Conkas: A Modular and Static Analysis Tool for Ethereum Bytecode

Nuno Veloso
nuno.veloso@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

January 2021

Abstract

Since the beginning of Ethereum, started the development of tools to aimed developers to not introduce bugs with static analysis tools that show warnings to developers if they introduce some bugs. However, vulnerabilities exist as shown by TheDAO attack, a very famous attack that stole a lot of Ether. To try to reduce the probability of new future attacks of happening we introduce Conkas, a modular static analysis tool that use symbolic execution to find traces that lead to vulnerabilities and uses an intermediate representation (IR). The users can interact with Conkas via Command-Line Interface (CLI) and the output will be the result of the analysis. Conkas supports Ethereum bytecode or contracts written in Solidity and is compatible with all version of Solidity, but the analysis is done at the bytecode level. Conkas support already 5 modules that detect vulnerabilities related to DASP10 categories, being Arithmetic, Front-Running, Reentrancy, Time Manipulation and Unchecked Low-Level Calls. Conkas is also easy to extend, meaning that you can add your custom modules to detect other types of vulnerabilities. We analyse Conkas with SmartBugs and our tool has the best precision, with 54%.
Keywords: Ethereum Virtual Machine, Blockchain, Smart Contracts, Static Analysis, Symbolic Execution

1. Introduction

Nowadays the technology revolution is constant in such a way that is possible to obtain solutions that some years ago are unthinkable. Blockchain exists since some years ago and is an innovative technology. Later, in 2009, Satoshi Nakamoto introduces Bitcoin [16], a crypto-coin that not rely on third-party trust. Thanks to Blockchain, Bitcoin gain a lot of popularity because it ensures consensus with all participants in the network. This distributed consensus is ensured with an algorithm called Proof-of-Work (PoW) that also make a Sybil-attack[1] unworkable [4]. Ethereum allows any user to create a program, called Smart Contract, and run in this distributed network. Ethereum also has a cryptocurrency called Ether (Eth), and those Smart Contracts can send and receive Ether between each other via transactions. Once a Smart Contract is deployed, the owner of that contract can no longer make updates to fix it if someone finds a vulnerability.

Smart Contracts are programs that are executed by nodes or peers in the network (blockchain). Nick Szabo describe a Smart Contract as a vending machine that shares exactly the same properties as a Smart Contract in blockchain [21]. A vending machine has hard-coded some rules that define the behaviour in certain conditions and execute some actions based on those conditions. Smart Contracts can implement a series of applications like financial, insure, etc. Ethereum was the first platform in blockchain to implement a virtual machine Turing-Complete [24]. These contracts are written in a language like Solidity [3], the most used nowadays, and are compiled to Ethereum bytecode that will be executed at Ethereum Virtual Machine (EVM).

Smart Contracts are far from vulnerabilities free and some of it was exploited in the past, leading to a loss in terms of money, like TheDAO attack [5]. In Ferreira et al. [8] was created a manually annotated dataset with vulnerable contracts and the results show that none of the tools was able to find 50% of the vulnerable contracts. If someone would like to analyse some smart contract in order to see if it has some vulnerability, the probability of finding a true positive is low as shown by Ferreira et al..

This paper present Conkas, a modular and static

[1] A Sybil-attack consists of a participant playing several identities simultaneous, and the network can have more false identities than real identities, allowing the participant to take control of the network.

analysis tool based on symbolic execution that tries to find a trace that leads to some vulnerability. It is also easy to add custom modules to detect new types of vulnerabilities. The tool provides a Command-Line Interface (CLI) where users can interact with Conkas. The tool supports smart contracts written in solidity or just the Ethereum bytecode. We describe the architecture of our tool, how easy is to add more modules to detect new types of vulnerabilities, how easy is to add more EVM instructions that can emerge with new updates from Ethereum in order to keep updated our tool and describe the 5 modules that Conkas already support. These modules are based on DASP10 [9] and can detect vulnerabilities of the following categories: Arithmetic, Front-Running, Reentrancy, Time Manipulation and Unchecked Low-Level Calls. Conkas uses an intermediate representation (IR) which is Rattle [20] but we made some modifications to make it even stronger and fit our needs. We will describe those modifications as well.

