingMachine.bin files, mined and identified within the blockchain by the address A. Then Alice will be able to use the contract by specifying a function to call, some inputs and the address of the Smart Contract VendingMachine. The miners will receive this transaction and aggregate it with other transactions in a block. The block will be appended to the Ethereum Blockchain and the transaction will finally be completed.

In fact, a Smart Contract plays the role of a third party automatizing the transaction. Let's take another example with Alice and Bob in which a Smart Contract could take the role of Vinted [18]. Bob wants to sell a t-shirt to Alice via the contract (replacing Vinted):

1. Bob and Alice both send the t-shirt and the money to the contract.
2. The contract responds by sending the t-shirt to Alice.
3. Alice has accepted the transaction (i.e the t-shirt is conform).
4. The contract sends the money to Bob. The transaction is now completed, thanks to the contract avoiding a third party and avoiding people to get robbed.

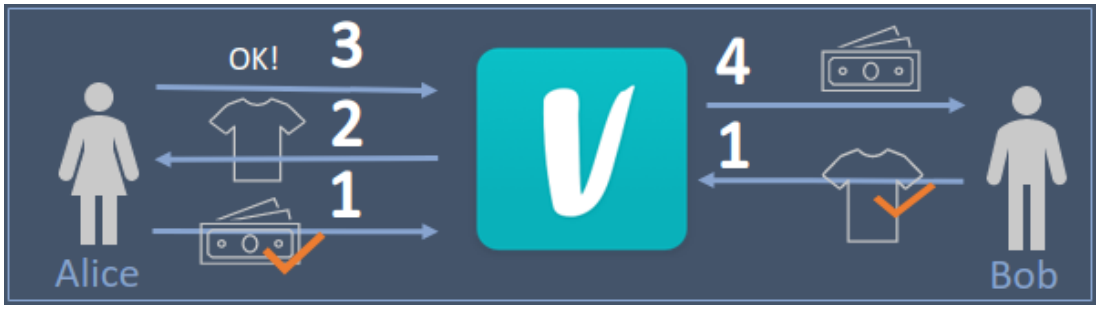

Figure 2.3: Vinted Third Party Scheme

The last important point about Ethereum Blockchain is gas. Gas is a unit of resource meant to be consumed. Each time an instruction is run on the EVM, it costs gas. Each transaction has a gas limit in order to prevent people from surcharging the network with complex tasks within contract functions. If a transaction finished correctly, any remaining gas is refunded to the user which initiated the transaction. Otherwise, if the gas limit has been reached, the transaction is aborted and reverted. This ensures the termination of a function.

2.3 Solidity

Solidity[5] is the most common used programming language for Smart Contracts deployed on the Ethereum Blockchain. Its syntax is very similar to Java. For instance Class equivalences in Java are called contracts in Solidity. Global variables within contracts are persistent while function parameters and local variables are temporary. The set of types available is also similar to Java : integers, unsigned integers (uint8 to uint256), Booleans, Dynamically-sized byte arrays (Strings and bytes), floats, data-sets, etc. There are also additional types such as addresses which allow to call external contract functions. Each solidity contract is compiled into an EVM bytecode. This bytecode is also very similar to Java Bytecode and is run on EVM. An example of a small Smart Contract representing the vending machine of the previous section is provided right below.

```
1  pragma solidity 0.6.11;
2
3  contract VendingMachine {
4
5      // Declare state variables of the contract
6      address public owner;
7      mapping (address => uint) public snackBalances;
8
9      // When 'VendingMachine' contract is deployed: Set the
           deploying address as the owner of the contract, Bob
10     constructor() public {
11         owner = msg.sender;
12     }
13
14     // Allow anyone to purchase a snack that cost 1 ether
15     function getSnack() public payable {
16         require(msg.value <= 1, "You must pay at least 1 ETH per
               snack");
17         snackBalances[owner] += 1;
18         snackBalances[msg.sender] -= 1;
19     }
20
21     // Allow the owner (Bob) to collect his money
22     function collectMoney(uint amount) public {
23         require(msg.sender == owner);
24         payable(owner).transfer(amount);
25     }
26 }
```

2.4 Program Analysis Methods

From the known program analysis methods, this section will describe 3 of them : Model Checking, Static analysis and Fuzzing (divided into Black Box fuzzing, Grey Box fuzzing and White Box fuzzing). These 3 approaches depend on their dynamic/static and automatic/manual behaviours. Many tools provide combination of multiple techniques.

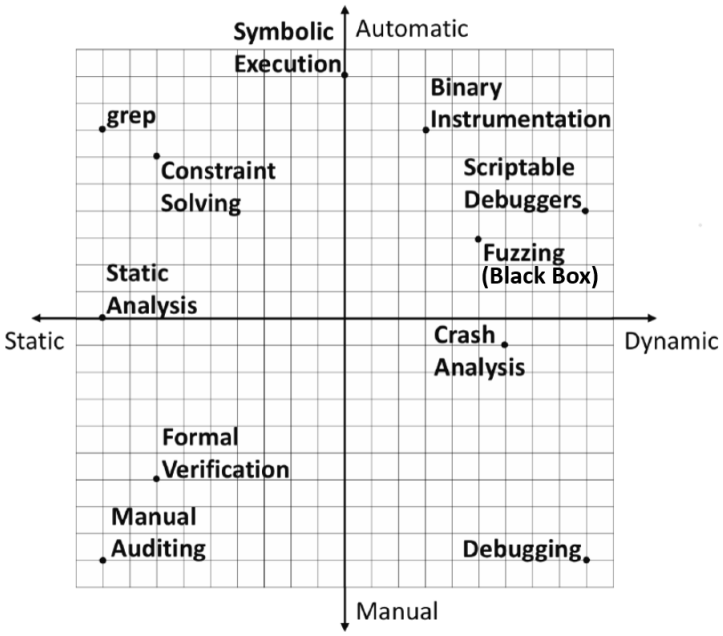

Figure 2.4: Graphic showing the different kinds of program analysis methods [19], based on two axis from static to dynamic and from manual to automatic.

2.4.1 Model Checking

Model Checking is part of Formal Verification[20]. It checks several properties of a system by verifying whether each of all executions satisfies a formal specification based on:

- Safety : Something bad will never happen.
- Liveness : Something good will happen.
- Fairness : If something is required infinitely often, it will be successful each time.

If the program does not behave as expected, the Model Checker will return a feedback of the unexpected behaviour generated with a counterexample. Soundness and completeness of Model Checkers are very important. A sound Model Checker will not produce any false negative results unlike complete Model Checkers which avoid false positives but can miss unexpected behaviours.

Model-Checking consists in converting a system into a Timed Automaton(TA) which is a Finite State Machine extended with a clock. This TA is then provided to the Model-Checker which will verify that a formal specification holds on each possible execution of the system (see figure 2.5). This specification of the system

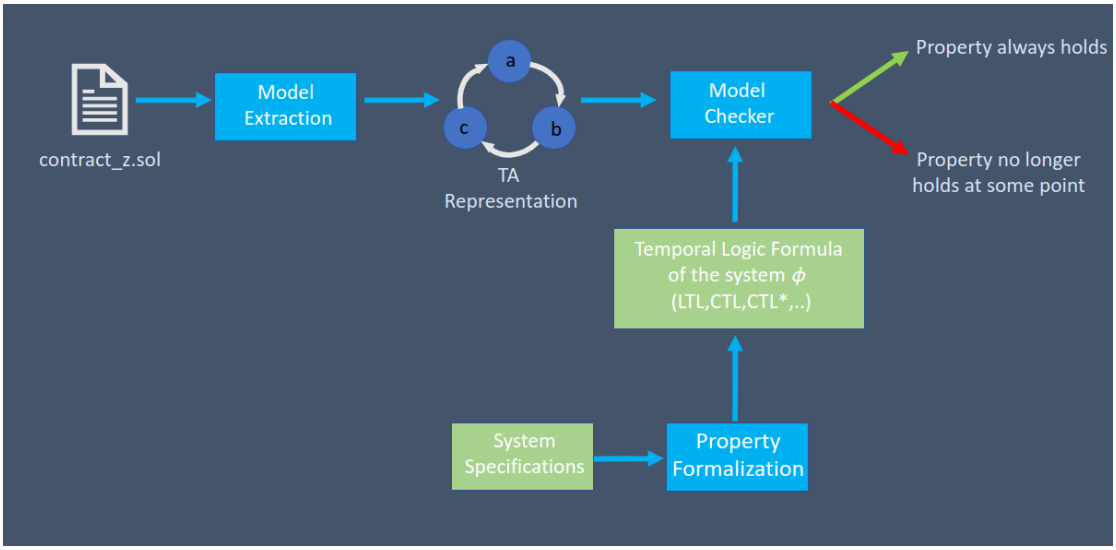

Figure 2.5: Model Checker

could either be : " My program will not crash" or "My program will eventually finish" and can be formalized with temporal logic formulas[21], depending the context, such as :

- Linear Temporal Logic(LTL)[22] - The syntax of LTL formula is given by

$$\varphi ::= True \mid False \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid X\varphi \mid \varphi_1 \; U \; \varphi_2 \mid G\varphi \mid F\varphi \quad (2.1)$$

  With a as an atomic proposition and temporal operators as described in fig 2.6. LTL is used to specify linear time properties and reasons over each linear path (i.e sequence of states) of a program in order to figure out if an event will happen eventually.

- Computation Tree Logic(CTL)[23] - The most common syntax of CTL formula is given by :

$$\begin{aligned} \varphi ::= True \mid False \mid a \mid \varphi_1 \ \wedge \ \varphi_2 \mid \neg\varphi \mid X\varphi \mid \varphi_1 U \varphi_2 \mid \\ AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A[\varphi_1 U \varphi_2] \mid E[\varphi_1 U \varphi_2] \end{aligned} \tag{2.2}$$

  With quantifier over paths also described in fig 2.6.
  On the other hand, CTL describes a branching time logic(i.e structured as a tree). It checks properties by verifying all possible simultaneous executions of a program starting from a state, whether the initial conditions are satisfied.

- CTL* : Superset containing both CTL and LTL because some LTL formulas can't be formalized into CTL formulas and vice versa.

- ..

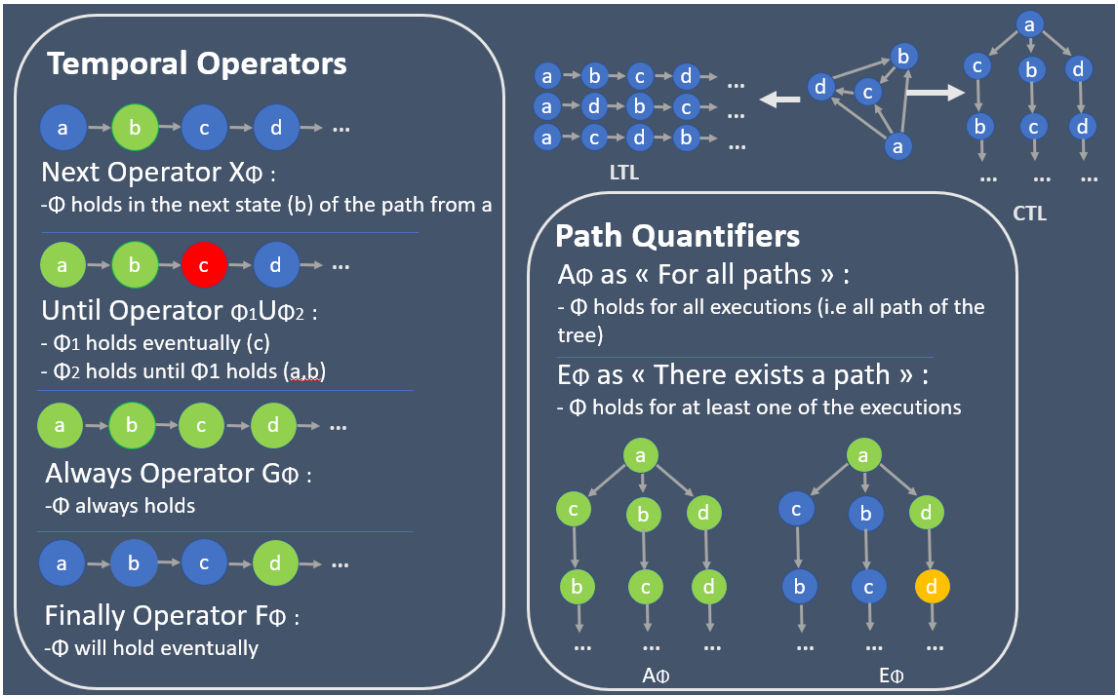

Figure 2.6: Linear Temporal Logic(LTL) and Computation Tree Logic(CTL) operators and path quantifiers

We are now able to explain how basic model checking is performed for LTL and CTL :

LTL Model Checking

Concerning LTL Model Checking, from the negation of the LTL formula derived, a Non-Deterministic Büchi Automaton (NBA)[24] $Aut_{\neg\varphi}$ is constructed. $Aut_{\neg\varphi}$ will only accept linear paths which satisfy $\neg\varphi$. If a linear path from the checked system does satisfy $\neg\varphi$, the checker can conclude that the property is eventually not held during the execution of the system. The LTL Model Checker will return the counterexample in which the property is not satisfied.

CTL Model Checking

Alternatively, since it is state based, a CTL Model Checker will try to solve recursively the Boolean satisfiability problem SAT(φ) : the set of states from the system satisfying the CTL formula φ. Then it will check whether any initial state satisfies SAT(φ). Finally, if a state of the system does not satisfy φ, a path from the initial state will be reported from the CTL Model Checker.

Various kinds of Model Checkers

In the past, Model Checkers were suffering from State-space explosion therefore they only were able to check small programs. Nowadays, many improvements allow to verify more elaborated programs. For instance, there are much more kinds of Model Checkers but we'll describe concisely these two :

- Statistical Model Checkers (SMC)([25],[26]) : Some systems often behave in a stochastic manner such data inputs, failures, or delays with probabilistic distributions may occur during the execution of those systems. Statistical Model Checkers usually work such that systems are simulated and specifications are whether validated by hypothesis testing. A SMC is in order to prove with a statistical evidence that a property will hold during the whole execution. This simulation method reduces drastically the state and time spaces required and allows to model check bigger systems, with a bounded probability of making an error.
  UPPAAL SMC[27] provides such a feature.

- Model Checking based on Satisfiability Modulo Theories (SMT) : SMT are a generalization of SAT. The purpose of SMT is to allow verification in first order logic because systems are not usually written at Boolean level. It provides additional theories such as theory of equality and theory of integers/reals

in order to reasons about arithmetic operations, arrays, several data-types (lists, trees, enumerations,etc). It works by negating the FOL formula derived from the constraints specified and trying to find if this negated formula is satisfiable by a SMT Solver(i.e The constraint is violated eventually). Z3 Theorem prover[28] is a SMT based solver which was developed by Microsoft Research and often used in SMT Model Checking. Its goal is to convert SMT formulas into Boolean level logic formulas and then pass them to a SAT Solver which will verify the satisfiability of the negated initial formulas, as it is performed for LTL formulas.

2.4.2 Static Analysis

Static Analysis[29] Programs(SAP) verify programs statically such that they are not executed. Usually, SAP take the source code or a compiled form of the program under test and convert it into an evaluable intermediate representation. This intermediate representation is passed through an adapted analysis tool which usually checks for particular properties or patterns indicating errors, syntax violations, vulnerabilities, etc. Unfortunately, Static Analysis may produce false positive results.

There are several types of static analysis methods : Control analysis which focuses on the control-flow. Data analysis ensures the correctness and the well usage of the data. Failure analysis inspects vulnerabilities or errors within codes, etc.

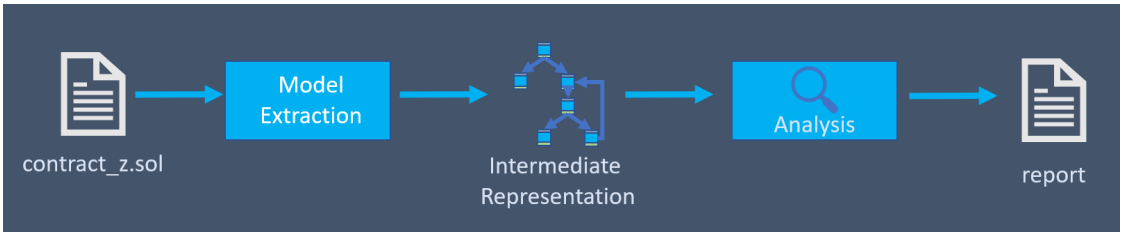

Figure 2.7: Static Analysis functioning

2.4.3 Fuzzing

In 1990, B.Miller, L.Fredriksen and B.So introduced the concept of fuzzing[30]. It is defined by an automated testing technique covering boundary cases and by passing invalid data inputs to applications, in order to ensure the absence of exploitable vulnerabilities. Fuzzing is also compared to Brute-Force testing. They developed

"Fuzz" which was generating streams of random characters passed as arguments on several utility programs. Fuzz was able to crash 24% of them.
Nowadays, programs and applications are far more sophisticated than in the 90's. Therefore, fuzzers had to improve by focusing on several points such as relevance of the data generation and the target monitoring which were random in the past. This is now a mainstream practice in assessing software security. 3 major flavors of fuzzing[31] are proposed :

Blackbox Fuzzing

Blackbox Fuzzing is the simplest way to fuzz programs because it doesn't require any information about them. The only requirement is an input generation strategy which will target specific vulnerabilities and reduce state-space/ time-space. The first way to generate inputs is by starting from a grammar, a model or a dictionary. The second way consists in using mutation to generate new inputs from previous ones. Then the blackbox fuzzer will try to break the program under test and report which inputs generated crashes.

Whitebox Fuzzing

In opposition with Blackbox Fuzzing, Whitebox Fuzzing has all the knowledge about the program or system and uses model-based testing techniques such that it can be more accurate concerning the inputs generation. Inputs are generated such that the program under test takes different paths of the control flow graph at each execution. The goal is to repeat this process until all feasible execution paths are discovered and reported if one of them leads to a vulnerability targeted within the specifications. The biggest difference between whitebox fuzzing and blackbox fuzzing is that blackbox fuzzing is used to discover ways to break programs and takes usually less time to generate inputs, unlike whitebox fuzzing which takes a longer time but ensure better results in terms of states discovered.

Greybox Fuzzing

Greybox Fuzzing is a technique requiring a knowledge about the programs being tested. It consists in a tradeoff between blackbox Fuzzing and whitebox Fuzzing techniques, in order to reduce the time complexity.

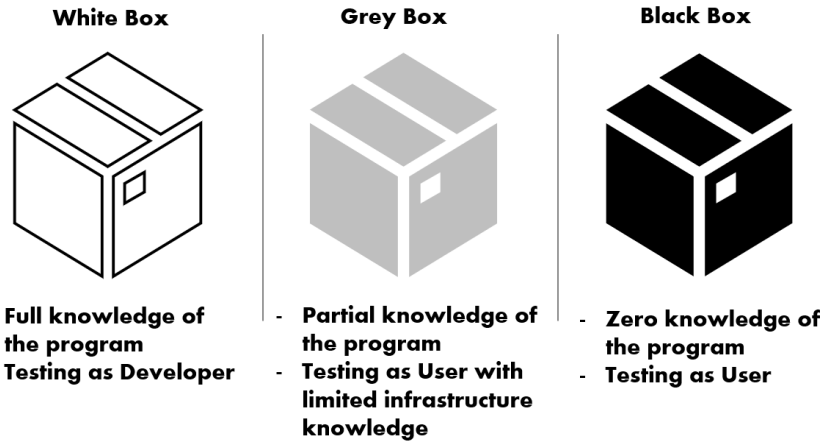

Figure 2.8: Fuzz testing comparison

2.5 Wablieft project

In order to compare those different analysis techniques, we used the Wablieft Project([32] as test contracts. Wablieft aims to improve medical services through a shared marketplace for service providers. The current organization leads to deficient distribution of the workload. Some services (such as medical equipment, specific doctors, free rooms, etc) are usually under-used or over-used due to a lack of organization. The goal of Wablieft consists in optimizing this challenging task by pooling the information related to services and allowing the patients to directly select the most profitable ones, depending on the location, the price or even the timing. All parties involved could benefit from this shared marketplace, thanks to the improved organization. Hospitals would redirect patients towards this marketplace by giving them an unique coupon as a result to avoid administration tasks, patients would have a better flexibility to fit their planning and their budget. Finally, medical service providers would have a better workload management.

At the beginning of the project, Wablieft was formally modelled using Uppaal SMC[27] in order to prove its correctness. The Uppaal SMC model shown in fig 2.9 is composed of nodes representing states of the system and edges representing actions modifying those states.