2. Background

In this section, we introduce Blockchain technology, Ethereum, Smart Contracts and Ethereum Virtual Machine, how they act and work. Then we introduce the 5 types of categories that Conkas already support, based on DASP10, which is Arithmetic, Front-Running, Reentrancy, Time Manipulation and Unchecked Low-Level Calls.

2.1. Blockchain

Blockchain is replicated by all nodes. It is organized like a chain of block's hash ordered by time, where each block has several fields. Simplifying it can be seen like a database or a linked-list belonging to a specific network where each block only has a reference to the previous block. When a block is added it can never be updated nor removed from this database or linked-list, being immutable. Exists permissionless networks, which means that that network is public because there is no need for permissions to be part of that network. In the other side, permissioned networks are private networks, meaning that if a user wants to join, he needs permission to join. Each block to be added to the blockchain needs to be mined by a miner and the miner needs to satisfy some restrictions imposed by the network to add successfully that block. In each epoch, each miner proposes a block to add to the blockchain, and in that block are present a list of transactions that this miner include in that block. To a miner successfully add a block to the Ethereum network he needs to perform a Proof-of-Work (PoW). If the miner finds a solution that satisfies the PoW, this miner broadcast the block to the other miners and the other miners need to verify whether the solution proposed is correct in order to add to their blockchain. This protocol is called Nakamoto consensus. For more detail the reader can read the Bitcoin paper [16] or the Ethereum paper [4].

2.2. Ethereum

Ethereum was created by Vitalik Buterin in 2014 and formally written by Gavin Wood [24]. It is a decentralized platform that sits above blockchain technology. This network is permissionless and what makes it different from Bitcoin is that it supports Smart Contracts that are executed in Ethereum Virtual Machine (EVM).

The state of the Ethereum is equal to the last block added to the blockchain and it consists of the mapping of addresses to accounts. It is like a dictionary where the key is an address (of 160 bits) and the value is the account. Being σ the last state of the blockchain and γ an address of an account, then $\sigma[\gamma]$ gives us the last state of that account at address γ.

Ethereum supports contract accounts which contain cryptocoins, code to be executed and persistent memory. It also supports externally owned account which only contains cryptocoins and is controlled by a private key. A user that wants to transfer some cryptocoins or interact with a contract needs to make a transaction.

2.3. Smart Contracts

Smart Contracts are an autonomous agent stored in the blockchain. It is identified by an address of 160 bit and the code that represents the smart contract can manipulate variables like a traditional programming language. These smart contracts cannot be updated being extremely important that they do not have any bugs before they are deployed.

The most common language to write Smart Contracts is Solidity, an object-oriented programming language, influenced by many languages like C++, Python and Javascript. The code is compiled to be executed in EVM.

2.4. Ethereum Virtual Machine

EVM executes code present in Smart Contracts written in a Turing-complete language and with that exists the problem of non-termination. To mitigate this problem, the creator of Ethereum introduced the gas concept. Each EVM instruction has a pre-defined price, meaning the necessary gas to execute that instruction. When a user sends a transaction to invoke a contract, this user needs to specify the amount of gas that is willing to provide, called gasLimit. It also needs to provide what is the price by a unit of gas, called gasPrice. If the execution runs out-of-gas the execution is aborted with an exception and the state are reverted to the initial state.

The EVM operates in pseudo-registers of 256 bit, meaning that it does not operate under registers but under an expandable stack that is used to pass arguments to functions and other opcodes. The stack uses 256-bit values and has a maximum of 1024 entries, being volatile. Memory is a byte-array structure and can be read or written starting from any index and arbitrary length. It is initialized with 0 size and can only be extended, being volatile also. Storage is a dictionary of 256 bit to 256 bit, and it is persistent.

2.5. Known Vulnerabilities
In this subsection, we introduce the five categories of vulnerabilities that Conkas detect and we also show a way of mitigating when possible.

2.5.1 Reentrancy

This vulnerability is the most known because of the TheDAO attack [5]. In Ethereum, when a contract invokes another, the execution of the contract waits until the other contract finish to remain the execution. An attacker can take profit of this intermediate state. The first way that exists to mitigate this type of vulnerability is to avoid the call function and use the send function or the transfer function. These two function restricts the gas of the execution to 2300, a predefined value. With this amount of gas, any contract that calls another will abort with a RunOutOfGas exception, reverting the transaction. However, this is not true anymore due to an update of Ethereum that change some gas associated with each instruction [13]. Other mitigations are to use the Checks-Effects-Interactions pattern or a guard pattern.

2.5.2 Arithmetic

Arithmetic vulnerabilities are integer overflow/underflow and occur when a variable cannot hold a specific value and store another. This type of vulnerability is well-known from other contexts and it is the same in EVM context. One way to mitigate this type of vulnerability is to use a well-known library to add, subtract, etc (e.g. SafeMath[2]).

2.5.3 Unchecked Low-Level Calls

In Ethereum exists several ways to a contract invoke another, by several instructions, for example, CALL, CALLCODE, DELEGATECALL and STATICCALL. These instructions are considered low-level instructions. When a user writes a contract, the user will use higher-level functions to call another contract like send or transfer. However, send does not propagate exceptions, only return false when an exception occurs. If the callee execution aborted with an exception, and the user uses the send function and does not check the return value, the caller execution will continue as nothing happened. To mitigate this type of vulnerability is recommended to use the transfer function or when using other functions or directly use low-level instructions, check the return value in order to handle exceptions.

2.5.4 Front-Running

This vulnerability is also known as Transaction Order Dependence (TOD). It happens when exists two transactions (T_1 and T_2) and each of them invokes the same contract. Transaction T_1 can occur in state σ or in state σ' if transaction T_2 occur first and results in that state σ'. This order is chosen by a miner. A malicious user can take profit with this. Imagine a case where a user submits a solution to a puzzle, the malicious user can see the solution and he can create another transaction with the solution saw and provide a much higher gas in order to be mined first, receiving the award. There is no known way to mitigate this type of vulnerability because the order of transactions is dependent on miners. The developers need to take care of this.

2.5.5 Time Manipulation

The contracts may need to have timely information, and this information can be obtained by a variable which is *block.timestamp*. This variable just reads the information present in the block where the transaction is found. However, this information is controlled by miners and developers should avoid using it. In Ethereum specification (yellow paper) there is none restriction about this value, it only mentions that it needs to be higher than the previous block. In Ethereum Geth[3] and Parity[4] implementations, both reject times that deviates 15 seconds from the actual time. There is no known form to mitigate this type of vulnerability because the miners control this information, the only way is for developers to avoid the use of *block.timestamp.*

3. Related Work

We studied tools that use symbolic execution and 3 tools that use an IR. We studied Oyente [12], a well-known tool in this field. They analyse bytecode and start to construct a CFG and then iterate over all possible path generating traces. These traces are

[2] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol`

[3] `https://github.com/ethereum/go-ethereum/blob/b71334ac3de38338e618aaf8ea6b4a884d2d80f5/consensus/ethash/consensus.go#L46`

[4] `https://github.com/openethereum/openethereum/blob/ee2b16dfe4895c19c052322a588878e447a13706/ethcore/src/verification/verification.rs#L392`

then passed to a module that makes the analysis. They use Z3 to reduce the search space. Oyente can detect vulnerabilities of type Transaction Order Dependence (TOD), Timestamp Dependence, Mishandled Exceptions and Reentrancy. There are a lot of other tools that are based on Oyente [17, 2, 22, 11]. We studied another symbolic executing tool called Mythril [15]. It constructs a CFG and in this, it makes their symbolic analysis. Mythril uses Z3 as well to prune the search space. Manticore [14] is another symbolic execution tool that is capable of doing their analysis at x86/x64 and ARM binaries and on Smart Contracts.