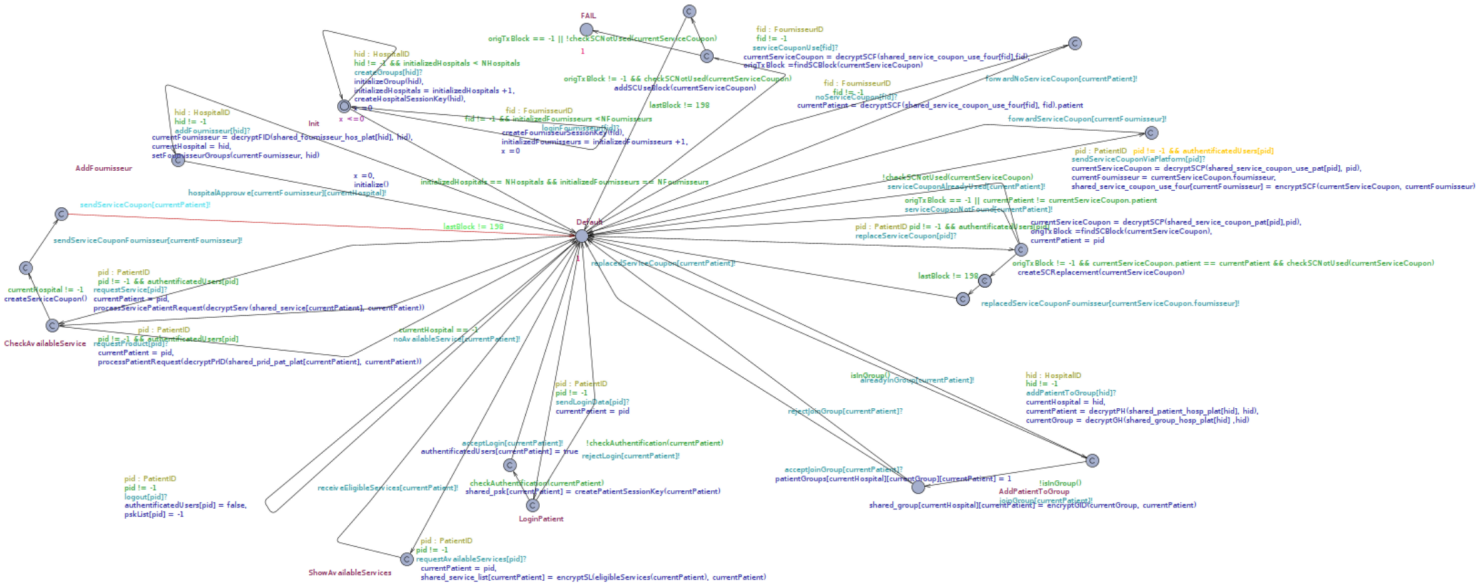

Figure 2.9: Uppaal SMC model of Wablieft Marketplace taken from [32]

The next step was to find a system allowing to keep track of each coupon given to patients. The Blockchain technology fits quite well this specification since it provides an immutable public ledger of all transactions involving coupons. Additionally, this would allow to verify the origin of each coupon produced and reduce the risks of falsification (reused, revoked or false coupons). The second advantage of using Blockchain technology is the ability to audit the shared marketplace easily in order to ensure its proper behaviour. Therefore, this model has been developed through Smart Contracts running on the Ethereum Blockchain.

In order to compare the different analysis techniques, we received several Solidity versions of Wablieft. These tests will be described in further sections. Thanks to our collaboration with Lionel Metongnon, we had the opportunity to discover several vulnerabilities within the project and then report a feedback in order to get rid of those vulnerabilities.

Chapter 3

Related Works

Due to the current global attention paid to blockchain, many people are digging into the subject of Smart Contract vulnerabilities. A lot of frameworks have been proposed for Smart Contracts analysis. This is why we chose to select several tools, compare them and select the most efficient in each category (i.e Fuzzer, Static Analysis and Model-Checking), with an Open-Source licence in order to enhance it if possible.
Here, we'll describe what tools we discovered and will give a quick overview of them.

- ContractFuzzer[33] : It was the very first framework for detecting vulnerabilities of Smart Contracts, on Ethereum Platform. This fuzzer is based on static and dynamic analysis implying test-oracles which verifies specific types of inputs, depending on which vulnerability is tested (i.e timestamp dependencies, reentrancy,block number dependencies,etc). It is Open-Source and the last update is from 2020.

- Oyente[34] : Oyente uses symbolic execution. Symbolic execution consists in representing each input's variable by a symbolic input value. Each path of the program execution is verified statically under specific conditions. With this technique, we are able to figure out how a program should respond under specific inputs without the need of a dynamic run, which would require to simulate an execution environment. This allows us to figure out whether some inputs may produce vulnerabilities or not. For instance, in order to verify the consistency of interleaved transactions with a contract, this method allows us to explore all different execution paths, one by one, of interleaved function-calls instead of having to deal with the non-determinism and the complexity of Ethereum Blockchain's behaviour.

- Securify2[35] : Securify2 is currently under enhancement and supported by the

Ethereum Foundation. It performs a static analysis on contracts and checks for property violations. It can guarantee that certain behaviours are whether safe or unsafe and reduce significantly false positive results. Securify2 showed better performance in term of vulnerabilities discovered than ContractFuzzer and is also Open-Source.

- SmartCheck[36] : Smart check is an efficient tool for Smart Contract analysis but only analyses flat contracts (i.e contracts with no import statement). Functions involving imported contracts are simply ignored by the fuzzer.

- Fuse[37] :The contract is passed to a vulnerability test scenario. Each function is decomposed in order to be checked. Then, the scenario encoding module decides which vulnerability to explore and generates all the relevant inputs, in order to prevent path explosion. Finally, a report is generated. It hasn't been updated since 2019.

- Manticore[38] : Manticore also uses symbolic execution in order to analyse binaries and Smart Contracts. In addition, it guarantees no much false positive results. It's currently under development and Open-Source.

- GasFuzzer[39] : It's directly based on ContractFuzzer but takes the amount of gas into account in order to minimize its cost.

- Zeus[40] : Zeus is clearly one of the best actual Model Checker for Smart Contracts. It performs an automatic formal verification on the high level language (i.e solidity files) It automatically translate contracts in a formatted version in which time related properties can be checked.

- Veri-Solid[41] : Veri-Solid is a very efficient tool for Model-Checking on Smart Contracts. It allows the user to generates a contract from an Finite-State-Machine Representation. Additionally, It takes into account time dependencies in order to test time related properties. Since it only does FSM to contract and not the opposite, we didn't choose to explore this tool.

- SMT Checker - Solidity's Model Checker([42],[43],[44]) : It's a feature of Solidity's compiler. It does not support all solidity features yet therefore the analyse is not complete. On the other hand, SMT checker is known to be sound since it displays the keyword *'might'* when it treats one of these unsupported features. It's based on Satisfiability Modulo Theories[45]. SMT-checker tries to validate the 'require' or 'assert' statements, by proving that these conditions are always respected. It also checks overflow and underflow errors, trivial conditions and unreachable code.

Now that several tools were proposed, we would like to classify them in order to select and improve efficient ones. We discovered an interesting work[46] comparing the efficiency of those tools. Fig 3.1 presents an array showing which vulnerabilities (list of them right below) were handled by Smart Contracts analyzers. We will be describe them later, in Chapter .

Name	**Vulnerabilities**						**Report month**
	RE	UE	LE	TO	IO	UA	
Oyente	✓	✓		✓	✓		2016-10
ZEUS	✓	✓	✓	✓	✓		2018-02
Maian			✓			✓	2018-03
SmartCheck	✓	✓	✓		✓		2018-05
Securify	✓	✓	✓	✓		✓	2018-06
ContractFuzzer	✓	✓					2018-09
teEther						✓	2018-08
Vandal	✓	✓					2018-09
MadMax			✓		✓		2018-10

Figure 3.1: Tools comparison from [46]

- reentrancy(RE) [43]
- Unhandled Exceptions(UE)
- Locked Ether(LE)
- Transaction Order Dependency(TO)
- Integer Overflow(IO)
- Unrestricted Action(UA)

Thanks to these comparisons, it will help us to select relevant tools in each category. Our choices will be described in further sections.

Chapter 4

Description of Smart Contract Vulnerabilities

Deploying a Smart Contract requires a lot of verification because mistakes may be costly for unconscious users. Once a Smart Contract has been deployed within the blockchain, it cannot be modified and everyone can interact with it, since it's public. This is why security of Smart Contracts is a key. In this chapter, we'll go through an non-exhaustive list of most important Smart Contract vulnerabilities. Fortunately, these known vulnerabilities are usually verified by several tools.

4.1 Reentrancy

Smart Contracts have the ability to call external Smart Contract functions. This ability may cause some serious vulnerabilities since not all external contracts are safe. Even worse, some are malicious and will try to abuse of this vulnerability. For instance, an attacker could exploit it by creating a contract B which will call the withdraw function of vulnerable contract A (see figure 4.1). This withdraw function has in theory the role of being able to send money from A to B, if B's account is positive. But when sending ether from A to B, if no receive function is defined, then the fallback function is executed. Fallback function is an unique and unnamed function which is executed in the case of the function called by an external party (user or contract) does not match the specifications of any function within the contract.

In our example, the attacker could add a statement within the fallback function which will call again the withdraw of contract A. In this way, the attacker is able to steal the total amount of ethers inside the contract A's balance. The vulnerability arises because the line that updates the account of malicious contract

B arrives only after the transfer to B is done. If the withdraw function is called back by B directly after sending the ether, the value of B's account will not yet be updated, and therefore the function can be called back. There are two solutions to this kind of vulnerability:

- Make sure all internal state changes are performed before the call is executed. For example make the update of the balance before sending the ethers.
- Use a reentrancy lock, to avoid the possibility of having two calls on the same function at the same time.

Reentrancy is an important vulnerability for Smart Contracts. The most known hack was the DAO hack[6], explained in the Introduction, which cost approximately 5.3 million of Ether's with a net worth of 150 million dollars at the time (14% of the total Ether's running in the market). Fortunately, the victims were able to recover their funds by a hard fork[47] of the ethereum blockchain, causing a roll back of all the transactions during the hack.

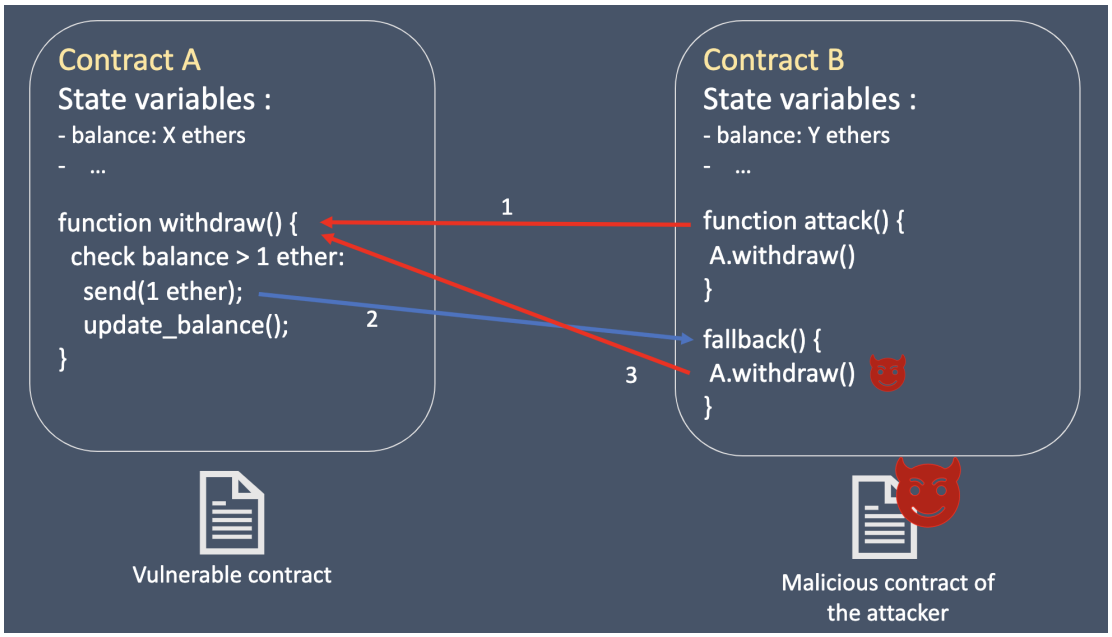

Figure 4.1: Reentrancy vulnerability

4.2 Arithmetic Over/Underflow

An arithmetic over/underflow occurs when an arithmetic operation involving fixed-size variables tries to store an out-of-range value in one of them. This is because EVM specifies fixed-size data types for integers which are represented over a specific range of values (e.g. uint8 : [0,255], uint16 : [0:65535], int,...).
A way to illustrate how to exploit an underflow vulnerability, is by trying to subtract 1 from an 8 bit unsigned integer (uint8) whose value is assigned to 0. It will result in storing 255. It works the same way for integer overflow : we try to add 1 to an uint8 assigned to 255. The fresh stored value will be 0. This example also works for signed variables. Let's take a real-case example (figure 4.2):

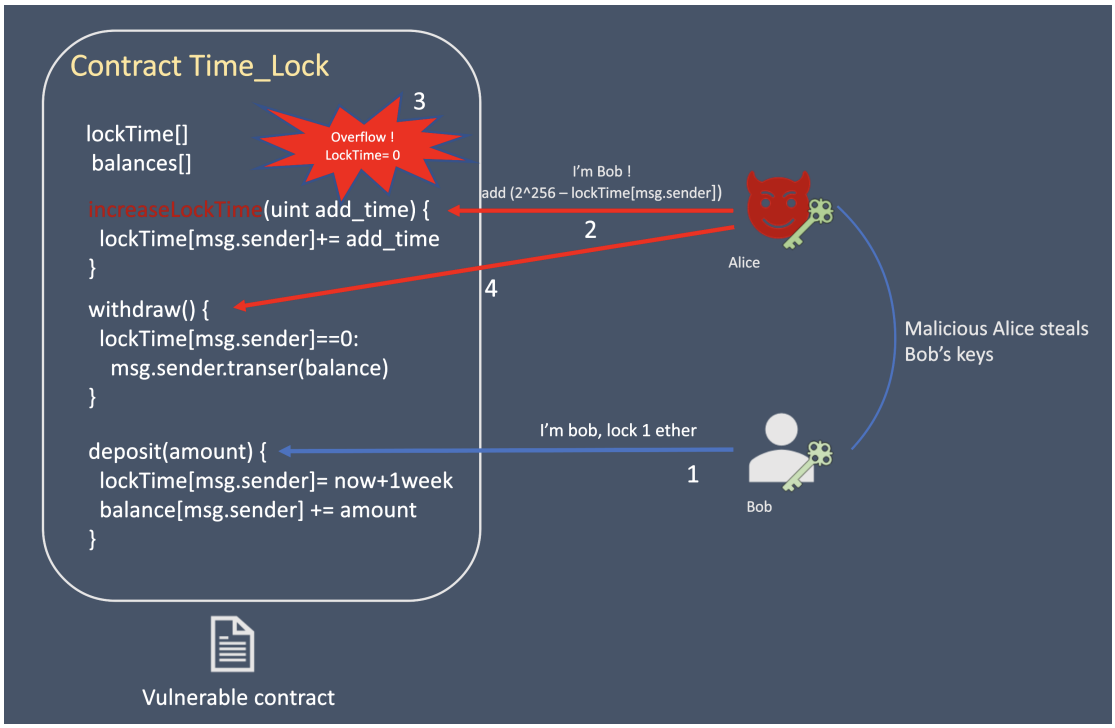

Figure 4.2: Arithmetic overflow vulnerability: The time_lock contract acts as safe deposit box. Users can deposit their ethers and specify a duration for their coins to be locked in the contract. This should prevent malicious users to steal their eithers, even if they have the secret keys of the user. The only action a user can do is extending the locking period. The vulnerability here is that the duration is stored in an unsigned integer and the function involving this variable does not prevent from integer overflow. Alice, who is malicious, could manage to steal the keys of Bob (i.e she will be recognised as Bob) and then simply call IncreaseLockTime(2^{256}-RemainingTime) in order to unlock and steal Bob's money.

A simple way to mitigate this vulnerability is by adding require statements within the code. These statements specify some conditions about arguments passed to functions.

4.3 Unchecked return values

When a Smart Contract call returns an error, all state modifications which were supposed to be pushed towards the Ethereum blockchain are reverted. As well for the state modifications generated by the other contracts involved in the failed transaction. This ensures the atomicity of transactions, meaning they either complete successfully or have no effect on whole state of the blockchain.

However, a problem could occur when a contract is not aware of a failed external contract call, which has not thrown any error. In Solidity, several functions (such as the *tranfer* function) return an error when they fail. Alternatively, other functions such as *call* and *send* functions return a Boolean indicating whether the call succeeded or failed. If this boolean is not correctly handled, the execution could be considered as succeeded and will modify the blockchain state, leading to unexpected results. This vulnerability is shown in the below code:

```
1  contract Lottery {
2      bool public cashedOut = false;
3      address public winner;
4      uint public prizeAmount;
5
6      function sendToWinner() public {
7          require(!cashedOut);
8          winner.send(prizeAmount);
9          cashedOut =true;
10     }
11
12     function withdrawUnlocked() public {
13         require(cashedOut);
14         msg.sender.send(this.balance);
15     }
16 }
```

This contract represents a Lottery-like contract, in which the winner receives the prizeAmount in ether. As a result, the contract balance is emptied. The vulnerability is located at line 8, where a send function is performed without checking the response. Even if the transfer to the winner fails intentionally by a malicious action or unintentionally by a lack of gas, the return value of send is not checked. This sets the value of cashedOut to true regardless of whether ether was sent or not. After that, the withdrawUnlocked is not protected anymore, and

anyone can withdraw the winner's prize via the withdrawUnlocked function.

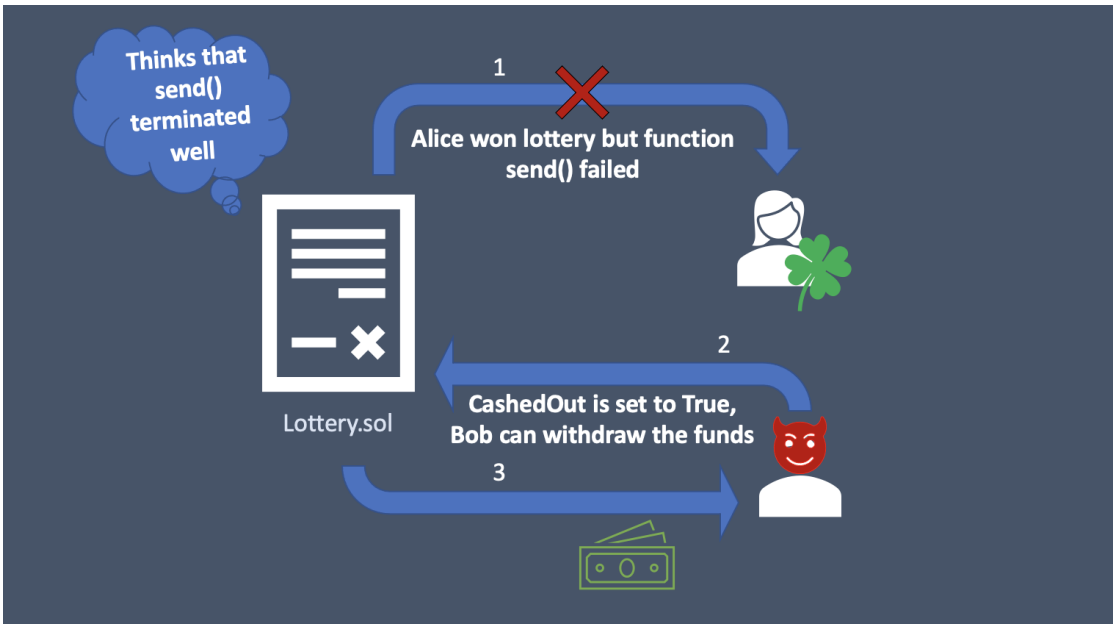

Figure 4.3: Alice won the lottery but Lottery Contract failed to send the rewards without being aware of the failure. Bob can exploit this vulnerability to withdraw the rewards of Alice

There are two possible solutions to this problem: either use the transfer function for sending ether, as this function returns an error if it fails and the transaction is reverted. Or use a require like the following line, which would stop the code execution because the return value of the send function would false if the transfer fails, and the transaction would be reverted too.

```
require(winner.send(prizeAmount));
```

Chapter 5

Model-Checking : Solidity SMT Checker

The Model Checker selected is the SMT Checker[44] feature of Solidity compiler. It is still under an active development and not enabled by default but offers many advantages that will be described further.
As described in the Background chapter, a SMT Model Checker takes, as input, a First Order Logic Formula describing a security property that the programmer wants to verify and determines whether this property holds during each execution. This method fits quite well the specifications of Solidity functions which have *assert* and *statements*, also written in First Order Logic. At first, it considers require statements as assumptions and tries to prove that the conditions within the assert statements, in addition with several correctness properties, are true for each possible execution. The best way to prove that a property is holding every time is by proving that the contradiction this property occurs at least once. If the property is not holding every time, a counterexample leading the program to violate these constraints is provided.

Solidity's SMT checker(SSC) works by the combination of two engines : Bounded Model Checking(BMC) and a Constrained Horn Clauses(CHC) systems solver. They have different characteristics and are independents from each other, except the fact that CHC is run first, then if BMC finds a vulnerability not discovered by CHC, it is reported. It is sound but not complete since it does not support yet all Solidity's features.

In addition of the properties specified by the assert statements, it also automatically verifies several properties at compilation time :

- Arithmetic over/underflows
- Insufficient funds on transfer
- Division by zero
- Out of bound index access of arrays
- Trivial conditions and unreachable code

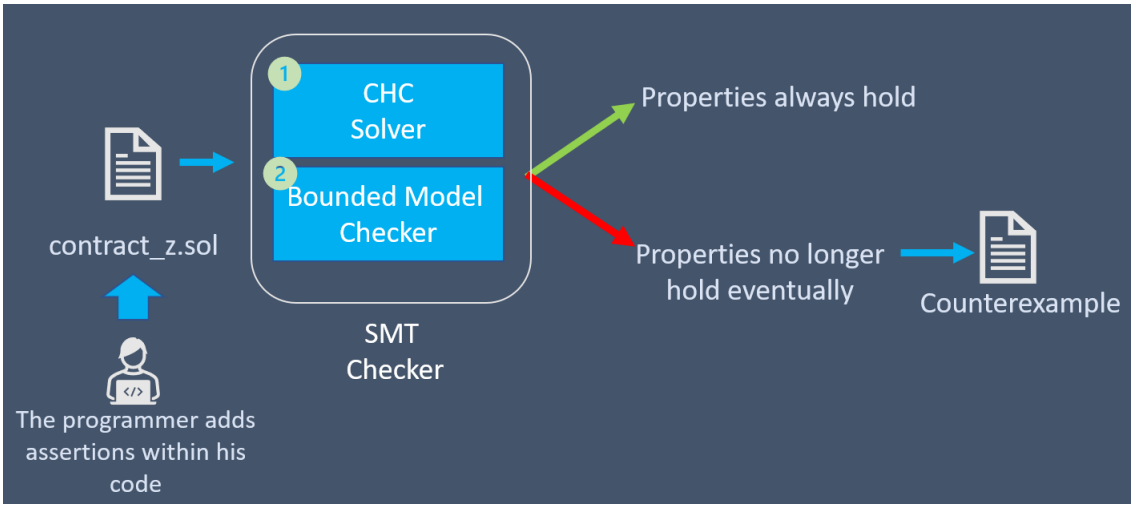

Figure 5.1: Overview of SMT checker. CHC is run first and BMC adds an additional search. An output containing counterexamples is provided.

An example of arithmetic overflow is shown in the next code. It occurs because add()'s output is limited to 256 (uint8). In the add function, an overflow may occur if a1+b1 is higher than 256. This is reported by the Model Checker in fig 5.2.

```
contract Overflow_test{

    function add(uint8 a1, uint8 b1) public pure returns (uint8) {
        return a1 + b1;
    }
}
```

```
Warning: CHC: Overflow (resulting value larger than 255) happens here.
Counterexample:

a1 = 1
b1 = 255
 = 0
```

Figure 5.2: Solidity contract showing an example of arithmetic overflow.

5.1 Bounded Model Checking

The Solidity Bounded Model Checker([48]) uses Z3 solver([28]) which is briefly described in the backgrounds to perform SMT model checking on Smart Contracts. The adjective "Bounded" comes from the limited amount of iterations through loops/recursive calls within functions that BMC verifies, in order to avoid state-space explosion. Developers chose to only go through 1 iteration, as a result to just touch the variables within this iteration. This obviously leads to several false positive warning displayed by BMC. BMC verifies each function by isolation, one by one. It starts deriving SMT constraints to be checked by running through the abstract syntax tree of the contract execution, in a depth first traversal. The state and context of the contract are reset each time a new function is verified.

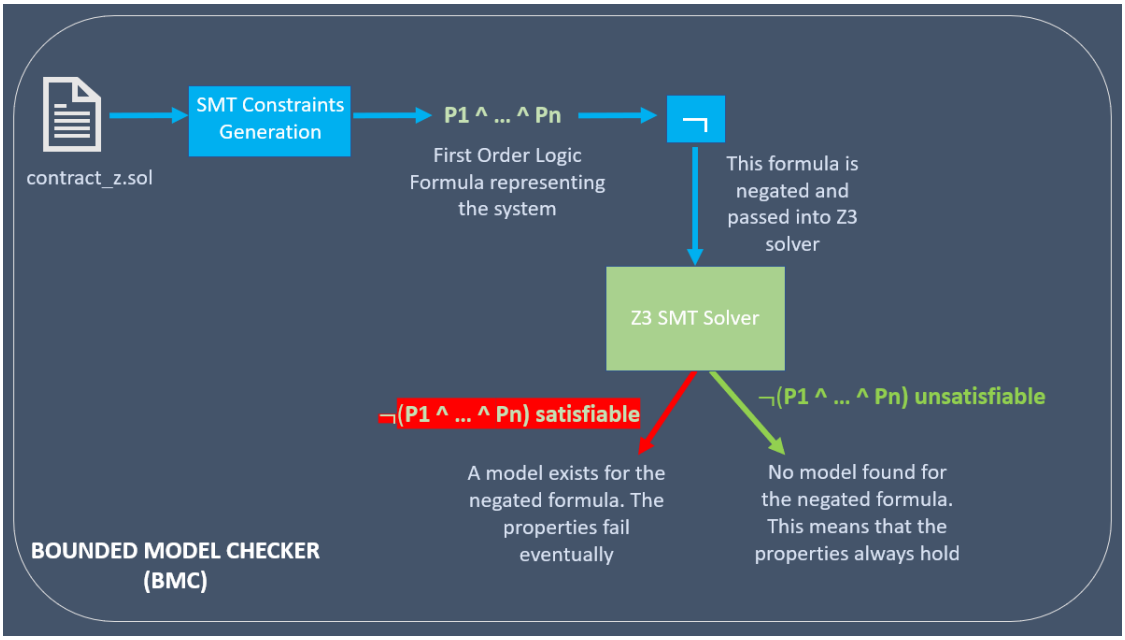

Figure 5.3: Bounded Model Checker. SMT constraints are generated from the source, negated and then passed through Z3 SMT solver. A counterexample is output if the properties checked doesn't hold

5.1.1 SMT constraints

BMC derives 5 types of constraints during the depth-first traversal of the AST.

Branch Conditions

Branch conditions are conjunctions of conditions that must hold in the current state of the program : $(c_1 \ \wedge \ ... \ \wedge c_n)$
Each time an "**IF** $(cond)$ **THEN** X **ELSE** Y " statement is encountered:
cond is added to the branch conditions during the execution of X. Then it is replaced by $\neg cond$ during the execution of Y. Both of them are removed after exiting the If-Else Statement.

- **THEN** branch : $(c_1 \wedge ... \wedge c_n \wedge cond)$
- **ELSE** branch : $(c_1 \ \wedge \ ... \ \wedge c_n \wedge \ \neg cond)$

Control-Flow

These formulas are derived from the "assert" and "require" statements. Require statements act as filters for symbolic variables generated by BMC. On the other hand, the conditions within assert statements represent the negation of post-conditions specified by the programmer that should be fulfilled during the execution of the contract, which are verified statically. Require statements also allow the programmer to verify which property he wants to model-check, in particular safety and correctness.
Depending on which of both appears, the new Branching condition derived becomes:

- **require**(a) : $branching\_cond \Rightarrow a$
- **assert**(a) : $branching\_cond \Rightarrow a$

Type Constraint

: variables passed as input parameters are recognised by the program and initialised to a specific range depending their type. For instance : uint8 x is initialised as $0 \leq x \leq 2^8$; int y as $0 \leq y \leq 2^{32}$ and 20 bytes address $addr$ as $0 \leq addr \leq 2^{20*8bits}$.

Variable Assignment

A Variable assignment is encoded in SMT following a Static Single Assignment(SSA). SSA form requires that each time a variable **x** is re-assigned, a new SMT variable x_i is created as a new version (i.e **i** is incremented each time **x** is reassigned). SSA simplifies the analysis by always taking the last updated variable into account. In

addition, when a variable is updated in several branches, BMC uses an if-then-else function ite(c,x_1,x_2) with

- c : the current branch condition
- x_1 : the variable corresponding to the end of the **THEN** branch
- x_2 : the variable corresponding to the end of the **ELSE** branch

Fig 5.4 shows an usage of ite function. Variable x is initialised to 0 and is updated to 10 or 50 depending whether y is assigned to 0.
According to SSA, var x is defined as an SMT variable named $x_0 = 0$. During the execution, x could be re-assigned by two distinct values, therefore two new SMT variables are generated : $x_1 = 10$ and $x_2 = 50$. The SMT solver will choose, after the If-Then-Else statement, by which of x_1 or x_2, the new SMT variable x_3 will be assigned by computing $x_3 = ite((y == 0), x_1, x_2)$.

```
x=0
if (y == 0):
    x = 10;
else
    x = 50;
```

Figure 5.4: If-Then-Else example with variable assignment

Verification Target

Verification target is the formula aggregating the previous derived constraints, including current branch conditions. This final formula is passed to the Z3 SMT solver in order to check its satisfiability.

5.1.2 SMT constraints extraction example

Fig 5.5 shows an example of how SMT constraints are extracted.
Initially, input parameters uint8 x and uint8 y are assigned to symbolic variables between 0 and 255 included. Then the *require* statement is implied whether the condition (x==10) is taken. We observe that y has 3 possible new assignments. They are formulated at the 5'th line (on the left side of the figure). A new SMT variable y_4 is created with the value 30 or 20 depending of whether (x==20), thanks to ite function. Similarly, x_5 is created. The final formula initially looking like : $B_c \rightarrow (y_5 \leq 100)$(assert statement) is negated before being passed through the SMT Solver and becomes $\neg(B_c \rightarrow (y_5 \leq 100))$ which is equal to $B_c \wedge \neg(y_5 \leq 100)$,

the left side FOL formula.
This is a way to represent constraints in first order logic. This formula can finally be passed through the Z3 solver which will verify its satisfiability. 5.3 describes a general functionning of BMC.

```
1   contract Contract_test {
2       function verify(uint8 x, uint8 y){
3           if (x == 10){
4               require(y <= 50);
5               y= 10
6           else if (x == 20)
7               y = 20;
8           else
9               y = 30;
10          assert(y <= 100);
11      }
12  }
```

```
x0 ≥ 0 ∧ x0 < 256 ∧
y0 ≥ 0 ∧ y0 < 256 ∧
(x0 = 10) ⇒ (y0 ≤ 50)∧
y1=10 ∧ y2 = 20 ∧ y3=30 ∧
y4 = ite(x == 20, y2, y3) ∧
y5 = ite(x == 10, y1, y4) ∧
¬y5 ≤ 100
```

Figure 5.5: SMT formulas derivation from Solidity contract. The final negated constraint passed to the SMT solver is written on the left side.

5.2 Constrained Horn Clauses

To alleviate the problem of loops not explored, they combined BMC with another engine named Solicitous[49] and based on the control-flow. It consists in solving a system of Constrained Horn Clauses(CHC)([50]) in order to check its satisfiability. A clause is a disjunction of literals : $L_1 \vee L_2 \vee ... \vee L_n$. More accurately, a Horn clause is a clause containing at most one positive literal. Finally, A Constrained Horn Clause is a logical implication involving predicates and defined by :

$$\varphi \wedge p_1[X_1] \wedge ... \wedge p_n[X_{n-1}] \Rightarrow h[X_n] \tag{5.1}$$

With :

- X_i as terms (polynomial terms, linear combinations,etc)
- φ is a first order formula over a theory T[51].
- $p_0[X_0]$,..,$p_n[X_{n-1}]$ are predicates based on X_i terms and not appearing in φ.
- $h[X_n]$ is called the head and the rest is called the body. $h[X_n]$ is the implication of the body.
- An entailment having the form : $(a \Rightarrow b)$ could be expressed as $(\neg a \vee b)$ which is a clause, as described previously.

Constrained Horn Clauses are very powerful because they can represent transition systems handling loops. They are able reason about liveness and safety of those systems by discovering an inductive invariant for every loop of a Smart Contract functions. If a model satisfies the negated CHC system derived, it means that the properties specified by the programmer are not holding at any time.

At this point, we still need to generate those CHC. First, we need to model a contract function in order to represent it as a **Control Flow Graph**(CFG). Then, the set of CHC are generated from an algorithm using several rules formalizing the behaviours of the function into predicates. These predicates are used to represent thereachable states from each state of the CFG. Finally, after having computed set of CHC for each function, these formulas are negated and passed through a SMT solver (supporting horn clauses) (see fig 5.6). If one property is not holding eventually, the CHC system solver will return a counterexample which generated the assertion fail.

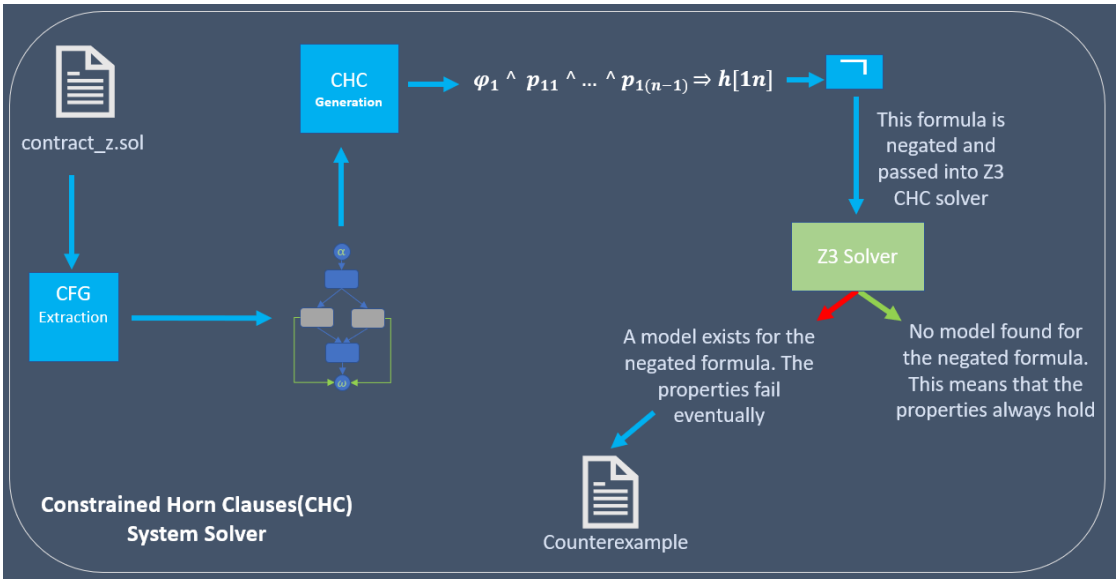

Figure 5.6: CHC System Solver. A CFG is generated from the source code, then the CHC are extracted and passed through a SMT (supporting CHC) solver. A counterexample is output if the properties checked doesn't hold

5.2.1 Control Flow Graph

In our case, a CFG is a graph representation of all execution paths of a Smart Contract function. The graph nodes represent sequences of statements not changing the control flow of the function. Edges are labelled by Boolean expressions and represent actions changing the control flow such as *loops, branching and function calls*. To jump from one CFG block to another one, the Boolean condition must be True. When one function is executed, the execution is done atomically. If an error occurred, the execution is reverted until the initial state.

5.2.2 Contract Model

We need a formal representation of a contract in order to reason about CFG of functions and finally generate the CHC. A contract is defined by the triplet $< \mathbf{s}, \mathbf{I(s)}, \mathbf{F} >$ with :

- **s** : a set of states.
- **I(s)** : the initial state.
- **F** : the set of all functions within the contract.

Similarly, the CFG of a Solidity function f $\in \mathbf{F}$: f($\mathbf{x}$) = $\mathbf{y}$, in which $\mathbf{x}$ is the set of input variables and $\mathbf{y}$ the output variables, is modeled by the tuple <$\mathbf{G}$,α,ω,ρ> with :

- $\alpha \in V$ is the entry block of the function f.
- $\omega \in V$ is the exit block of the function f.
- $\mathbf{G} = (V, E, \lambda, \mu, S)$ Represents a Directed Graph with V as the set of CFG blocks and E as edges between nodes. λ is the set of each λ_v for $v \in V$ such that λ_v contains the set of instructions performed by the CFG block v. Similarly, μ contains each μ_e for $e \in E$ such that μ_e contains the Boolean conditions required for jump e to occur between 2 blocks. S represent the Safety Blocks which contains safety properties, which could be defined by the programmer, formalized by the *assert* statements.
- $\rho : s \bigcup x \bigcup y \rightarrow l$ is *the injection* function which maps every state variable $s_i \in \mathbf{s}$, return variable $y_j \in \mathbf{y}$ and function argument $x_k \in \mathbf{x}$ to distinct local variables $l_l \in \mathbf{l}$ accessed by instructions of the function f. Local variables can be seen as registers storing temporary each value used by the functions.

Each Safety Block has an Exit edge μ_{exit} towards sthe exit block ω, containing the condition specified in the assertion triggered. If a jump through μ_{exit} from a Safety Block is triggered, the program reverts the instructions in order to restore the initial state and reports the variables that triggered the assertion fail. This is called a rollback and used to avoid states modification within contracts when an exception has occurred.

5.2.3 Rules for CHC creations

For each function f_i of a contract, the goal is to generate a set of CHC that will be passed through a SMT solver. Those clauses are generated by constructing several rules given the CFG <$\mathbf{G}$,α,ω,ρ> of $f_i(\mathbf{x})$=$\mathbf{y}$. These rules are derived following an algorithm (see appendix A.1) running through each state of each function. The resulting CHC system will be negated and passed to a solver which will verify whether a model exists for the negated CHC system. In this section, 3 important rules will be described in order to give you an idea of what a Constrained Horn Clause looks like. You can find the others within the documentation written by the developers of the engine[49] :

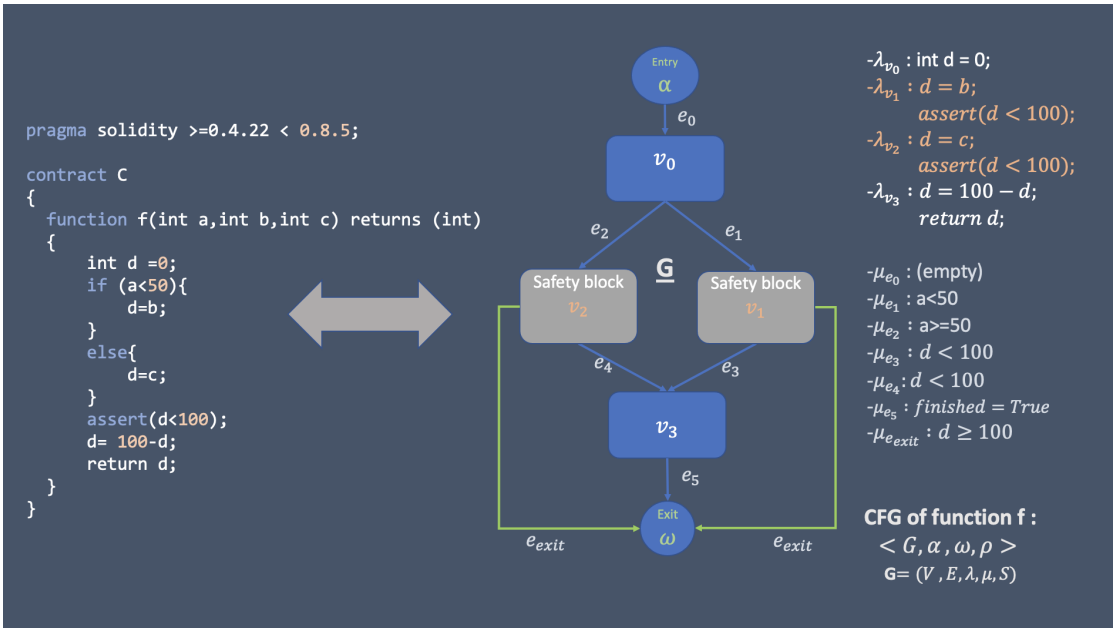

Figure 5.7: CFG representation of a solidity function

Entry Rule : $Entry_f$

The entry rule defines a First Order Logic predicate $\mathcal{P}$ constructed by conjunctions of statements. This rule models the initial state assigning the local variables($\mathbf{l}$) to the corresponding current state($\mathbf{s}$) and input($\mathbf{x}$) variables.

$$\bigwedge_{i \in \mathbf{s} \bigcup \mathbf{x}} i = \rho(i) \wedge \rho(\tilde{r}) = 0 \rightarrow \mathcal{P}_f^{\alpha}(\mathbf{s},\mathbf{x},\mathbf{l}) \tag{5.2}$$

- $\rho(\tilde{r})$ is an integer value describing whether a safety property $\tilde{r}$ holds. $\rho(\tilde{r})$ is initialised to 0. Having $\rho(\tilde{r}) \neq 0$ means that the safety property $\tilde{r}$ has failed and causes $\mathcal{P}$ to be false.
- $i = \rho(i)$ represents the update of the current state and input variables with their newest representations in the local variables.The injection function described in 5.2.2 will recover the latest version of these values from the local variables.
- $\mathcal{P}_f^{\alpha}(\mathbf{s},\mathbf{x},\mathbf{l})$: the resulting predicate representing the set of states reachable from the entry CFG block, with $\mathbf{x}$ as input arguments and $\mathbf{l}$ as local variables.