Zeus [10] is a tool that uses an IR. It takes the solidity code and policies as input and it will check whether the solidity code meets these policies. The solidity code is converted to LLVM bitcode that act as an IR and the assertions of the policies are introduced. Then it invokes a verifier to determine whether the assertions are violated. Securify [23] creates their own IR. It takes as input the bytecode and is transformed into Single Static Assignment (SSA). Then it makes a static analysis in all paths and creates Datalog facts. Then, these Datalog facts are evaluated to check whether the violation patterns are violated. Slither [7] also creates its own IR. It receives as input the Abstract Syntax Tree (AST) and recovers some information, like the CFG, contract inheritance and a list of solidity expressions. Then it converts to SlithIR [6], a custom IR, that uses SSA form. Later, it analyses the code looking for vulnerabilities.

We also studied other intermediate representations like Rattle [20] that we used in our tool. Beyond that, SlithIR does not fit our needs because it needs the source code. The solidity authors also have their IR, YUL [19] but it has less information than Rattle and does not use instructions in SSA form. EthIR [1] is another IR that is an extension of Oyente tool and produce a rule-based representation (RBR).

4. Conkas

In this section, we introduce Conkas a modular static analysis tool that uses the symbolic execution model. We describe Rattle and our modifications in order to use as our Intermediate Representation (IR). We will describe our architecture and how easy is to add more modules to implement custom policies. Conkas will be publicly available on GitHub [5].

4.1. Rattle

Rattle[6] is developed by Ryan Stortz and is a tool designed to work with Smart Contracts already deployed in Ethereum blockchain, working with ethereum bytecode. Rattle constructs a Control Flow Graph (CFG), creates an Intermediate Representation (IR) in the form of Static Single Assignment (SSA) and instructions are converted from stack form to a register form. With this transformation, Rattle can remove several instructions like DUP's, SWAP's, PUSH's and POP's.

This tool uses a disassembler from Manticore's tool and we change it to a library called pyevmasm[7] in order to be up to date with new instructions. The other modification was to change the function that removes the swarm hash, which is a hash appended to the bytecode. The old function generated errors when a swarm hash has a different signature than the expected. The swarm hash can have several signatures and new ones can emerge, so we modify it to check the length of the swarm hash and remove it. We also add more restrictions to other functions because it was missing and sometimes we got errors. When we have the instructions in SSA form we sometimes end up with PHI instructions. Rattle removes the PUSHs instructions and by that, we did not know in which path the values are created, so we modify Rattle to keep some PUSHs instructions only when these instructions are related to PHI instructions. We also update some jump locations when intermediate blocks were removed due to some optimizations. We also add a variable to one class to have the necessary information to later do the mapping between instructions to line numbers in Solidity file. We made also minor changes that can be seen in our fork of Rattle[8].

4.2. Conkas' Architecture

Conkas uses a modular architecture and can be seen in Figure 1.

Conkas makes available a Command-Line Interface (CLI) where users can interact with it. The users can provide the bytecode associated with a contract that they want to analyse or they can provide the Solidity source code. When source code is provided this will be compiled using a compiler that best suits. At the moment, it is only possible to provide source code written in Solidity. The bytecode will be passed to Rattle module.

In Rattle module it will elevate the bytecode to an IR and instructions will be converted to SSA form and converted from a stack form to a register form. The CFG will also be constructed. These artefacts will be passed to the next module which is Symbolic Execution Engine.

The Symbolic Execution Engine is responsible to iterate over CFG and generate traces. A trace is a possible path to be executed and contains information like the register, memory, storage, return

[5] https://github.com/nveloso/conkas
[6] https://github.com/crytic/rattle
[7] https://github.com/crytic/pyevmasm
[8] https://github.com/nveloso/rattle