Jump Rule : $Jump_f$

Each jump $e =< a, b >$ in the CFG of function f from CFG block a to CFG block b with {a,b} $\in$ V is modeled by the rule :

$$\mathcal{P}_f^a(\mathbf{s,x,l}) \wedge SSA_{\lambda_a}(\mathbf{l,l'}) \wedge SSA_{\mu_e}(\mathbf{l}) \rightarrow \mathcal{P}_f^b(\mathbf{s,x,l'}) \tag{5.3}$$

With :

- $\mathcal{P}_f^a(\mathbf{s,x,l})$: a predicate representing the set of states reachable from a, with **x** function arguments and **l** local variables.
- $SSA_{\lambda_a}(\mathbf{l,l'})$ the Static Single Assignment (mentioned in 5.1) over local variables. This predicate models the new mapping of the set **l** to the new set **l'** $= \{x' | x \in \mathbf{l}\}$ and the behaviour of the instructions executed within *block a* into a logic formalization.
- $SSA_{\mu_e}(\mathbf{l})$ represents the logical condition for which jump e is taken

Summary Rule : Sum_f

The summary rule $\mathcal{S}_f(s, x, s', y)$ links the input and the ouput of function f. An input is represented by the sets of input variables **x** and state variables **s**. On the other hand, the outputs are represented by the set **y** and the new state variables **s'**. This CHC rule has the role can be decomposed in 4 parts.

$$\mathcal{P}_f^\omega(\mathbf{s,x,l}) \wedge \underbrace{(\rho(\tilde{r}) \neq 0 \Rightarrow \bigwedge_{i \in \mathbf{s}} i' = i)}_{revert\ constraints} \wedge \underbrace{(\rho(\tilde{r}) = 0 \Rightarrow \bigwedge_{i \in \mathbf{s}} i' = \rho(i))}_{commit\ constraints} \wedge \underbrace{\bigwedge_{i \in \mathbf{y}} i = \rho(i)}_{returns\ constraints} \rightarrow \mathcal{S}_f(\mathbf{s,x,s',y}) \tag{5.4}$$

- The *revert* constraint ensures that the execution is always reverted when a safety property $\tilde{r}$ is not holding anymore and also update the new states variables. This roll back ensures that an error will not affect the state of a contract.
- The *commit* constraint models a commit of the newest computed state values. In addition, the predicate specifies that the safety property $\rho(\tilde{r})$ is holding.
- The *return* statements fill the return values by the corresponding local variables mapped by $\rho(i)$
- A predicate $\mathcal{P}$ already described in the previous rules.

These 3 rules are required to build a set of CHC modeling a function f. Other rules exists to model a whole contract C, such as rules for internal or external function calls, safety and reachability checking rules,etc. This total set of rules allows theoretically to model any possible behaviour of a Smart Contract C.
You can find a total example produced by the developers in the paper describing the CHC system solver[49].

5.3 Tests performed by SMT Checker

5.3.1 Vulnerabilities found in Wablieft Project

In this section, we will present the vulnerabilities discoverable by the Solidity SMT Model Checker. First of all, we'll describe the vulnerabilities detected on Wablieft and provide concrete examples. Then, we'll go through other detectable vulnerabilities not present on Wablieft. For each example, we'll also give a solution in order to avoid these vulnerabilities.

Underflow

For the piece of code below, an underflow may occur. This could happen whether a uint256 variable, which is supposed to store values between 0 and 2^{256} - 1, takes a negative value. In this example, _values represents an array composing the Set structure. Having the length of _values equal to 0 when passed as an argument would imply variable lastIndex to store the value 0 - 1 = -1. Since unsigned int are only positive, the effective value stored within lastIndex will become 2^{256} - 1. A simple solution would be to check by a *require* statement that the value after the addition is greater than the value before addition.

```
1 struct Set {
2     bytes32[] _values;
3 }
4
5 function _remove(Set storage set, bytes32 value) private returns (
    bool) {
6     uint256 lastIndex = set._values.length - 1;
7     //...
8 }
```

Overflow

Concerning this other piece of code, an overflow may occur. That is, a uint256 variable which stores a value strictly greater than 2^{256} - 1. This overflow occurs when the value stored within the variable size takes a value greater than 2^{256} - 1,

since there is no restriction. A simple solution would be to check that the value after each addition.

```
mapping(uint => uint[]) private prescriptionToCoupons;

uint size = 0;

for (uint i = 0; i < patientPresc.length; ++i){
    size += prescriptionToCoupons[patientPresc[i]].length;
    //..
}
```

States Properties

In the two previous examples, we only showed the use of the Model Checker over pure code, proving properties about specific operations (additions, differences). But in Smart Contracts we can also check properties involving contract states. Assertions are really useful to check whether some security properties are holding during each possible execution and thus detect potential errors. In section 10.2, we'll describe several Wablieft properties checked through assert statements.

For instance, we consider a contract representing a Robot which can move from left to right, by updating its variable x with values between 0 and 2^{256} - 1. The robot's state position is represented by the Smart Contract below (figure 5.8).

```
contract Robot {
    uint256 x = 0;

    function moveLeft() public {
        require(x > 0);
        --x;
    }

    function moveRight() public {
        require(x < (2**256)-1);
        ++x;
    }

    function inv_Robot() public view {
        assert(x >= 0);
    }
}
```

Figure 5.8: Testing invariant x >= 0

In this contract, the function inv_Robot represents an invariant of the finite state machine which says that x must be greater or equal to 0. The Model Checker manages to prove that, regardless of how many or which function calls we make, the invariant can never fail as we excepted. Indeed, if the property had been violated in only one state, the Model Checker would have thrown an "Assertion Violation". Since this Model Checker is sound(for supported language elements)[44] and then overestimates properties, we can be sure that he will not miss any property violation.

In order to verify the correctness of the tool, we could trap it by giving it a location we know we can reach. We can specify the property saying that x will never be equal to 10, through the function reach_10 (as shown below). This property is obviously false, and the Model Checker will tell us that it has discovered a state of the execution in which the property is violated.

```
function reach_10() public view {
    assert(!(x == 10));
}
```

5.3.2 Other detectable vulnerabilities

This section will present discoverable vulnerabilities by the Solidity SMT Model Checker, but not present in Wablieft. These examples have been created to illustrate the conditions for these vulnerabilities to occur, and which solutions are available to avoid them.

Reentrancy

Mutexes are widely used to secure functions against concurrent executions. During the execution of the function **run**() from our mutex Contract, presented right below, variable x is involved. This variable can also be accessed and modified by public functions (i.e accessible to everyone). Calling an external contract between line 20 and 22 might lead to devastating behaviours since it could corrupt the value of x. This behaviour is called Reentrancy because an external unverified function can alternate the state of our contract. Solidity SMT Checker is able to verify this type of vulnerabilities by comparing variables x and xBeforeCall at the start and at the end of the function, through an assert statement (line 22).
A way to mitigate Reentrancy in this case could be the addition of a mutex, avoiding multiple concurrent executions of **run**(), as it is shown in line 19. Without this mutex, the Solidity Model Checker would have thrown an error specifying that

variable x could be corrupted.

```
interface Unknown { function attack() external; }

contract Mutex {
    uint x;
    bool lock;
    Unknown immutable unverifiedContract;

    constructor(Unknown someContract) { unverifiedContract =
        someContract; }

    modifier mutex {
            require(!lock);
            lock = true;
            _;
            lock = false;
    }

    function set(uint _x) mutex public { x = _x; }

    function run() mutex public {
            uint xBeforeCall = x;
            unverifiedContract.use();
            assert(xBeforeCall == x);
    }
}
```

Insufficient funds for ether transfer

A simple but important vulnerability verified by the Solidity SMT Model Checker but not by Securify2 is the possibility of sending an amount of ether not available in the contract. One way to get rid of this vulnerability and prevent the Model Checker from raising an alert, is to use a *require* statement that will act as a precondition to the function, in order to only perform transactions when the required amount of Ethers to be transferred is sufficient.

```
function transferEther(uint amount) public payable {
        require (address(this).balance > amount);
        payable(msg.sender).transfer(amount);
}
```

Chapter 6

Static Analysis : Securify2

Securify2([52],[53]) is a popular security verifier for Ethereum Smart Contracts. It is known to be fully automated, scalable and able to prove whether contract behaviours are safe by verifying the satisfaction of several properties. 37 known vulnerabilities (see B.1) are currently supported by Securify2. We had the occasion to improve this tool with a useful feature allowing the verification of Smart Contracts containing *import* statements, allowing a contract to import contracts from different files. This contribution (Appendix B.3), whose pull request has been accepted on the Securify2 Github repository will be described in section 9.1.

Securify2 proceeds its analysis in several steps. It firstly extracts semantic informations about each function of the contract. These semantic informations are composed of facts describing the relations between the variables, the control-flow and the data-flow in order to understand how the functions behave (see fig 6.1). Securify2 verifies vulnerabilities by formalizing them into 2 kinds of patterns such as Compliance and Violation patterns which imply the satisfaction (.resp the violation) of security properties written as First Order Logic Formulas and based on the semantic facts previously derived. As an example, in order to verify that multiple transactions are executed following the right order, Securify2 checks one Compliance and one Violation patterns involving the semantic facts of the control flow. The formalization of these patterns will be described in a further section.

Finally, a report containing all warnings and violations is provided to the user of the program. 6.2 shows an example of how functions behaviours are treated through the 2 different patterns, depending whether they are classified as safe or not. 37 vulnerabilities are listed but programmers are able to add their own properties to be verified with the cost of a deep knowledge of the behaviours they want to check.

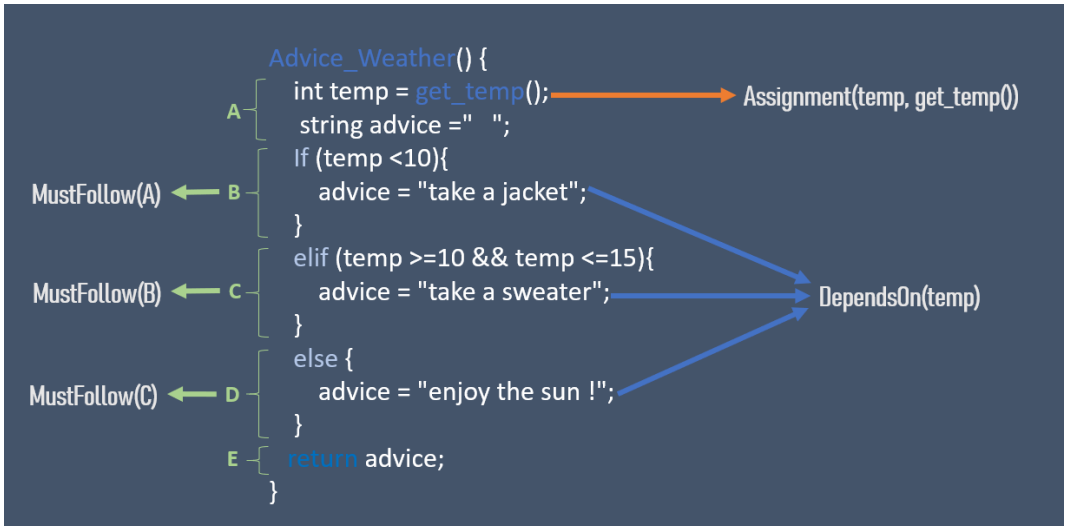

Figure 6.1: Advice_Weather helps the user of this function to choose the right clothe. These semantic facts are formalized into true predicates. *Assignment(temp,get_temp())* specifies that variable temp is assigned at this point and filled with the output of *get_temp()*. *MustFollow()* and *MayFollow()* statements are facts based on the control flow, with A,B,C,D and E, the sequence of statement blocks. Finally, *DependsOn(temp)* specifies that variable *advice* must depend on variable *temp*.

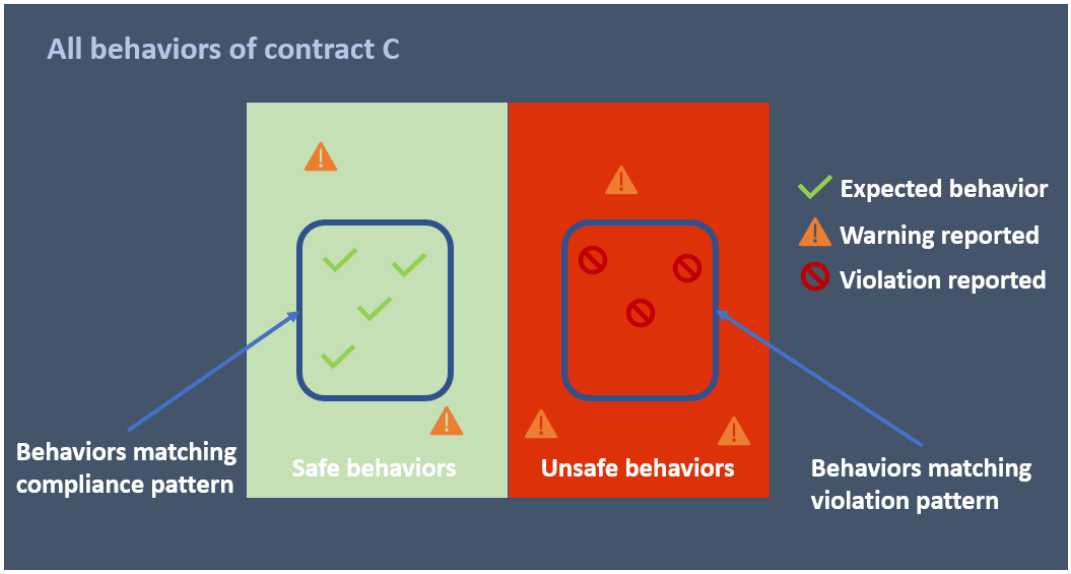

Figure 6.2: This describes how vulnerabilities are reported by Securify2. Warning are reported when a behaviour doesn't match any compliance pattern. On the other hand, A vulnerability is reported when it matches a violation pattern.

6.1 Implementation

More formally, Securify2 works in 3 steps (see fig 6.3) :

1. Intermediate Representation Generation
2. Inferring Semantic Facts
3. Checking Security Patterns (Compliance and Violation)

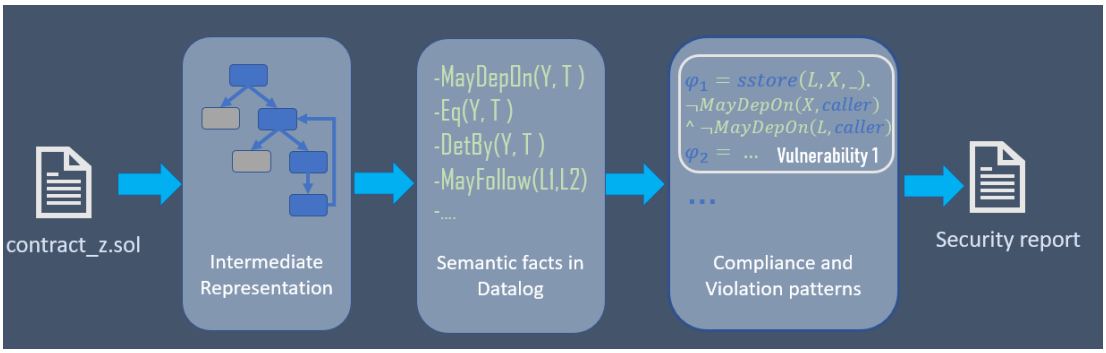

Figure 6.3: Securify2 works in 3 major steps. For each function, it extracts the intermediate representation formatted as a Control Flow Graph. This CFG allows to easily derive semantic facts about the behaviours of each function. These semantic facts are checked through several compliance and violation patterns in order to spot known vulnerabilities.

6.1.1 Intermediate Representation Generation

Basically, Securify2 recovers the Abstract Syntax Tree(AST) of the contract under test from the Solidity Compiler. This AST is converted into a control flow graph (see section 5.2.1). Each node of the CFG is represented by a block containing statements of the function. It uses Single Static Assignment (see section 5.1.1) to assign each operation result to a fresh variable. This intermediate representation is complete in a way that each Solidity Smart Contract can be represented. An example of IR is shown in figure 6.4 and a more accurate one in appendix B.1.

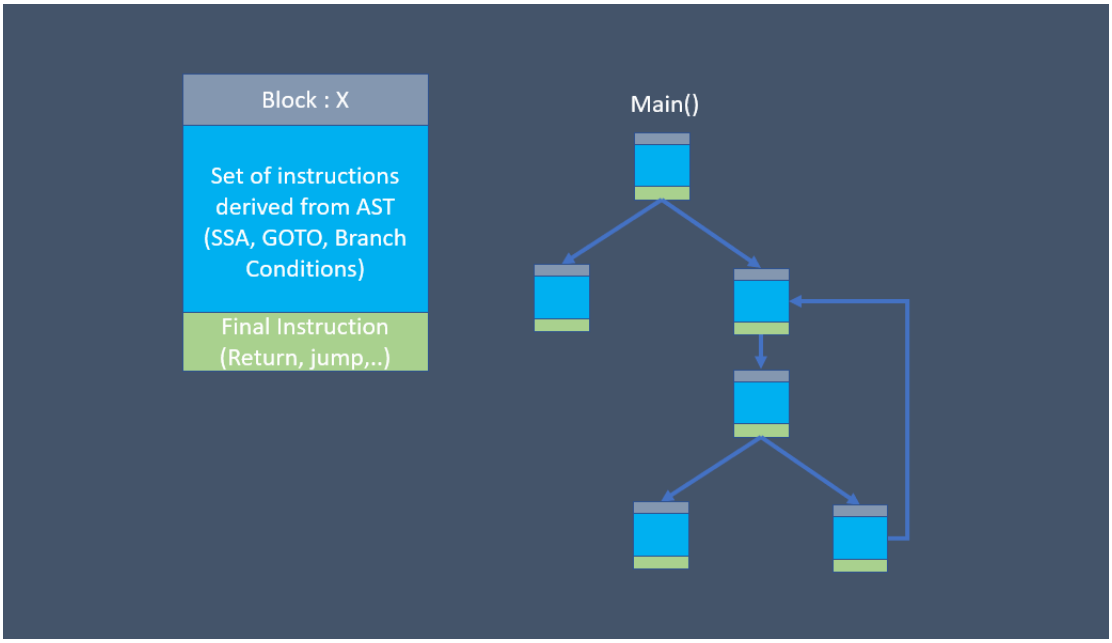

Figure 6.4: Intermediate Represensation used by the analyser of Securify2. Each block is defined by an index and contains continuous statements written following the SSA form.

6.1.2 Inferring Semantic Facts

Securify2 runs through the intermediate representation to infer the semantic facts based on data-flow and control-flow. These semantic facts are written in Datalog[54] which is a fully declarative logic programming language allowing to infer facts about function behaviours. Its syntax is defined by :

$$
\begin{array}{ll}
(Program)\ P ::= \bar{r} & (Predicates)\ p, q \in \mathcal{P} \\
(Rule)\ r ::= a \Leftarrow \bar{l} & (Term)\ t \in \mathcal{V} \bigcup \mathcal{C} \\
(Atom)\ a ::= p(\bar{t}) & (Datalog\ variables)\ X, Y \in \mathcal{V} \\
(Literal)\ l ::= a | \neg a & (Constants)\ x, y \in \mathcal{C}
\end{array}
$$

For instance, Capital(A,B) shows an example of a Datalog expression. In this case, it is a fact that city B is the capital of country A.

Securify2 derives several Datalog facts to infer about all behaviours of each contract functions :

Statement Facts

As seen in the previous section, the intermediate representation is a Control Flow Graph composed of blocks containing continuous statement of a function. Each statement is represented in Datalog by a Statement fact describing their type and the relations between the statements and the variables. As some examples, AssignFact(X, L_1, A_1) stands for a variable assignment and links variable A_1 with statement L_1 in block X. similarly for StoreFact which links variables with a location within the contract storage, LoadFact for loaded variables from the contract storage, UnaryOpFact which describes unary operations, etc.

Flow-Dependency Predicates

These Datalog facts describe the control-flow and the ordering relations between statements within blocks : Follows(L_1,L_2) , MayFollow(X, L_1,L_2) and MustFollow(X, L_1,L_2) (resp. with Precede instead of Follow, such as Precedes(X, L_1,L_2) , etc). For instance MustFollow(X, L_1,L_2) specifies that statement L_1 must follow the statement L_2 within block X. Additional facts such as goto((L_1, L_2) or stop(L) are availables.

Data-Dependency Predicates

Data-Dependency predicates describes the dependency relations between variables. For instance, MayDepOn(L,A,T) guarantees that variable A depends (i.e may be influenced) on the tag T which can either be a variable or a statement. Additionally, we have the predicate Eq(L,A,T) describing an equivalence between variable A's value and tag T. In this case, T could be the return value of the function call "sender_address()" which returns the address of the sender, and A, the actual value of the sender's address. Finally, DetBy(L,A,T) indicates that an update of tag T implies an update of variable A.

6.1.3 Checking Security Patterns (Compliance and Violation)

Securify2 developers have defined their own language for expressing security patterns composed of semantic facts previously derived and First Order Logic operators :

$$\varphi ::= \mathit{SemanticFact}(B_i, L_j, X_k) \mid \varphi_1 \ \wedge \ \varphi_2 \mid \neg\varphi \mid \exists X.\varphi \mid \exists T.\varphi \mid \exists L.\varphi \qquad (6.1)$$

With X as variables, L as statements.

Patterns allow to quantify over variables, labels, statements and the control-flow. Here are 2 vulnerabilities formalized into compliance and violation patterns:

Ether Liquidity and parity wallet hack

In 2017, 500 000 Ethers (1.5 billions $ worth in may 2021) were stuck within a Wallet Contract. A wallet contract is a Smart Contract acting as a wallet which stores ethers just as a user account. It provides additional functionalities such as a timer on the amount of stored Ethers (see example in fig 4.2). Multi-Signature wallets are similar but requires multiple approvals to withdraw the balance of the wallet-contract. Back in 2017, one of these wallet contract was using an external contract as a function library. A malicious user gained the ownership of this library contract, and erases it causing the 500 000 ethers to get frozen and unreachable from their owners.
2 compliance and Violation patterns were derived by Securify2 developers to avoid a similar vulnerability.

Property	Type	Security Pattern
LQ: Ether liquidity	*compliance*	$\mathit{all}\ \mathrm{stop}(L_1).\ \mathit{some}\ \mathrm{goto}(L_2, X, L_3).\ X = \mathrm{callvalue} \wedge \mathit{Follow}(L_2, L_4)$
	compliance	$\mathit{some}\ \mathrm{call}(L_1, \_, \_, \mathit{Amount}).\mathit{Amount} \neq 0 \vee \mathit{DetBy}(\mathit{Amount}, \mathrm{data})$
	violation	$(\mathit{some}\ \mathrm{stop}(L).\ \neg\mathit{MayDepOn}(L, \mathrm{callvalue})) \ \wedge \ (\mathit{all}\ \mathrm{call}(\_, \_, \_, \mathit{Amo}$

Figure 6.5: Security patterns checked by Securify2 to discover Ether Liquidity vulnerability. This figure is taken from [35]. The problem was that users could transfer ether towards the contract, but not withdraw their funds anymore. The first compliance pattern verifies that not all functions increases the contract's balance. The second compliance pattern is similar and is satisfied whether all transactions that can complete with success are able to decrease the wallet-contract balance, it is another formulation of the first pattern. The violation pattern is satisfied whether all withdraw function calls transfer 0 ether. This means that the balance is frozen.

No Write After Calls (Reentrancy)

As explained in figure 4.1, reentrancy caused a lot of damages, especially in 2016 with the DAO contract[6]. To spot reentrancy, Securify2 developers added one compliance pattern verifying that functions of verified contracts are not able to call malicious functions changing maliciously the contract state, using control-flow facts and reasoning about the internal storage of the contract. On the other hand, the violation pattern over-estimates a property verifying whether a function call to an external function, which may be malicious, is followed by a write towards the storage of the contract.

Property	Type	Security Pattern
NW: No writes after call	*compliance*	$all\ \text{call}(L_1, \_, \_, \_).\ all\ \text{sstore}(L_2, \_, \_).\ \neg MayFollow(L_1, L_2)$
	violation	$some\ \text{call}(L_1, \_, \_, \_).\ some\ \text{sstore}(L_2, \_, \_).\ MustFollow(L_1, L_2)$

Figure 6.6: Compliane and violation patterns expressing Reentrancy

6.2 Tests performed by Securify2

In this section, we will present the vulnerabilities that the Securify2 tool can detect. First of all the vulnerabilities detected on the Wablieft project to have a concrete example of use. Then, other vulnerabilities not present on Wablieft, but detectable by this tool, will be presented.

6.2.1 Vulnerabilities found in Wablieft

To understand the first vulnerability, some small details about the Solidity language are necessary. In Solidity there are two places where variables are located : in memory and in the storage. The storage is used when some variables need to be part of the contract state, such as an account balance, the contract's owner or several attributes which need to be stored in the blockchain. On the other hand, memory is used to save temporary variables , and disappears at the end of the execution on the EVM. By default, state variables (variables declared outside of functions) are located in the storage, while variables declared inside functions are stored in memory.

Unrestricted write to storage

In the code sample from Wablieft just below, a set structure, located in the storage, is passed as a parameter to this function. The vulnerability comes from the fact

that the function does not check which user (or address) is calling the contract, and therefore, who has the right to modify the data that will be stored in the blockchain.

```
struct Set {
    bytes32[] _values;
    mapping (bytes32 => uint256) _indexes;
}

function _remove(Set storage set, bytes32 value) private returns (
    bool) {
    set._values[toDeleteIndex] = lastvalue;
    set._indexes[lastvalue] = toDeleteIndex + 1;
    //..
}
```

In Solidity , a simple way to avoid this problem and control from which address a contract is called is by adding a constraint on it. This constraint could be defined by a *require* statement, as shown in the example right below.

```
address owner = 0x99 ... ;
require(msg.sender == owner);
```

Concerning Wablieft, this function is only called through another function which is protected by a *require* statement. Therefore, the access to the contract storage is protected. Additionally, Securify's analysis is useful because it warns the testers about the dangers of calling this kind of unprotected functions.

Minor vulnerabilities

Securify2 also allows the detection of many smaller vulnerabilities, which are less critical but could inform the programmer about various behaviours of the contracts.

- External Calls of Functions: A public function never called within the contract but only from an external contract should be marked up as external.

  ```
  function addMarketplaces (address [] memory _marketplaces)
      onlyOwner public {
      //..
  }
  ```

- Avoid complex Solidity version: Allowing the contract to be compiled under a Solidity version that is later than the contract makes it difficult to know exactly how the contract will be executed. A way to prevent this unexpected behavior is by forcing the version of the compiler to fit the contract version, with the removal of the "^" symbol from the line below.

  ```
  pragma solidity ^0.6.10;
  ```

6.2.2 Other detectable vulnerabilities

Reetrancy

Securify2, like Solidity's Model Checker, can detect possible reetrancy. But not in the same way as the Model Checker. Indeed, while the Model Checker allows to check the reentrance via the *assert* statements, Securify2 will rather look at the sequence of the semantic facts.

```
1  contract Reentrancy {
2      mapping (address => uint) public balances;
3
4      function withdraw(uint amount) public {
5          if (balances[msg.sender] >= amount){
6              (bool success, ) = msg.sender.call{value:amount}("");
7              require(success);
8              balances[msg.sender] -= amount;
9          }
10     }
11 }
```

In the Reentrancy contract above, the users's balance are checked at the beginning of the function. After that, an ether transfer call is performed, then only the balance is updated. If a user called back the function before the update, they could exploit the vulnerability. The solution to this problem is to make sure all internal state changes are performed before any call is executed. It would be sufficient to move the update of the balances[msg.sender] just before the call. Otherwise, Securify2 will throw a warning reporting a possible reentrancy. Indeed the statements modifying the state of the contract after an ether transfer could lead to that vulnerability.

Unchecked return value

Another example is the value returned by an external function call that is never used nor checked. The execution will continue even if the called contract throws an exception. Unexpected behaviours in the subsequent program logic could be caused by an accidental failure during the execution. A solution could be to check whether the return value was successful in order to ensure that the call will not change the behaviour of the function.

```
1 contract ReturnValue {
2     function callnotchecked(address someContract) public {
3         someContract.call("");
4     }
5 }
```

This vulnerability detected by Securify2 was implemented when Solidity version 4 was still widely used. At this time, newer solidity compiler versions (they are currently at version 8) spotted this vulnerability on their own by issuing a warning, and this is the case for most of the vulnerabilities found by Securify2. While these vulnerabilities should therefore occur less and less, Securify's analysis is still relevant for users who use older contracts versions or compiler versions which may not be aware of these vulnerabilities.

Securify2 analyses many other vulnerabilities that will not be covered in this document, but which are listed in the appendix B.1.

Chapter 7

Fuzzing: Echidna

Echidna[55] is an Open-Source Fuzzer for Ethereum Smart Contracts. The input generation technique of Echidna is based on finding violation in *assertion* statements, user-defined properties and gas use estimation. It supports two Ethereum Smart Contract languages such as Solidity and Vyper, which is a python lookalike language targeting the Ethereum Virtual Machine. Echidna has been used in several large paid security audits. Developers focus on three main aspects : easy to use, good coverage of contract vulnerabilities and fast. The major problem they had with using fuzzing techniques on Smart Contract is the complexity of representing the semantics of blockchain execution. Because as said earlier, when a contract has been deployed, it cannot be updated anymore. Therefore there are many techniques such as modeling or even emulating a local blockchain.

7.1 Implementation

The architecture is divided in 2 parts : a preprocessing stage and a fuzzing campaign. During the preprocessing part, the solidity source code of a contract is compiled and analyzed by Slither([56]), a static analyzer for Ethereum Smart Contracts which returns a set of vulnerabilities discovered within the contract. In addition, the compiled contract is also provided by Slither (.abi and .bin files). During the Fuzzing campaign, Echidna generates transactions from the compiled files and the output of Slither, computed during the previous phase, such that it can target more easier and faster the vulnerabilities in order to generate counterexamples.

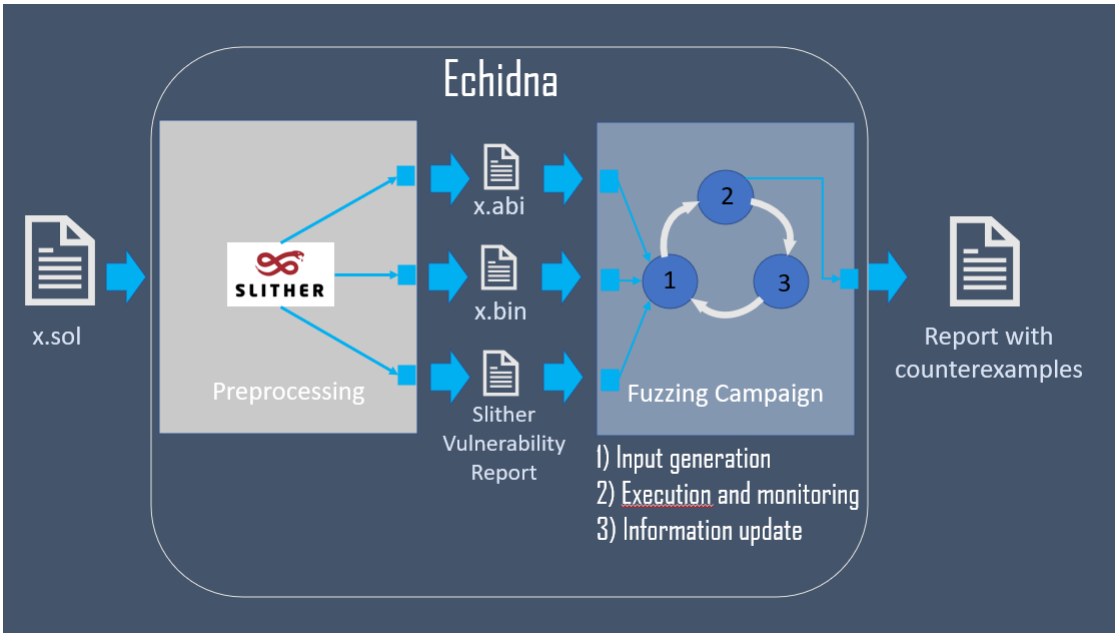

Figure 7.1: Echidna's architecture

7.2 Tests performed by Echidna

7.2.1 Detectable vulnerabilities

The test panel of the echidna fuzzer is quite limited. Indeed, unlike the two other tools presented above, no test is preconfigured. Using either asserts or functions acting as invariants is the only way provided to verify contracts.

We use the echidna_testInvariant function to check a property that must be true throughout the execution, otherwise an error is thrown, in the same way than an assert. For echidna, all functions returning a boolean and starting with the keyword echidna work as asserts and are checked through the whole execution. In the example, the fuzzer will test many inputs until it finds the one that sets the boolean state1, state2 and state3 to true, leanding to throw the error via the function echidna_testInvariant.

```
1  bool state1 = false;
2  bool state2 = false;
3  bool state3 = false;
4
5  function f1 (uint x) public {
6      require (x == 10);
7      state1 = true;
8  }
9
10 function f2 (uint x) public {
11     require(state1);
12     require(x==30);
13     state2 = true2;
14 }
15
16 function f3 () public {
17     require(state2);
18     state3 = true;
19 }
20
21 function echidna_testInvariant () public returns (bool) {
22     return (!state3);
23 }
```

The program then gives the set of functions to call and their respective parameters, in the order to obtain the invariant or assert that is violated. Here for example it will be first f1(10), then f2(30) and finally f3(). We can help the fuzzer by specifying it which functions to ignore for the testing, in order to avoid unnecessary paths. The technique has its limitations, and for instance, it cannot handle reentrance. Indeed, it would be difficult for it to tell in which order functions should be called to run a certain assertions, since a reentrance occurs when a function is called before another one is completed.

Chapter 8

Testing techniques comparison

8.1 Summary of the techniques

Securify2 - Static Analysis

Static analysis examines the entire source code looking for syntactic matches of known vulnerabilities. The tool analyses code behaviours and matches semantic fact sequences into "compliance" or "violations" patterns in order determine whether they are vulnerable or not.

Echidna - Fuzzing

Echinda Fuzzer works in two steps, in the manner of a concolic execution[57]. At first, it statically analyzes the contract under test with Slither[56] and then generates inputs based on its vulnerability report. These inputs aims to trig the reported vulnerabilities by fuzzing. The input space could be extremely huge. Therefore, as long as Echinda is running, it never stops generating new inputs in order to discover the most vulnerabilities.

Solidity SMT Checker - Model Checking

This technique formally verifies whether user-defined security properties are holding at any time given a Smart Contract. The Solidity SMT checker also automatically verifies several correctness properties such as arithmetic operations and assertion failures.

8.2 False Positives and Negatives

Securify2 - Static Analysis

The disadvantage of static analysis is its non-triviality to test the entire system for certain codes, test tools have to infer a large number of approximations for the analyse, which may lead to inaccuracies. These approximations lead to report several false positive vulnerabilities. The problem with false positives is that it is up to the user to determine afterwards whether it is really a vulnerability or not. A large amount of false positives decrease the relevance and the accuracy of the analysis, as these results cannot be trust at 100%. Nevertheless, the developers of Securify2 managed to reduce the amount of false positives by using compliance patterns which are able to classify parts of code that are safe behaviour. In addition, Securify2 indicates to the user either it is a verified vulnerability or a "warning" i.e. a potential vulnerability that is either a true positive or false positive.

Finally Securify2 statically explores all contract behaviours and analyses them avoiding undesirable false negatives.

Echidna - Fuzzing

There are no false positives in fuzzing: Every data input that allows the program to be disrupted is a real potential attack and a real security threat.

SMT Checker - Model Checking

The SMT Model Checher claims to be sound but only for the language elements supported. It will never indicate a false negative. Among the detected vulnerabilities, it will differentiate between confirmed vulnerabilities by announcing "vulnerability happens here" and uncertain ones with "vulnerability might happen here". For confirmed vulnerabilities, The Solidity SMT Checker will try to return a counter-example in order to show what inputs led to the property failure. Concerning Wablieft, many vulnerabilities were reported but had to be checked manually by the tester, while for some it was obvious that they were false positives.

8.3 Analysable vulnerabilities

Securify2 - Static Analysis

The great advantage of Securify2 is the testing automation, and the large list of detectable vulnerabilities (see Appendix B.1). In addition, it is also possible to

add new matches of semantic fact sequences and therefore detect new vulnerabilities.

The major deficiency of Securify2 is the impossibility to check the value stored inside particular variable at any point in the code due to the fact that it does not handle *assert* statements. Therefore, some interresting tests are impossible to be executed. On the other hand, this weakness is compensated by its large list of detectable vulnerabilities. For example Securify2 will not test reentrancy via asserts. Instead, if a function detects an update of an account balance after a money transfer, it will warn the tester of a potential vulnerability.

The other major disadvantage of Securify2 is that it cannot reason about numerical properties, such as overflows, underflows, out of bounds index accesses.. Fortunately, Solidity's Model Checker is very complementary as it allows you to test these unverified properties.

SMT Checker - Model Checking

As explained above, the SMT Checker allows to test many properties related to numerical properties. It checks automatically for over/underflows, out of bounds array accesses, verification whether the funds of a contract are sufficient to perform a transfer. It also detects itself whether each statement and variables of the code are reachable and throws an error if not. On the other hand, Securify2 assumes that all instructions in the contract are accessible and doesn't detect it. [35].

The real difference between Solidity SMT checker and Securify2 is the possibility to check the *assert* statements put in the contract. It allows you to test more complex vulnerabilities and properties based on whether the contracts are in particular states, with certain values assigned to their variables. For instance, we can test by a function containing an assert and acting as an invariant that a variable will never reach a certain value (example of the Robot figure 5.8).
As explained before, this also allows the tester to detect reentrance (Mutex example) by checking, via an assert, that the value assigned to a variable has not been maliciously corrupted until the end of the execution.

Although checking the asserts is a powerful method to ensure correctness, it is not automated at all and requires the tester to deeply understand the expected and unexpected behaviours of its program. For instance, we had to understand how Wablieft was meant to behave in order to determine which assertions to place within the code.

Echidna - Fuzzing

The Echidna fuzzer does not detect vulnerabilities on its own. Echidna has no preconfigured tests for overflow/underflow, out of bound array accesses, etc. It only allows, like the SMT checker, to check whether the assert statements placed within the code are not violated. Unfortunately, it does not handle calls to external contracts which the code is unknown. In the Mutex example, we were not able to test the reentrancy with the fuzzer because we did not have any knowledge of the external contract called. Therefore, Assertion testing with Echinda is way less powerful than Solidity SMT Model Checker.

As a result, it doesn't really have an advantage over Solidity SMT Checker. Its only real advantage is that it provides the exact sequence of functions (with parameters for each of them) that allows an assert or an invariant to be violated. See example figure 7.2.1.

Chapter 9

Contribution to Securify2

9.1 Description of the implemented feature

Although Securify2 was able to analyze many properties of Smart Contracts, it was not able to analyze contracts which make function calls to imported contracts located in other files such as libraries.

As the Wablieft project (Section 2.5) contained contracts spread into several files, we had to adapt Securify2 to make its analysis feasible. We managed it by manuallycreating a new and unique file containing a aggregated code of all the contracts in the project. Then we were able to perform the analysis on it.
Another solution was to improve Securify2 by adding the code of an open-source flattener software. The last solution was to create a flattener ourselves. This is the option we chose. A Solidity flattener allows to flatten contracts, that is, allowing to combine in a newly created file an initial solidity file with the contracts that are imported by it.

For example, in the Wablieft project, there is a contract named WablieftOracle.sol, which imports the contracts Ownable.sol, AccessControl.sol and WablieftInterface.sol. These are located in different folders, as shown in Figure 9.1. The role of the flattener will be to create a new "flattened" file which will contain the code for all these contracts.

We could have created this "flattened" file manually instead of using a flattener but we decided not to because this technique can be very tedious and time consuming if the number of contracts to be analyzed is large, as well as if these contracts are scattered in a large number of files. We also decided not to use existing flattener software because from the ones we found, among those that were open-source,

```
contracts
├── WablieftOracle.sol
├── WablieftInterface.sol
├── access
│   ├── AccessControl.sol
│   └── Ownable.sol
├── GSN
│   └── Context.sol
└── utils
    ├── Address.sol
    └── EnumerableSet.sol
```

Figure 9.1: Contract WablieftOracle.sol directory tree

each preented at least one drawback. Some were coded in languages other than Python, the language used for Securify. Some used the npm library and its solidity compiler solcjs which is not exactly the same as the solc compiler used by Securify2 (some parameters are available with solc and not with solcjs). For the sake of consistency and compatibility with the existing program we therefore decided to create a flattener in python ourselves.
We contacted the designers of Securify2 and asked them why the software would not handle imports. They told us that this feature was still to be developed and that they would be very interested in our contribution. So we started to implement the flattener (code in Annexe B.3), and after a first version and some adjustments in consultation with the developers, it was finally accepted on the Securify2 Github directory (https://github.com/eth-sri/securify2/tree/develop).

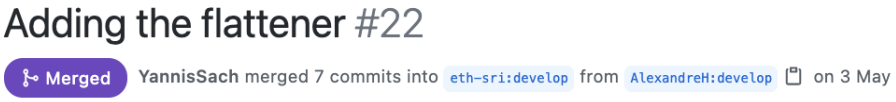

Figure 9.2: Pull request accepted for Securify2 !