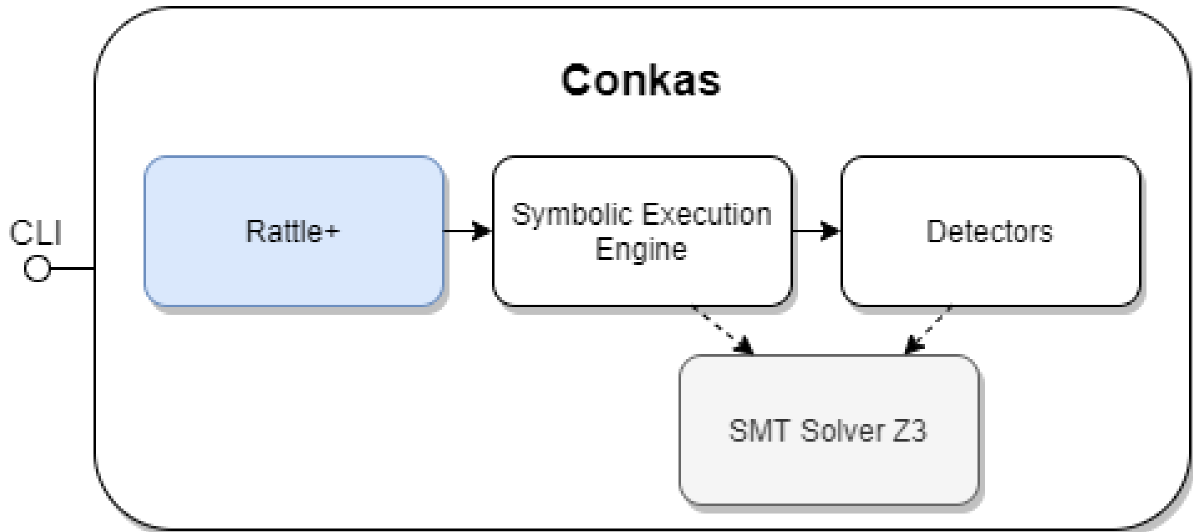

Figure 1: Architecture of Conkas tool. In Blue the modified module already existing. In grey the module already existing.

value, depth hit, restrictions to follow this path, the context where it is executed and information if the transaction was reverted, stopped, if it is destroyed or hit an invalid instruction. When it hits a conditional JUMP the engine will always iterate first over the false condition. To prevent that the engine will run forever exists the concept of depth, and if a max depth was hit it will stop the execution of that path. This module creates symbolic variable and symbolic expressions using the module SMT Solver Z3.

Those traces generated by the Symbolic Execution Engine will be provided to Detectors module. In this module exist sub-modules, one responsible to detect a category of vulnerability. Conkas support 5 categories, Arithmetic, Front-Running, Reentrancy, Time Manipulation and Unchecked Low-Level Calls. Each sub-module returns the vulnerabilities founded to the user. This module also queries STM Solver Z3 to check if some constraints are possible or not. It can also create new constraints and asks Z3 to verify whether the constraint is possible or not in order to determine whether a vulnerability exists or not.

4.3. System Requirements

Conkas is written in Python3 and needs an environment with Python3 and the following modules: cbor2, py-solc-x, pycryptodome, pyevmasm, solidity_parser and z3-solver. There is also a docker image available.

4.4. Supported Vulnerability Modules

The submodule that is responsible to check vulnerabilities of type arithmetic is capable of detect integer overflow and underflow. The overflows are detected at instructions ADD and MUL and the underflows are detected at instruction SUB. There are a lot of benign overflows and underflows and to identify those benign vulnerabilities, we made a heuristic.

Reentrancy vulnerabilities can occur when exists a CALL instruction in which the storage variable that holds Ether to send is different from zero at the moment of the CALL instruction or when exists a write to storage that holds Ether after sending it. We need to check if CALL instruction can be invoked again and for that, we check if the previous restrictions before CALL instruction can be true again with the updated values at CALL instruction, and if it is true, it is vulnerable. But this is not enough. We can have a case where we have a storage counter that is incremented in every call and a stack variable is created with counter value and at the end of the function, it checks if this stack variable is equal to the counter variable and if not it must revert in order to be considered not vulnerable. For that we need to check for a condition at the end of the function and see if the one value is a stack value and the other is a storage value and check if they have the same value. When they are different, the transaction needs to revert to be considered safe, otherwise is false. The policy for this submodule is not exactly what we described above because the symbolic variables have no information about where it belongs.

The policy created to detect Time Manipulation vulnerabilities is to verify if exists constraints in which an operand is based on time. The only instruction that is based on time is the TIMESTAMP instruction. We need also check the return value of a function, if some storage variable is set to a vari-

able based on time or if it is an input to SHA3 instruction.