9.2 Implementation of the flattener

The implementation of the flattener started from the observation that the solc compiler had a very useful parameter (–ast-compact-json), allowing, when compiling a solidity file, to generate a json containing the abstract syntax tree (AST) of the file's contracts. The abstract syntax tree is a tree representation of the abstract syntactic structure of a given code. An AST [58] captures the essential structure of

the input in a tree. For example it omits to represent punctuation marks of a solidity code such as semi-colons to terminate statements or commas to separate function arguments. They are not necessary because such information is directly represented in ASTs by the structure of the tree. For example, the simple declaration of a variable x

```
uint x;
```

is represented in the AST (in simplified form) by :

```
{
    "visibility": "internal"
    "constant": false,
    "mutability": "mutable",
    "name": "x",
    "nodeType": "VariableDeclaration",
    "typeName":
    {
        "name": "uint",
        "nodeType": "ElementaryTypeName",
    },
}
```

With the AST of Solidity codes, it was possible for us to create a flattener, which actually works like top-down parser. Indeed, as a top-down parser the flattener runs through an input (the generated AST generated from a solidity code), from the its highest level to the terminal values. The difference with a parser is that the parser will check whether the input respects the grammatical rules of the input's language. Instead, the flattener will go through all the elements of the AST, that contains both the data from the original file and the imported contracts, in order to write a new solidity file little by little while respecting the grammar rules of the solidity language.

The AST of a solidity code generated via the solidity compiler will always have the same structure, regardless of the compiled code. It will be in json form, and will have the same high-level element each time, the json key "source-unit" whose value contains the data of the entire file. We can see in the figure 9.3 how a solidity file is structured: a source-unit can be composed of several contract definitions, structure definitions, a compiler-required version (the line with pragma) etc.

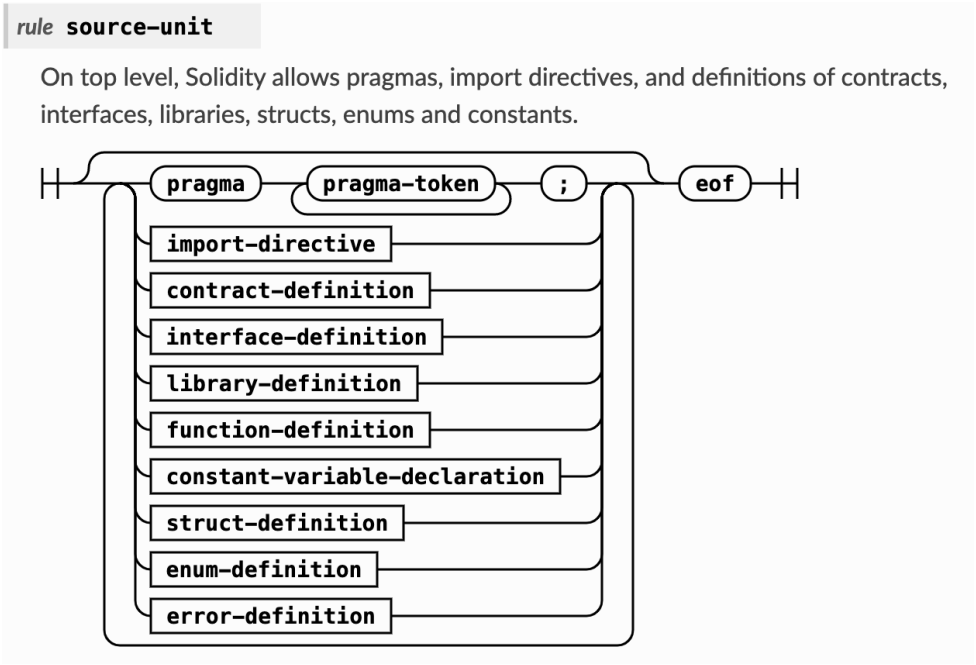

Figure 9.3: source-unit - base element of solidity files [59]

The figure 9.3 tells us that a solidity file can also be empty. We tested that, and the AST generated from an empty solidity file gave us the json in the next figure The nodeType value is source-unit, the top of the file structure. If the file had contained contracts or functions, they would have been found in the value corresponding to the "nodes" key in the json.

```
======= test.sol =======
{
  "absolutePath": "test.sol",
  "exportedSymbols": {},
  "id": 1,
  "nodeType": "SourceUnit",
  "nodes": [],
  "src": "0:0:0"
}
```

In this array of nodes, each element would have its own nodeType json key corresponding to its type. And these elements would have been composed of sub-parts over and over again, until arriving at json nodes that contain terminal values, such as a nodeType "boolean" that contains either a True or False value, as shown in figure 9.4.

The flattener will generate the "flattened" file by parsing the json that contains both the original contract and the imported contracts as we see for the

source-unit type (figure 9.3). Each node has its own type and we have created a function for each type that knows how to write that type. As solidity is a context free language [60], the writing rules will remain the same, no matter where the node is located in the json. For example we know that a type "functionCall" is written as: *expression '(' functionCallArguments ')'* no matter where it is. The advantage is that this rule can be reused, for example a function call will be written in the same way if it's used in a variable assignment or in a function parameter.

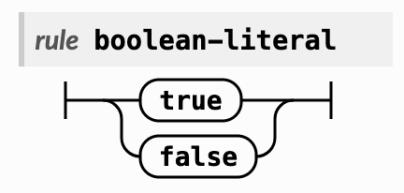

Figure 9.4: Boolean - terminal value of a soldity code

Some additional notes are interesting about the implementation of the flattener: Firstly, it only works if the importing contract and the imported contracts are compilable. Otherwise, the AST json cannot be generated. Secondly, the flattener can handle the different versions of solidity (from 5 to 8). Because although grammar elements have been added in the course of the versions, the structure of the language and the generated AST have remained the same. It was therefore possible to adapt the flattener so that it could handle the old with the new features of the language. But if other additions were to be made to the language, the flattener would not be able to handle them, without making an update.

9.3 Tests of the flattener

In order tp test the flattener, we used a large database available on the Etherscan.io website where a list of the last 10,000 published open-source Smart Contracts was downloadable. To use these contracts, we first sorted them according to their versions. The figure 9.3 gives an overview of the distribution of the versions, the unknown version corresponding to contracts whose version did not match to any of our compilers (for example, a contract that only allows version 0.5.7 when we had version 0.5.12).

Our methodology consisted in taking the AST of the original contract and comparing it with the AST of the contract that was generated by the flattener. Having two different ASTs means that the fresh flattened contract generated is

Solidity 4	Solidity 5	Solidity 6	Solidity 7	Solidity 8	Unknown
1002	1905	5026	1138	297	632

Figure 9.5: Versions of Etherscan contracts

not equivalent to the base contract and therefore leads to an incorrect output. All Solidity versions until the latest released (0.8.4) were tested in order to ensure the flattener correctness against all Solidity language elements. We can observe that the majority of the contracts in our training set were in version 6, therefore the flattener should provide the most accurate outputs with this version of Solidity.

The flattener was therefore implemented using a trial and error approach by testing on real contracts that the rewriting of the flattener was correct. Although we could have situations where the flattener does not know how to handle language elements, this is very unlikely, especially for version 6, as the flattener has been tested on a large number of contracts. If in the future new versions of Solidity were to arrive, it might not be able to handle some of these contracts well. But the way the program is designed, it would be easy to update it to handle these new features.

Chapter 10

Contribution to Wablieft

The wablieft project described in section 2.5 gave us a concrete example of how Smart Contracts could be useful in real life. The description of the project in the paper[32] gave us a good understanding of Wablieft. Additionnaly, the properties and behaviours described within it allowed us to adjust our tests in order to verify that the Smart Contracts were behaving as expected.

Therefore, Wablieft allowed us to check more general vulnerabilities that may be present on all Smart Contracts, depending on how they have been coded. These tests allowed us to check that the vulnerabilities supposed to be detected by the tools were actually detected, and therefore we were able to verify the proper functioning of these tools. This also allowed us to compare the testing capabilities of these tools, and to verify their advantages and disadvantages over the others.

In this chapter we will first list the specific vulnerabilities discovered in Wablieft, followed by the verification of various properties related to the correct functioning of the program according to the expected behaviour based on the paper.

We will not re-explain the vulnerabilities analyzed by the tools and which were not present in Wablieft as they were explained in detail in the test sections of Securify2 and SMT Checker. We will just present here the parts of Wablieft's Smart Contracts that were problematic, either because there were a real threat or because care should be taken as the code was not presented as dangerous at the time but could become in the future.

10.1 Vulnerabilities detected

The vulnerabilities that were discovered by the different testing methods were communicated to Lionel Metongnon during the year. Therefore, after reviewing them with him, we only kept those that seemed relevant in our master thesis and which could generate potential vulnerabilities (dangerous or not). These vulnerabilities are also explained in the Tests performed on Wablieft sections of Securify2 and the SMT Checker. But the main part will be presented here to create a complete summary of the vulnerabilities found on Wablieft.

The first vulnerability found in the Wablieft code is an overflow that could occur if the sum of the values added to size becomes too large :

```
size += prescriptionToCoupons[patientPresc[i]].length;
```

The second vulnerability found was an underflow that could occur if the value of set._values.length was 0 and then subtracted 1:

```
uint256 lastIndex = set._values.length - 1;
```

The third vulnerability was the remove function allowing to modify values in the storage (values that will be recorded pushed and recorded in the blockchain) while the access to this function was not restricted to certain authorised users. In fact, as this function was not used in the contracts and therefore not callable, it is not a vulnerability. But if later on, this function was used, it would be necessary to put restrictions on its usage because it is not protected:

```
function _remove(Set storage set, bytes32 value) private returns (
    bool) {
     set._values[toDeleteIndex] = lastvalue;
     set._indexes[lastvalue] = toDeleteIndex + 1; //..
}
```

The last two vulnerabilities are not really vulnerabilities, but rather good contract writing methods to respect: The first is not to allow compilers of later versions to be able to compile this contract, as these could be executed in a different way than expected. To be on the safe side, it is recommended to only allow the compiler version that matches the contract:

```
pragma solidity ^0.6.10;
```

The last recommendation is to set the visibility of a function to external (only callable by another contract) instead of public. Indeed, the keyword "public" also allows internal calls of this function, but when it is not necessary, it might as well be forbidden:

```
function addMarketplaces (address [] memory _marketplaces)
    onlyOwner public {//.. }
```

10.2 Verification of properties

A coupon can be used only once

The properties we wanted to check were taken straight from the paper describing Wablieft[32]. As a reminder, Wablieft proposes a Marketplace allowing customers to use coupons to receive health care services, aiming to improve the organization within hospitals. The patient, who has previously received his coupons via a doctor, is supposed to use them only once each.

Thanks to the SMT Checker (Model Checking) we were able to discover that the reuse of a coupon was possible via the useCoupon function. It was by setting the assert at line 6 that we discovered that an unexpected behaviour could occur at this point in the code. Indeed, we wanted to check that when a coupon was set to be used, it was never set to true beforehand. And this was not the case. By adding the couponNotUsed modifier that existed elsewhere in the code but was not used at that point, we ensured that someone who tried to reuse a coupon a second time would be blocked at the entry of the function. This discovery allowed us to improve the Wablieft code by avoiding this undesirable behaviour.

```
1
2   function useCoupon (
3     uint _id
4   ) external _couponExists(_id) _couponNotUsed(_couponId) {
5       //..
6       assert(coupons[_id].used == false);
7       coupons[_id].used = true;
8       //..
9   }
```

A forged coupon (i.e. not issued by the marketplace) cannot be used

For this property we had to check that only fresh coupons created by the shared marketplace can be used on the shared marketplace. Verifying this property was quite easy to check because in the code, a require checked, via the library AccessControl, that only hospitals could create coupons. If not, the function call would be blocked at the *require* statement.

```
1 function createCoupon(uint _id, uint _serviceId, uint _hospitalId)
      external {
2     //..
3     require (hasRole(HOSPITAL, _msgSender()), "Does not have
          Hospital Role");
4     //..
5 }
```

Coupon issued for the right patient

The last property we checked was that a patient cannot receive coupons issued to other patients. This property was also easily verified. Indeed, there is only one function that allows to associate patients with coupons. This is the function createConsultation, and this one is protected by a require that ensures that only a hospital can call it. It is during this function that coupons are issued to patients, and by checking at the end of this function that the patient assigned corresponds to the one expected, the property is verified. This verification was carried out via the assert presented on line 8 of the code.

```
1  function createConsultation(
2      uint _id,
3      uint _doctorId,
4      uint _patientId,
5      uint[] memory _couponIds
6      ) external _patientExists(_patientId) _doctorExists(_doctorId)
           {
7      //..
8      assert(consultations[_id].patientId == _patientId);
9      //..
10 }
```

The last property we wanted to check was that "A revoked coupon cannot be used". It means that since a coupon may be lost or replaced, this coupon could be revoked. Once a coupon is revoked it cannot be used anymore. We were unable to check this property because there was currently no function allowing us to remove coupons, but after discussions with Lionel Metongnon about how it could be implemented, checking the property would be simple. Indeed, by creating a list in which the revoked coupons would be added, we could test with a *require* statement that the coupon used by a user did not match any coupon of this list, and in the contrary case the function receiving a revoked coupon would be stopped.

Chapter 11

Conclusion

The first goal of our master thesis was to discover and determine efficient ways to verify Smart Contracts. We started by learning the backgrounds of the Ethereum Blockchain and the major difference with Bitcoin Blockchain : the Smart Contracts. They provide a large amount of benefits such as taking the role of a third party in transactions.

Smart Contracts are written by humans. Unfortunately, this may lead to several vulnerabilities costing millions of dollars. A way to prevent these vulnerabilities is by verifying them before their deployment. We briefly compared multiple verification techniques and their related tools in order to finally select three out of them : Solidity SMT Model Checker or SSC(**Model Checking**), Securify2(**Static Analysis**) and Echinda(**Fuzzing**).

During the testing phase on a set of vulnerable contracts, we made various observations about their advantages and disadvantages. Some are more automated than the others. For instance, Securify2 has the benefit of having a large and fully automated test battery provided by the developers. This automation is also its limitation since it only verifies a list of 37 vulnerabilities. Similarly, SSC also provides several automated verification features such as arithmetic operations, but its main advantage is its versatility. Indeed, Testers are able to specify much more specific properties through the *assert* statements, which are not covered by Securify2. In return, a strong knowledge of the contract under test is required.
Echinda also targets the *assert* statements through its fuzzing techniques but does not handle the other vulnerabilities automatically verified by SSC nor Securify2. As SSC, Echinda also requires a strong knowledge of the contract. We figured out that Echinda, with its fuzzing techniques, discovered way less vulnerabilities than the other tools.
We concluded that Securify2 and SSC are complementary and their combination

allows a sufficient and powerful analysis without the need of Echidna. At this time, Some features of the Solidity Language are not yet handled by SSC, but will tend to be in the future. We strongly believe that the Solidity SMT Model Checker will produce extremely accurate results when all the language features will be supported.

Our second goal was to combine these techniques in order to discover the most vulnerabilities on a concrete situation such as Wablieft, the shared market place for medical services. By collaborating with the authors of Wablieft, we were able to improve our knowledge about the project and then to look up for vulnerabilities way more efficiently. We managed to discover several vulnerabilities, even critical ones, by complementary applying each tool on the Wablieft Solidity project.

Concerning Securify2, we managed to improve the program by allowing the testers to verify their Smart Contracts importing other local contracts. As a reward, we got our pull request accepted by the developers on the development branch of their Github repository.

Finally, executing verification tools on Smart Contracts is only the last step of the verification. As said earlier, Smart Contracts are written by humans and subjects to error. Defensive programming could reduce drastically the time required for a tester to eliminate as many vulnerabilities as possible. We will end up this conclusion by introducing several important programming rules to follow before using a verification tool.

Simpler is more secure

Developers must try to reduce their code complexity. We can take the example of Swiss cheese and holes. Swiss cheese is known to have a lot of holes inside it. Therefore, having more cheese involves having more holes. This example illustrates quite well the code containing errors analogy. A small program having exactly the same behaviour as a bigger program should probably be safer.

Never reinvent the wheel

A lot of safe libraries or contracts already exist. One of the most famous is safeMath which implements functions defining safe mathematical operations, in order to avoid any related vulnerability such as arithmetic overflows or arithmetic underflows. Another advice is to define functions with sequence of statements appearing multiple time in the code, as a goal to avoid repetition and to decrease the code size.

Code quality

Writing a Smart Contract should be taken seriously because any bug can lead to financial loss. In addition, any Smart Contract is publicly available. A correct and well documented code structure should help the Ethereum community to read and understand a Solidity project, in the case of an audit.

Using a test-blockchain

Running a contract locally on a test-blockchain and feeding it with every kind of arguments, in the manner of a fuzzer, could help the programmer to understand the behaviours of the contract. This is not as powerful as a real fuzzer but the programmer could already know some specific inputs which might crash the program.

Appendix A

SMT Checker

A.1 CHC system generation algorithm

Input : A contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$.
Output : The set of CHC Π_C.
Initially: $\Pi_C = \{(\mathsf{Init}_C), (\mathsf{ExtBase}_C)\}$.

foreach $f = \langle G, \alpha, \omega, \epsilon, \rho \rangle \in F$ **do**
  Let $\boldsymbol{a}, \boldsymbol{r}, \boldsymbol{l}$ respectively the arguments, returns and local variables of f.
  Let $\Pi_f \coloneqq \{(\mathsf{Entry}_f), (\mathsf{Sum}_f)\}$
  Let $G = (V, E, \lambda, \mu)$
  foreach $e = \langle v, w \rangle \in E$ **do**
    if *v contains a call to $g(\boldsymbol{a}_g) \to \boldsymbol{r}_g$* **then**
      Create ρ_{call} from λ_v
      if (Sum_g) *is known* **then** $\mathsf{SSA}_{\lambda_v} \coloneqq (\mathsf{Call}_{g,\rho_{call}})$;
      else $\mathsf{SSA}_{\lambda_v} \coloneqq (\mathsf{ECall}_{\rho_{call}})$;
    else
      $\mathsf{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \coloneqq \mathrm{Model}(\lambda_v)$
    end
    $\mathsf{SSA}_{\mu_e} \coloneqq \mathrm{Model}(\mu_e)$
    $\Pi_f \coloneqq \Pi_f \cup \{(\mathsf{Jump}_{f,e})\}$
  end
  $\Pi_C \coloneqq \Pi_C \cup \Pi_f$
  if $f \in F^+$ **then**
    $\Pi_C \coloneqq \Pi_C \cup \{(\mathsf{ExtInd}_{C,f}), (\mathsf{RootTr}_{C,f})\}$
  end
end

Figure A.1: Algorithm generating CHC system from [49]

Appendix B

Securify2

B.1 Verified Vulnerabilities by Securify2

Smart Contract Vulnerabilities			
ID	Pattern name	Severity	Smart Contract Weakness (SCW)
1	TODAmount	Critical	SCW-114
2	TODReceiver	Critical	SCW-114
3	TODTransfer	Critical	SCW-114
4	UnrestrictedWrite	Critical	SCW-124
5	RightToLeftOverride	High	SCW-130
6	ShawdowedStateVariable	High	SCW-119
7	UnrestrictedSelfdestruct	High	SCW-106
8	UninitializedStateVariable	High	SCW-109
9	UninitializedStorage	High	SCW-109
10	UnrestrictedDelegateCall	High	SCW-112
11	DAO	High	SCW-107
12	ERC20Interface	Medium	-
13	ERC721Interface	Medium	-
14	IncorrectEquality	Medium	SCW-132
15	LockedEther	Medium	-
16	ReentrancyNoETH	Medium	SCW-107
17	TxOrigin	Medium	SCW-115
18	UnhandledException	Medium	-
19	UnrestrictedEtherFlow	Medium	SCW-105
20	UninitializedLocal	Medium	SCW-109
21	UnusedReturn	Medium	SCW-104
22	ShadowedBuiltin	Low	-

23	ShadowedLocalVariable	Low	-
24	CallToDefaultConstructor?	Low	-
25	CallInLoop	Low	SCW-104
26	ReentrancyBenign	Low	SCW-107
27	Timestamp	Low	SCW-116
28	AssemblyUsage	Info	-
29	ERC20Indexed	Info	-
30	LowLevelCalls	Info	-
31	NamingConvention	Info	-
32	SolcVersion	Info	SCW-103
33	UnusedStateVariable	Info	-
34	TooManyDigits	Info	-
35	ConstableStates	Info	-
36	ExternalFunctions	Info	-
37	StateVariablesDefaultVisibility	Info	SCW-108