Front Running policy only considers Timestamp Order Dependence (TOD), and we check if we have a function that can change a storage variable and that variable is used in a CALL instruction. We also check if we have another function that just uses this storage value in a CALL instruction.

The last policy created is to detect Unchecked Low-Level Calls and is straightforward. We need to check if the return value of the CALL instruction is present in the constraints of that trace.

4.5. How easy is to add new modules

Conkas was designed for a developer with basic knowledge of programming might add easily new modules to detect other types of vulnerabilities. This way, the only thing that he needs to do is to add an entry to the object inside the file __init__.py inside vuln_finder folder, being the key the name of the module and the value the function to be called. This function needs to follow the signature as shown in Listing 1.

```
1 def vuln_x_analyse(traces: [Trace], find_all: bool) -> [Vulnerability]:
2     pass
```

Listing 1: Function's signature to add new modules

This function receives the traces generated by the Symbolic Execution Engine and a boolean that when it is true it means that the module should find all vulnerabilities and not to stop when it finds one. As a return value, the function should return a list of Vulnerability instances.

4.6. How easy is to add new EVM instructions

In order to keep Conkas up to date, it is necessary to be easy to add new EVM instruction that can emerge with Ethereum updates. To add new EVM instructions the logic is the same as mentioned in subsection 4.5. The object to add the function responsible to symbolically execute is in file __init__.py at sym_exec/instruction folder. The function signature can be seen in Listing 2.

```
1 def inst_x(instruction: SSAInstruction, state: State) -> [SSABasicBlock]:
2     pass
```

Listing 2: Function's signature to add new instructions

The first argument is an instance of SSAInstruction and the second argument is an instance of State and can return a list of instances of type SSABasicBlock.

4.7. Symbolic Memory Access

In symbolic execution exists a problem that is the symbolic access of a data structure like the indexes and/or the length. In EVM some instructions can read or write in memory and receive as argument the index and the length. If index and/or length are symbolic it is not trivial to execute that. In both cases, the tool will approximate the solution. The approach done was when the length is symbolic, the instruction returns a new symbolic variable, like in [11]. If the length is concrete but the index is symbolic then the value read or stored is divided into bytes and the index will be each symbolic byte added 1 until reach the length.

4.8. Unit Tests

Unit tests give some guarantees when something changes in the source code. Conkas has several unit tests for all EVM instructions that Conkas support. For each EVM instruction exists on average 2 unit tests, with a total of 194 unit tests. Beyond that, we write 14 contracts written in Solidity to test our modifications of Rattle, see subsection 4.1. These tests were verified manually, one by one, to check that Rattle works as expected and make more robust.

4.9. Command-Line Interface (CLI)

Conkas makes available a Command-Line Interface (CLI) that allow users to interact with Conkas. Users must provide a file with Ethereum bytecode of a contract that the user want to analyse or the source code of the contract written in Solidity, however, the analysis is done at the bytecode level. It is possible for the user to specify which categories of vulnerability he wants to detect and has a flag to specify if he wants to stop as soon as Conkas finds a vulnerability or not. Users can define the logging level and the max depth to limit the search space. Users can also specify a value for timeout that will be used in Z3 when Conkas queries it to determine if some restrictions are satisfiable or not. To see the help message users can invoke Conkas like this:

```
$ python3 conkas.py -h
```

An example of an invocation that will analyse only reentrancy vulnerabilities and will find for all reentrancy vulnerabilities that is in some_file.evm file is as follow:

```
$ python3 conkas.py -vt reentrancy -fav some_file.evm
```

4.10. Known Limitations

Conkas have a known limitation that inherited from Rattle which is it cannot look for contracts that depend on libraries, even if these dependencies are declared in the same file as the contract.

5. Results

In this section, we describe and discuss the results of the comparison of other static analysis tools with Conkas. We can analyse in two ways. The traditional way, which is, each tool creates its own dataset, extracting contracts from Etherscan[9] between some dates, and run all tools against that dataset and show the results. Another way is to contact the authors and request for those datasets, as been done in Pérez and Livshits [18]. However, we believe that this is not the way it should be done and we use a publicly available framework for the dataset to be equal to every tool. We use SmartBugs [8] to do our analysis. We easily added Conkas to this framework and started to do the analysis with 10 different tools. SmartBugs has two different datasets, one is much bigger than the other. The one that we used is the smaller one because the vulnerabilities are manually annotated.