B.2 Intermediate Representation

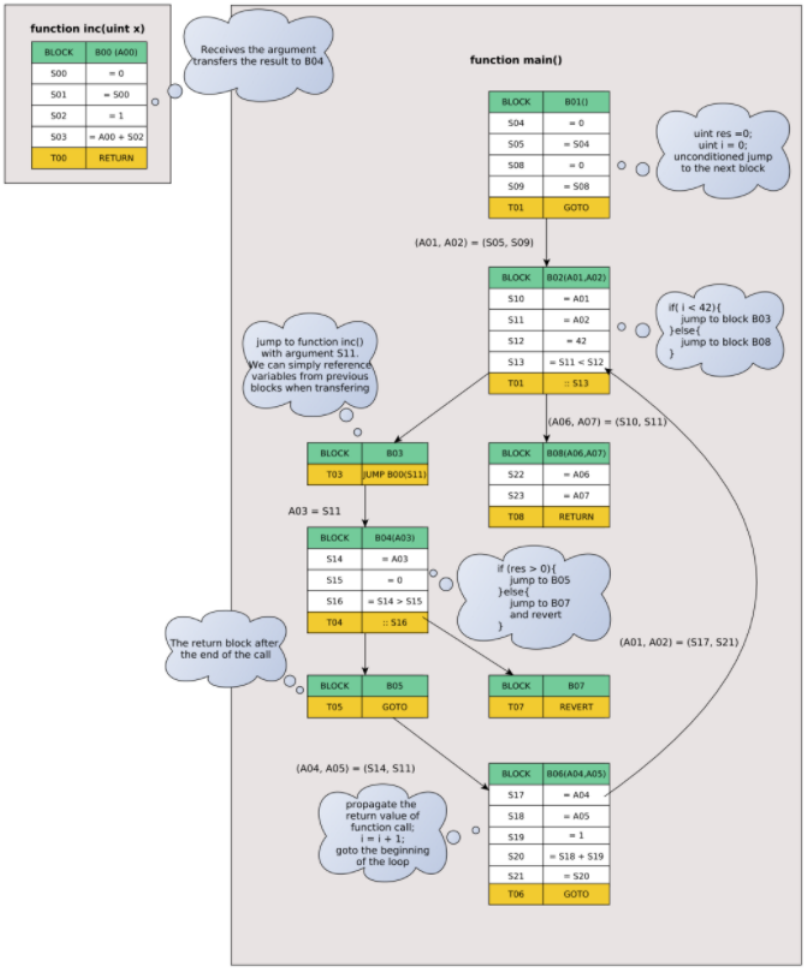

Figure B.1: Intermediate Representation of a Solidity main function running a function *inc*(*uintx*). This figure is taken from [53]

B.3 Flattener

```
1 #!/usr/bin/python3
2 #Author: Alexandre Halbardier & Martin Meerts
3
4 import json
5 import argparse
6 import sys
7 import subprocess
```

```
8
9  """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
10 Flattener for solidity files
11 """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
12
13 """
14 Functions corresponding to a nodeType
15 """
16
17 def sourceUnit(jsonFile):
18
19     license = "// SPDX-License-Identifier: " + jsonFile["license"]
           + "\n\n" if "license" in jsonFile.keys() and not jsonFile[
          "license"] is None else ""
20     return license
21
22 def pragmaDirective(jsonFile):
23
24     version = jsonFile["literals"][0]
25
26     if version == "solidity":
27         for i in range(1,len(jsonFile["literals"]),3):
28             version += " "
29             for j in range (0,3):
30                 if i+j < len(jsonFile["literals"]):
31                     version += jsonFile["literals"][i+j]
32
33     if version == "experimental":
34         version += " " + jsonFile["literals"][1]
35
36     if version == "abicoder":
37         version += " " + jsonFile["literals"][1]
38
39     pragmaVersion = "pragma "+ version+ ";\n\n"
40     return pragmaVersion
41
42 def contractDefinition(jsonFile):
43
44     documentation = jsonFile["documentation"] if "documentation"
          in jsonFile.keys() else None
45     documentation = documentation if type(documentation)== str
          else (getNodeType(documentation) if type(documentation)==
          dict else None)  #for solidity before 6
46     documentation = "/**\n* " + documentation.replace("\n","\n* ")
           + "\n*/\n" if not documentation is None else ""
47     abstract =  "abstract " if "abstract" in jsonFile.keys() and
          jsonFile["abstract"] else ""
48     contractKind = jsonFile["contractKind"] + " "
49     name = jsonFile["name"] + " "
```

```
50      listInheritedContracts = "is " + commaBetweenElements(jsonFile
            ["baseContracts"]) if commaBetweenElements(jsonFile["
            baseContracts"]) != "" else ""
51      usingForDirective = variableDeclaration = eventDefinition =
            modifierDefinition = structDefinition = functionDefinition
            = ""
52      listNodes = ""
53
54      for elem in jsonFile["nodes"]:
55          listNodes += "    " + getNodeType(elem)
56
57      listNodes= " {\n" + listNodes + "}\n\n\n"
58      return documentation + abstract + contractKind + name +
            listInheritedContracts + listNodes
59
60  def inheritanceSpecifier(jsonFile):
61
62      baseName = getNodeType(jsonFile["baseName"])
63      arguments = "(" + commaBetweenElements(jsonFile["arguments"])
            + ")" if "arguments" in jsonFile.keys() and jsonFile["
            arguments"] is not None else ""
64      return baseName + arguments
65
66  def functionDefinition(jsonFile):
67
68      documentation = jsonFile["documentation"] if "documentation"
            in jsonFile.keys() else None
69      documentation = documentation if type(documentation)== str
            else (getNodeType(documentation) if type(documentation)==
            dict else None)  #for solidity before 6
70      documentation = "/**\n         * " + documentation.replace("\n"
            ,"\n         * ") + "\n         */\n         " if not
            documentation is None else ""
71      kind = jsonFile["kind"] + " "
72      name = jsonFile["name"] + " "
73      parameters = getNodeType(jsonFile["parameters"]) +  " "
74      listModifiers = ""
75      visibility = jsonFile["visibility"] + " "
76      stateMutability = jsonFile["stateMutability"] + " " if
            jsonFile["stateMutability"] in ("mutable","view","payable",
            "pure") else ""
77      virtual = "virtual " if "virtual" in jsonFile.keys() and
            jsonFile["virtual"] == True else ""
78      overrides= "override " + getNodeType(jsonFile["overrides"]) +
            " " if "overrides" in jsonFile.keys() and jsonFile["
            overrides"] is not None else ""
79      returnParameters = "returns " + getNodeType(jsonFile["
            returnParameters"]) if not getNodeType(jsonFile["
            returnParameters"]) == "()" else ""
```

```
80      body = " " + getNodeType(jsonFile["body"]) +"\n\n" if jsonFile
            ["implemented"] == True else (";\n" if len(parameters)+len(
            returnParameters) <79 else ";\n\n" )
81
82      for elem in jsonFile["modifiers"]:
83          listModifiers+=getNodeType(elem) + " "
84
85      return  (documentation + kind + name + parameters +
            listModifiers + visibility + stateMutability +  virtual +
            overrides +returnParameters).replace("  "," ") + body
86
87  def parameterlist(jsonFile):
88
89      listParameters = "(" + commaBetweenElements(jsonFile["
            parameters"]) + ")"
90      return listParameters
91
92  def modifierInvocation(jsonFile):
93
94      listModifierName = getNodeType(jsonFile["modifierName"])
95      listArguments = "(" + commaBetweenElements(jsonFile["arguments
            "]) + ")" if "arguments" in jsonFile.keys() and jsonFile["
            arguments"] is not None else ""
96      return listModifierName + listArguments
97
98  def block(jsonFile):
99
100     listStatements = ""
101
102     for elem in jsonFile["statements"]:
103         listStatements += "        " + getNodeType(elem) + "\n"
104
105     listStatements = "{\n" + listStatements + "    }"
106     return listStatements
107
108 def variableDeclarationStatement(jsonFile):
109
110     listDeclarations = commaBetweenElements(jsonFile["declarations
            "])
111     initialValue = (" = " + getNodeType(jsonFile["initialValue"])
            if "initialValue" in jsonFile.keys() and not jsonFile["
            initialValue"] is None else "") + ";"
112
113     if len(jsonFile["declarations"])>1:
114         listDeclarations = "(" + listDeclarations + ")"
115
116     return listDeclarations + initialValue
117
118 def returnStatement(jsonFile):
```

```
119
120     expression = "return " + (getNodeType(jsonFile["expression"])
            if "expression" in jsonFile.keys() and jsonFile["expression
            "] is not None else "")  + ";"
121     return expression
122
123 def functionCall(jsonFile):
124
125     expression = getNodeType(jsonFile["expression"]) if "
            expression" in jsonFile.keys() and jsonFile["expression"]
            is not None else ""
126     listArguments = "(" + commaBetweenElements(jsonFile["arguments
            "]) + ")"
127
128     if jsonFile["kind"] == "typeConversion":
129         for elem in ["mutable","view","payable","pure"]:
130             if elem in expression:
131                 expression = elem
132
133     if jsonFile["kind"]=="structConstructorCall":
134         args = ""
135         for i in range(0,len(jsonFile["arguments"])):
136             args += (jsonFile["names"][i] + ": " if len(jsonFile["
                names"])!=0 else "") + getNodeType(jsonFile["
                arguments"][i]) + (",\n         " if i < len(
                jsonFile["arguments"])-1 else "")
137             listArguments = ("({\n         " + args +  "\n
                })") if len(jsonFile["names"])!=0 else  ("(\n
                       " + args +  "\n         )")
138
139     return expression + listArguments
140
141 def identifier(jsonFile):
142
143     name = jsonFile["name"]
144     return name
145
146 def variableDeclaration(jsonFile):
147
148     documentation = jsonFile["documentation"] if "documentation"
            in jsonFile.keys() else None
149     documentation = documentation if type(documentation)== str
            else (getNodeType(documentation) if type(documentation)==
            dict else None)  #for solidity before 6
150     documentation = "/**\n         * " + documentation.replace("\n"
            ,"\n         * ") + "\n         */\n         " if not
            documentation is None else ""
151     typeName = getNodeType(jsonFile["typeName"])
152     constant = " constant" if jsonFile["constant"] else ""
```

```
153     indexed = " indexed" if "indexed" in jsonFile.keys() and
            jsonFile["indexed"] else ""
154     immutable = " immutable" if "mutability" in jsonFile.keys()
            and (not jsonFile["mutability"] or jsonFile["mutability"]==
            "immutable") else ""
155     storageLocation = " " + jsonFile["storageLocation"] if not
            jsonFile["storageLocation"] == "default" else ""
156     override = " override " + getNodeType(jsonFile["overrides"]) +
             " " if "overrides" in jsonFile.keys() and jsonFile["
            overrides"] is not None else ""
157     visibility = " " + jsonFile["visibility"] if jsonFile["
            stateVariable"] ==True else ""
158     name = " " + jsonFile["name"] if not jsonFile["name"] == ""
            else ""
159     value = " = " + getNodeType(jsonFile["value"]) if "value" in
            jsonFile.keys() and jsonFile["value"] is not None else ""
160     variableDeclaration = documentation + typeName + constant +
            indexed + immutable + storageLocation + override +
            visibility + name + value
161     variableDeclaration = variableDeclaration[:-1] if (
            variableDeclaration == typeName and variableDeclaration
            [-1]==" ") else variableDeclaration
162
163     return (variableDeclaration + (";\n" if jsonFile["
            stateVariable"] ==True else "")).replace("  "," ")
164
165 def literal(jsonFile):
166
167     constant = "constant" if jsonFile["isConstant"]==True else ""
168     value = jsonFile["value"] if "value" in jsonFile.keys() and
            jsonFile["value"] is not None else ("hex\""+jsonFile["
            hexValue"]+"\"")
169     value = "hex\""+jsonFile["hexValue"]+"\"" if jsonFile["kind"
            ]=="hexString" else value
170
171     if "\\" in value:
172         value = value.replace("\\","\\\\")
173
174     if "\u0019" in value:
175         value = value.replace("\u0019","\\x19")
176     if "\u0001" in value:
177         value = value.replace("\u0001","\\x01")
178
179     value = value.replace("\a","\\a").replace("\r","\\r").replace(
            "\t","\\t").replace("\f","\\f").replace("\v","\\v").replace
            ("\b","\\b").replace("\n","\\n")
180     value = value if jsonFile["kind"]!="string" else ("\"" + value
             + "\"" if "value" in jsonFile.keys() and jsonFile["value"]
             is not None else value)
```

```
181     subdenomination = " " + jsonFile["subdenomination"] if "
            subdenomination" in jsonFile.keys() and jsonFile["
            subdenomination"] is not None else ""
182     return constant + value + subdenomination
183
184 def modifierDefinition(jsonFile):
185
186     documentation = jsonFile["documentation"] if "documentation"
            in jsonFile.keys() else None
187     documentation = documentation if type(documentation)== str
            else (getNodeType(documentation) if type(documentation)==
            dict else None)  #for solidity before 6
188     documentation = "    /**\n        * " + documentation.replace(
            "\n","\n        * ") + "\n        */\n        " if not
            documentation is None else ""
189     name = "modifier " + jsonFile["name"] + " "
190     virtual = "virtual " if "virtual" in jsonFile.keys() and
            jsonFile["virtual"] == True else ""
191     overrides= "override " + getNodeType(jsonFile["overrides"]) +
            " " if "overrides" in jsonFile.keys() and jsonFile["
            overrides"] is not None else ""
192     listParameters = getNodeType(jsonFile["parameters"])
193     body = getNodeType(jsonFile["body"])+ "\n\n"
194     return documentation + name + listParameters + overrides +
            virtual + body
195
196 def expressionStatement(jsonFile):
197
198     expression = getNodeType(jsonFile["expression"]) + ";"
199     return expression
200
201 def binaryOperation(jsonFile):
202
203     leftExpression = getNodeType(jsonFile["leftExpression"]) + " "
204     operator = jsonFile["operator"] + " "
205     rightExpression = getNodeType(jsonFile["rightExpression"])
206     return leftExpression + operator + rightExpression
207
208 def overrideSpecifier(jsonFile):
209
210     overrides = "(" + commaBetweenElements(jsonFile["overrides"])
            + ")" if jsonFile["overrides"]!=[] else ""
211     return overrides
212
213 def userDefinedTypeName(jsonFile):
214
215     name = jsonFile["name"] if "name" in jsonFile.keys() else
            getNodeType(jsonFile["pathNode"])
216     return name
```

```
217
218 def elementaryTypeName(jsonFile):
219
220     name = jsonFile["name"] + " "
221     stateMutability = jsonFile["stateMutability"] if "
        stateMutability" in jsonFile.keys() and (jsonFile["
        stateMutability"] in ("mutable","view","payable","pure"))
        else ""
222     return name + stateMutability
223
224 def placeholderStatement(jsonFile):
225
226     return "_;"
227
228 def assignment(jsonFile):
229
230     leftHandSide = getNodeType(jsonFile["leftHandSide"]) + " "
231     operator = jsonFile["operator"] + " "
232     rightHandSide = getNodeType(jsonFile["rightHandSide"])
233     return leftHandSide + operator + rightHandSide
234
235 def memberAccess(jsonFile):
236
237     expression = getNodeType(jsonFile["expression"]) + "."
238     memberName = jsonFile["memberName"]
239     return expression + memberName
240
241 def arrayTypeName(jsonFile):
242
243     baseType = getNodeType(jsonFile["baseType"])
244     length = "["+ ("" if (not "length" in jsonFile.keys() or
        jsonFile["length"] is None) else (jsonFile["length"] if
        type(jsonFile["length"])==str else getNodeType(jsonFile["
        length"]))) + "]"
245     return  baseType + length
246
247 def structuredDocumentation(jsonFile):
248
249     text = jsonFile["text"]
250     return text
251
252 def inlineAssembly(jsonFile):
253
254     inlineAssembly = "assembly " + (getNodeType(jsonFile["AST"])
        if "AST" in jsonFile.keys() else jsonFile["operations"])
255
256     if inlineAssembly[-1] == "}":
257         inlineAssembly= inlineAssembly[:-1] + "    }"
258
```

```
259     return inlineAssembly
260
261 def yulBlock(jsonFile):
262
263     listStatements = ""
264
265     for elem in jsonFile["statements"]:
266         listStatements += "\n            " + getNodeType(elem)
267
268     listStatements = "{" + listStatements + "\n        }"
269     return listStatements
270
271 def yulAssignment(jsonFile):
272
273     variableNames = commaBetweenElements(jsonFile["variableNames"
        ])
274     value = " := " + (getNodeType(jsonFile["value"]) if "value" in
         jsonFile.keys() and jsonFile["value"] is not None else "")
275     return variableNames + value
276
277 def yulIdentifier(jsonFile):
278
279     name = jsonFile["name"]
280     return name
281
282 def yulFunctionCall(jsonFile):
283
284     functionName = getNodeType(jsonFile["functionName"])
285     listArguments = "(" + commaBetweenElements(jsonFile["arguments
        "]) + ")"
286     return functionName + listArguments
287
288 def yulVariableDeclaration(jsonFile):
289
290     variables = "let " + commaBetweenElements(jsonFile["variables"
        ])
291     value = " := " + getNodeType(jsonFile["value"])
292     return variables + value
293
294 def yulTypedName(jsonFile):
295
296     name = jsonFile["name"]
297     return name
298
299 def yulExpressionStatement(jsonFile):
300
301     expression = getNodeType(jsonFile["expression"])
302     return expression
303
```

```
304 def yulLiteral(jsonFile):
305 
306     value = ("\"" + jsonFile["value"] + "\"" if jsonFile["kind"]==
        "string" else jsonFile["value"])
307     return value
308 
309 def yulSwitch(jsonFile):
310 
311     expression = "switch " + getNodeType(jsonFile["expression"]) +
         "\n            "
312     cases = ""
313 
314     for elem in jsonFile["cases"]:
315         cases += getNodeType(elem)
316 
317     return expression + cases
318 
319 def yulCase(jsonFile):
320 
321     value = ("default " if jsonFile["value"] == "default" else "
        case " + getNodeType(jsonFile["value"]))
322     body = getNodeType(jsonFile["body"])
323     return value + body
324 
325 def yulIf(jsonFile):
326 
327     condition = "if " + getNodeType(jsonFile["condition"])
328     body = getNodeType(jsonFile["body"])
329 
330     if body[-1] == "}":
331         body= body[:-1] + "    }"
332 
333     return  condition + body
334 
335 def yulForLoop(jsonFile):
336 
337     pre = "for " + getNodeType(jsonFile["pre"])
338     condition = " " + getNodeType(jsonFile["condition"])
339     post = " " + getNodeType(jsonFile["post"])
340     body = " " + getNodeType(jsonFile["body"])
341 
342     if pre[-1] == "}":
343         pre = pre[:-1] + "    }"
344     if post[-1] == "}":
345         post = post[:-1] + "    }"
346     if body[-1] == "}":
347         body = body[:-1] + "    }"
348 
349     return pre + condition + post + body
```

```
350
351 def yulFunctionDefinition(jsonFile):
352
353     name = "function " + jsonFile["name"]
354     parameters = "(" + commaBetweenElements(jsonFile["parameters"
        ]) + ") " if "parameters" in jsonFile.keys() else "() "
355     returnVariables = "-> " + commaBetweenElements(jsonFile["
        returnVariables"]) + " " if "returnVariables" in jsonFile.
        keys() else ""
356     body = getNodeType(jsonFile["body"])
357
358     if body[-1] == "}":
359         body = body[:-1] + "    }"
360
361     return name + parameters + returnVariables + body
362
363 def elementaryTypeNameExpression(jsonFile):
364
365     typeName = jsonFile["typeName"] if type(jsonFile["typeName"])
        ==str else getNodeType(jsonFile["typeName"])
366     return typeName
367
368 def functionCallOptions(jsonFile):
369
370     expression = getNodeType(jsonFile["expression"])
371     namesOptions = ""
372
373     for i in range(0,len(jsonFile["options"])):
374         namesOptions += (jsonFile["names"][i] + ": " if len(
            jsonFile["names"])!=0 else "") + getNodeType(jsonFile["
            options"][i]) + ("," if i < len(jsonFile["options"])-1
            else "")
375
376     namesOptions = "{" + namesOptions + "}"
377     return expression + namesOptions
378
379 def ifStatement(jsonFile):
380
381     condition = "if (" + getNodeType(jsonFile["condition"]) + ")"
382     trueBody = getNodeType(jsonFile["trueBody"])
383     falseBody = ("\n        else " + getNodeType(jsonFile["
        falseBody"])) if "falseBody" in jsonFile.keys() and
        jsonFile["falseBody"] is not None else ""
384
385     if trueBody[-1] == "}":
386         trueBody = trueBody[:-1] + "    }"
387     if len(falseBody)>0 and falseBody[-1] == "}":
388         falseBody = falseBody[:-1] + "    }"
389
```

```
390     return  condition + trueBody + falseBody
391
392 def structDefinition(jsonFile):
393
394     name = "struct " + jsonFile["name"]
395     members =""
396
397     for elem in jsonFile["members"]:
398         members += "    "+getNodeType(elem) + ";\n"
399
400     members = " {\n" + members + "    }\n\n"
401     return name + members
402
403 def mapping(jsonFile):
404
405     keyType = "mapping ("+getNodeType(jsonFile["keyType"]) + "=>"
406     valueType = getNodeType(jsonFile["valueType"]) + ")"
407     return keyType + valueType
408
409 def unaryOperation(jsonFile):
410
411     operator= jsonFile["operator"] + " "
412     subExpression = getNodeType(jsonFile["subExpression"])
413     return (operator + subExpression) if jsonFile["prefix"] ==
        True else subExpression + operator
414
415 def indexAccess(jsonFile):
416
417     baseExpression = getNodeType(jsonFile["baseExpression"])
418     indexExpression = "[" + (getNodeType(jsonFile["indexExpression
        "]) if "indexExpression" in jsonFile.keys() and jsonFile["
        indexExpression"] is not None else "")  + "]"
419     return baseExpression + indexExpression
420
421 def eventDefinition(jsonFile):
422
423     documentation = jsonFile["documentation"] if "documentation"
        in jsonFile.keys() else None
424     documentation = documentation if type(documentation)== str
        else (getNodeType(documentation) if type(documentation)==
        dict else None)  #for solidity before 6
425     documentation = "    /**\n        * " + documentation.replace(
        "\n","\n        * ") + "\n        */\n        " if not
        documentation is None else ""
426     name = "event " + jsonFile["name"]
427     parameters = getNodeType(jsonFile["parameters"])
428     anonymous = " anonymous;\n" if "anonymous" in jsonFile.keys()
        and jsonFile["anonymous"] else ";\n"
429     return documentation + name + parameters + anonymous
```

```
430
431 def emitStatement(jsonFile):
432
433     eventCall = "emit " + getNodeType(jsonFile["eventCall"]) + ";"
434     return eventCall
435
436 def usingForDirective(jsonFile):
437
438     libraryName = "using " + getNodeType(jsonFile["libraryName"])
        + " for "
439     typeName = (getNodeType(jsonFile["typeName"]) if "typeName" in
         jsonFile.keys() and jsonFile["typeName"] is not None else
        "*") + ";\n"
440     return libraryName + typeName
441
442 def forStatement(jsonFile):
443
444     initializationExpression = "for (" + (getNodeType(jsonFile["
        initializationExpression"]) if "initializationExpression"
        in jsonFile.keys() and jsonFile["initializationExpression"]
         is not None else ";") + " "
445     condition = (getNodeType(jsonFile["condition"]) if "condition"
         in jsonFile.keys() and jsonFile["condition"] is not None
        else "") + "; "
446     loopExpression = (getNodeType(jsonFile["loopExpression"]).
        replace(";","") if "loopExpression" in jsonFile.keys() and
        jsonFile["loopExpression"] is not None else "") +") "
447     body = getNodeType(jsonFile["body"])
448     return initializationExpression + condition + loopExpression +
         body
449
450 def tupleExpression(jsonFile):
451
452     components = commaBetweenElements(jsonFile["components"])
453     components = "[" + components + "]" if jsonFile["isInlineArray
        "] else "(" + components + ")"
454     return components
455
456 def newExpression(jsonFile):
457
458     typeName = "new " + getNodeType(jsonFile["typeName"])
459     return typeName
460
461 def typeBreak(jsonFile):
462
463     return "break;\n"
464
465 def conditional(jsonFile):
466
```

```
467     condition = getNodeType(jsonFile["condition"]) + " ? "
468     trueExpression = getNodeType(jsonFile["trueExpression"]) + " :
            "
469     falseExpression = getNodeType(jsonFile["falseExpression"])
470     return condition + trueExpression + falseExpression
471
472 def enumDefinition(jsonFile):
473
474     name = "enum " + jsonFile["name"] + " "
475     members = "{" + commaBetweenElements(jsonFile["members"]) + "
            }\n\n"
476     return name + members
477
478 def enumValue(jsonFile):
479
480     name = jsonFile["name"]
481     return name
482
483 def whileStatement(jsonFile):
484
485     condition = "while (" + getNodeType(jsonFile["condition"]) + "
            ) "
486     body = getNodeType(jsonFile["body"])
487
488     if body[-1] == "}":
489         body = body[:-1] + "    }"
490
491     return condition + body
492
493 def continueStatement(jsonFile):
494
495     return "continue;"
496
497 def tryStatement(jsonFile):
498
499     externalCall = "try \n        " + getNodeType(jsonFile["
            externalCall"])
500     clauses = ""
501     for i in range(0,len(jsonFile["clauses"])):
502         if i == 0 and "catch" in getNodeType(jsonFile["clauses"][i
                ]):
503             clauses += getNodeType(jsonFile["clauses"][i]).replace
                    ("catch","")
504         elif i != 0 and "returns" in getNodeType(jsonFile["clauses
                "][i]):
505             clauses += getNodeType(jsonFile["clauses"][i]).replace
                    ("returns","catch")
506         else:
507             clauses += getNodeType(jsonFile["clauses"][i])
```

```
508             if clauses[-1] == "}":
509                 clauses = clauses[:-1] + "    }"
510 
511         return externalCall + clauses
512 
513     def tryCatchClause(jsonFile):
514 
515         errorName = jsonFile["errorName"] + " " if "errorName" in
               jsonFile.keys() and jsonFile["errorName"] is not None else
               ""
516         parameters = getNodeType(jsonFile["parameters"]) if "
               parameters" in jsonFile.keys() and jsonFile["parameters"]
               is not None else ""
517         block = getNodeType(jsonFile["block"])
518 
519         return "\n        " + ("returns " + errorName + parameters +
               block if parameters != "" else ("catch" + errorName + block
               ))
520 
521     def doWhileStatement(jsonFile):
522 
523         body = "do " + getNodeType(jsonFile["body"])
524 
525         if body[-1] == "}":
526                 body = body[:-1] + "    }"
527 
528         condition = " while (" + getNodeType(jsonFile["condition"]) +
               ");"
529         return body + condition
530 
531     def identifierPath(jsonFile):
532 
533         name = jsonFile["name"]
534         return name
535 
536     def uncheckedBlock(jsonFile):
537 
538         listStatements = ""
539 
540         for elem in jsonFile["statements"]:
541             listStatements += "        " + getNodeType(elem) + "\n"
542 
543         listStatements = "unchecked {\n" + listStatements + "    }"
544         return listStatements
545 
546     def indexRangeAccess(jsonFile):
547 
548         baseExpression = getNodeType(jsonFile["baseExpression"])
```

```
549     startExpression = "[" + (getNodeType(jsonFile["startExpression
            "]) if jsonFile["startExpression"] is not None else "") + "
            :"
550     endExpression = (getNodeType(jsonFile["endExpression"]) if
            jsonFile["endExpression"] is not None else "") + "]"
551
552     return baseExpression + startExpression + endExpression
553
554 def getNodeType(jsonFile):
555
556     if (not "nodeType" in jsonFile.keys()):
557         print("the nodeType key doesn't exist")
558
559     else :
560         nodeType = jsonFile["nodeType"]
561         switcher = {
562         "SourceUnit": sourceUnit,
563         "PragmaDirective" : pragmaDirective,
564         "ContractDefinition" : contractDefinition,
565         "InheritanceSpecifier" : inheritanceSpecifier,
566         "FunctionDefinition" : functionDefinition,
567         "Block": block,
568         "ParameterList": parameterlist,
569         "VariableDeclarationStatement":
                variableDeclarationStatement,
570         "FunctionCall": functionCall,
571         "VariableDeclaration": variableDeclaration,
572         "Identifier": identifier,
573         "ModifierInvocation": modifierInvocation,
574         "Literal": literal,
575         "ModifierDefinition": modifierDefinition,
576         "Return": returnStatement,
577         "ExpressionStatement": expressionStatement,
578         "BinaryOperation": binaryOperation,
579         "OverrideSpecifier" : overrideSpecifier,
580         "UserDefinedTypeName" : userDefinedTypeName,
581         "ElementaryTypeName" : elementaryTypeName,
582         "PlaceholderStatement" : placeholderStatement,
583         "Assignment" : assignment,
584         "MemberAccess" : memberAccess,
585         "ArrayTypeName" : arrayTypeName,
586         "StructuredDocumentation": structuredDocumentation,
587         "InlineAssembly":inlineAssembly,
588         "ElementaryTypeNameExpression" :
                elementaryTypeNameExpression,
589         "FunctionCallOptions" : functionCallOptions,
590         "IfStatement":ifStatement,
591         "YulBlock" : yulBlock,
592         "YulAssignment" : yulAssignment,
```

```
593             "YulIdentifier" : yulIdentifier,
594             "YulFunctionCall" : yulFunctionCall,
595             "YulVariableDeclaration" : yulVariableDeclaration,
596             "YulTypedName" : yulTypedName,
597             "YulExpressionStatement": yulExpressionStatement,
598             "YulLiteral" : yulLiteral,
599             "YulSwitch" : yulSwitch,
600             "YulCase" : yulCase,
601             "YulIf" : yulIf,
602             "YulForLoop" : yulForLoop,
603             "YulFunctionDefinition" : yulFunctionDefinition,
604             "StructDefinition" : structDefinition,
605             "Mapping" : mapping,
606             "UnaryOperation" : unaryOperation,
607             "IndexAccess" : indexAccess,
608             "EventDefinition" : eventDefinition,
609             "EmitStatement" : emitStatement,
610             "UsingForDirective" : usingForDirective,
611             "ForStatement" : forStatement,
612             "TupleExpression" : tupleExpression,
613             "NewExpression" : newExpression,
614             "Break" : typeBreak,
615             "Conditional" : conditional,
616             "EnumDefinition" : enumDefinition,
617             "EnumValue" : enumValue,
618             "WhileStatement" : whileStatement,
619             "Continue" : continueStatement,
620             "TryStatement" : tryStatement,
621             "TryCatchClause" : tryCatchClause,
622             "DoWhileStatement" : doWhileStatement,
623             "IdentifierPath" : identifierPath,
624             "UncheckedBlock" : uncheckedBlock,
625             "IndexRangeAccess" : indexRangeAccess
626             #ImportDirect is handled by orderToImport function
627             }
628             func = switcher.get(nodeType)
629 
630         if (func is None):
631             print("!!!!!\n")
632             print("!!!!!\n")
633             print("Flattener Error for type: " + nodeType)
634             print("!!!!!\n")
635             print("!!!!!\n")
636 
637         else:
638             return func(jsonFile)
639 
640 """
641 Functions to help the writing
```

```
642 """
643
644 def splitAstElements(astToSplit):
645     newlist = []
646     totalBraces = 0
647
648     for i in range(0, len(astToSplit)):
649         line = astToSplit[i]
650         lineBraces = line.count("{") - line.count("}")
651         if (lineBraces > 0 and totalBraces ==0):
652             newlist.append(line)
653         if (totalBraces > 0):
654             newlist[-1] += line
655         totalBraces+=lineBraces
656
657     for i in range(0, len(newlist)):
658         newlist[i]= json.loads(newlist[i])
659
660     return newlist
661
662 def commaBetweenElements(jsonFile):
663
664     elements=""
665
666     if jsonFile is not None:
667         for i in range(0,len(jsonFile)):
668             elements += getNodeType(jsonFile[i]) if jsonFile[i] is
                    not None else ""
669             if i < len(jsonFile)-1:
670                 elements+= ", "
671
672     if len(elements)>79:
673         elements = ""
674         if jsonFile is not None:
675             for i in range(0,len(jsonFile)):
676                 elements += "\n                " + getNodeType(
                        jsonFile[i]) if jsonFile[i] is not None else ""
677                 if i < len(jsonFile)-1:
678                     elements+= ","
679                 else:
680                     elements+= "\n"+"        "
681
682     return elements
683
684 def orderToImport(absolutePath,listFiles):
685
686     importedContracts = [absolutePath]
687
688     for jsonFile in listFiles:
```

```
689             if absolutePath == jsonFile["absolutePath"]:
690                 for elem in jsonFile["nodes"]:
691                     if elem["nodeType"]=="ImportDirective":
692                         importedContracts = orderToImport(elem["
                            absolutePath"],listFiles) +
                            importedContracts
693
694         return list(dict.fromkeys(importedContracts)) #delete
                duplicates
695
696     def generateFile(order, listFiles):
697
698         nodes = ""
699
700         for i in range(0,len(order)):
701             for elem in listFiles:
702                 if elem["absolutePath"] ==  order[i]:
703                     if i==0 and elem["nodeType"]=="SourceUnit":
704                         nodes += sourceUnit(elem)
705                     for node in elem["nodes"]:
706                         if node["nodeType"] != "ImportDirective":
707                             nodes += getNodeType(node)
708
709         if "pragma solidity ^0.5" in nodes or "pragma solidity 0.5" in
                nodes:
710             nodes = nodes.replace("fallback","function")
711
712         return nodes
713
714     def main():
715         parser = argparse.ArgumentParser(description="Contract
                flattener. It mixes several Solidity files into one and
                manages the relations between them. The contract must be
                compilable by your current version of solc to be flattened.
                ")
716         parser.add_argument("file", help="Specifies main file that
                contains the imports")
717         args = parser.parse_args()
718         arguments = ["solc", "--ast-compact-json", args.file]
719         proc = subprocess.run(arguments,stdout=subprocess.PIPE,
                universal_newlines=True)
720         name = args.file.split("/")[-1]
721
722         if(proc.returncode != 0):
723             print("Error with solc compiler on this contract")
724             sys.exit(1)
725         else:
726             fw = open("flattened_"+name,"w")
727             listFiles =splitAstElements(proc.stdout.splitlines())
```

```
728             fw.write(generateFile(orderToImport(args.file,listFiles),
                    listFiles))
729             fw.close()
730
731 if __name__ == '__main__':
732     main()
```

Bibliography

[1] Stuart Haber and W Stornetta. *How to Time-Stamp a Digital Document, Crypto'90, LNCS 537*. 1991.

[2] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.

[3] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[4] Frederik Armknecht et al. "Ripple: Overview and outlook". In: *International Conference on Trust and Trustworthy Computing*. Springer. 2015, pp. 163–180.

[5] Solidity Documentation. *Solidity Grammar*. URL: `https://docs.soliditylang.org/`.

[6] Cryptopedia staff. *The DAO Hack Explained*. URL: `https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-the-dao-hack`.

[7] Dylan Yaga et al. "Blockchain technology overview". In: *arXiv preprint arXiv:1906.11078* (2019).

[8] Michael Nofer et al. "Blockchain". In: *Business & Information Systems Engineering* 59.3 (2017), pp. 183–187.

[9] Baocheng Wang et al. "Large-scale election based on blockchain". In: *Procedia Computer Science* 129 (2018), pp. 234–237.

[10] Sébastien Lugan et al. "Secure architectures implementing trusted coalitions for blockchained distributed learning (TCLearn)". In: *IEEE Access* 7 (2019), pp. 181789–181799.

[11] Shailendra Rathore, Yi Pan, and Jong Hyuk Park. "BlockDeepNet: a Blockchain-based secure deep learning for IoT network". In: *Sustainability* 11.14 (2019), p. 3974.

[12] Nathan Reiff. *Differences Between Ethereum and Bitcoin*. June 16, 2020. URL: `https://www.investopedia.com/articles/investing/031416/bitcoin-vs-ethereum-driven-different-purposes.asp`.

[13] S.Richards et al. *Introduction to Smart-Contracts*. Mar. 8, 2021. URL: https://ethereum.org/en/developers/docs/smart-contracts/.

[14] S.Richards et al. *Transaction in Ethereum Blockchain*. Nov. 30, 2020. URL: https://ethereum.org/en/developers/docs/transactions/.

[15] Fahad Saleh. "Blockchain without waste: Proof-of-stake". In: *The Review of financial studies* 34.3 (2021), pp. 1156–1190.

[16] Ethereum Documentation. *Nodes and Clients*. URL: https://ethereum.org/en/developers/docs/nodes-and-clients/.

[17] *Ethereum Virtual Machine*. URL: https://www.bitrates.com/guides/ethereum/what-is-the-unstoppable-world-computer.

[18] *Vinted*. URL: https://en.wikipedia.org/wiki/Vinted.

[19] Julian Cohen. "Contemporary Automatic Program Analysis". In: *Black Hat. Las Vegas* (2014).

[20] Martin Ouimet and Kristina Lundqvist. "Formal software verification: Model checking and theorem proving". In: *Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA* (2007).

[21] E Allen Emerson. "Temporal and modal logic". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 995–1072.

[22] Rob Gerth et al. "Simple on-the-fly automatic verification of linear temporal logic". In: *International Conference on Protocol Specification, Testing and Verification*. Springer. 1995, pp. 3–18.

[23] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[24] Fabio Somenzi and Roderick Bloem. "Efficient Büchi automata from LTL formulae". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 248–263.

[25] Gul Agha and Karl Palmskog. "A Survey of Statistical Model Checking". In: *ACM Transactions on Modeling and Computer Simulation* 28 (Jan. 2018), pp. 1–39. DOI: 10.1145/3158668.

[26] Axel Legay, Benoît Delahaye, and Saddek Bensalem. "Statistical model checking: An overview". In: *International conference on runtime verification*. Springer. 2010, pp. 122–135.

[27] Alexandre David et al. "Uppaal SMC tutorial". In: *International Journal on Software Tools for Technology Transfer* 17.4 (2015), pp. 397–415.

[28] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[29] Brian Chess and Gary McGraw. “Static analysis for security”. In: *IEEE security & privacy* 2.6 (2004), pp. 76–79.

[30] Barton P Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: (1989).

[31] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[32] Eduard Baranov, Thomas Given-Wilson, and Axel Legay. “Improving secure and robust patient service delivery”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2020, pp. 404–418.

[33] B. Jiang, Y. Liu, and W. K. Chan. “ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 259–269.

[34] Loi Luu et al. “Making smart contracts smarter”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.

[35] Petar Tsankov et al. “Securify: Practical security analysis of smart contracts”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 67–82.

[36] Sergei Tikhomirov et al. “Smartcheck: Static analysis of ethereum smart contracts”. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 9–16.

[37] W. K. Chan and B. Jiang. “Fuse: An Architecture for Smart Contract Fuzz Testing Service”. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 707–708.

[38] Mark Mossberg et al. “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1186–1189.

[39] I. Ashraf et al. “GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities”. In: *IEEE Access* 8 (2020), pp. 99552–99564.

[40] Sukrit Kalra et al. “ZEUS: Analyzing Safety of Smart Contracts.” In: *NDSS*. 2018.

[41] Anastasia Mavridou et al. “VeriSolid: Correct-by-design smart contracts for Ethereum”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 446–465.

[42] Solidity’s Documentation. *SMT Checker*. URL: `https://docs.soliditylang.org/en/v0.6.8/layout-of-source-files.html`.

[43] Solidity’s Documentation. *Security Considerations*. URL: `https://docs.soliditylang.org/en/v0.6.8/security-considerations.html`.

[44] Eric Rafaloff. *Solidity Model Checker*. June 16, 2020. URL: `https://www.aon.com/cyber-solutions/aon_cyber_labs/exploring-solidity-model-checker/`.

[45] Clark Barrett and Cesare Tinelli. “Satisfiability modulo theories”. In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[46] Daniel Perez and Benjamin Livshits. “Smart contract vulnerabilities: Does anyone care?” In: *arXiv preprint arXiv:1902.06710* (2019).

[47] Lucianna Kiffer, Dave Levin, and Alan Mislove. “Stick a fork in it: Analyzing the Ethereum network partition”. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 2017, pp. 94–100.

[48] Leonardo Alt and Christian Reitwiessner. “SMT-based verification of solidity smart contracts”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 376–388.

[49] Matteo Marescotti et al. “Accurate Smart Contract Verification Through Direct Modelling”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2020, pp. 178–194.

[50] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. “On solving universally quantified horn clauses”. In: *International Static Analysis Symposium*. Springer. 2013, pp. 105–125.

[51] Chen Chung Chang and H Jerome Keisler. *Model theory*. Elsevier, 1990.

[52] Petar Tsankov. *Github repository : Securify v2.0*. 2020. URL: `https://github.com/eth-sri/securify2`.

[53] Petar Tsankov. *Release of Securify v2.0*. Jan. 23, 2020. URL: `https://medium.com/chainsecurity/release-of-securify-v2-0-6304a40034f`.

[54] Peter Z Revesz. “Safe stratified Datalog with integer order programs”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1995, pp. 154–169.

[55] Gustavo Grieco et al. “Echidna: effective, usable, and fast fuzzing for smart contracts”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 557–560.

[56] Josselin Feist, Gustavo Grieco, and Alex Groce. “Slither: a static analysis framework for smart contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. 2019, pp. 8–15.

[57] Rupak Majumdar and Koushik Sen. “Hybrid concolic testing”. In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 416–426.

[58] Joel Jones. “Abstract syntax tree implementation idioms”. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[59] Solidity's grammar. *Solidity Grammar*. URL: `https://docs.soliditylang.org/en/v0.8.4/grammar.html`.

[60] Henrique Rocha et al. “Solidity parsing using smacc: Challenges and irregularities”. In: *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*. 2017, pp. 1–9.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**