5.1. True Positive Rate

The script used to generate the comparison results gathered by each tool only count a true positive when a tool indicates the correct vulnerability category and the line number where the vulnerability occur in the source code. In Table 1 are shown the results with this criterion. The results show that is visible that most of the tools do not detect much of vulnerabilities. Conkas is the only tool that is above 50% and the Smartcheck and Slither are the most closer tools with 42% and 40% respectively. The others have a low percentage of true positives. We conclude that these results could be better if we extend the line number interval because one problem we faced when developing Conkas was to give the correct line number.

In order to have a comparison a little much fair and to not penalize other tools with such a low success rate we modified the criterion a little bit and now for a vulnerability be considered a true positive each tool must indicate the correct vulnerability category and the line number where the vulnerability occurs in source code with an interval of -5 to +5. This criterion is used for the rest of this paper. A new version of the Table 1 is shown in Table 2.

In Table 2 the results are a little bit better for some tools. Honeybadger, Manticore and Mythril were the most that benefit with this new criterion. Conkas is still the best tool compared to all others. Smartcheck is the tool that detects the higher number of different categories having an advantage related to others. The tools that detect the same number of different categories are Manticore and Mythril, detecting 5 different categories. Conkas detects the double of vulnerabilities compared to Manticore and detects 6% more than Mythril. The delta line is the difference between the percentage by each tool of the Table 1 with Table 2. Conkas detects more 3 vulnerabilities because when the vulnerability is in the return value, the line number returned is at the function declaration.

5.1.1 What is missing to get 100%?

It was possible to assess that there are some contracts in Arithmetic category that are annotated with vulnerabilities, however, when these contracts are compiled, those vulnerabilities are gone because the vulnerability is in dead code. There are 4 vulnerabilities of this type, so Conkas cannot reach 100% precision because of this.

The low precision in Front Running category is due to the fact that this category has several subcategories, and we just look for Transaction Order Dependence (TOD) which is only one subtype.

Concerning Reentrancy category, Conkas sometimes report the line number incorrectly because we retrieve the line number associate with the CALL instruction. If a function calls another and the vulnerability only happens when the execution came from this path we will retrieve the line number in the callee function, but it should be in the caller function. It happens 2 times. Exists one contract that has a library dependency, so we cannot analyse that contract and that contract has 2 vulnerabilities annotated.

Conkas does not have 100% precision in Unchecked Low-Level Calls because sometimes we reach the maximum depth allowed and miss some vulnerabilities. We do not extend this value because it will increase the time necessary to analyse, but it can be done if it is the only contract to analyse. We faced others contracts that eliminate CALL instructions when they are compiled, removing the vulnerabilities. We cannot measure how many contracts this happen because this sub dataset is much bigger than the others.

5.2. False Positives

In Table 3 are shown the false positives for each tool and for each category. Conkas is the tool that has the higher number of false positives. However, we checked the dataset of SmartBugs and conclude that are some vulnerabilities that exist and are not annotated. We retrieve the files that Conkas report false positive as well from Mythril, Slither and Smartcheck to manually verify if the false positives are real or are true positives. We selected just these tools because they have the best true positive rate, above 40%. We chose randomly based on the hash of the name of the file of those files. We chose the MD5 hash function because we hope it will be as uniform as possible. The files selected are the ones

[9] `https://etherscan.io/`

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/24 0%	0/24 0%	0/24 0%	0/24 0%	4/24 17%	0/24 0%	0/24 0%	0/24 0%	6/24 25%	2/24 8%	8/24 33%
Arithmetic	19/23 83%	0/23 0%	0/23 0%	1/23 4%	7/23 30%	13/23 57%	16/23 70%	0/23 0%	0/23 0%	1/23 4%	22/23 96%
Denial Service	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%
Front Running	2/7 29%	0/7 0%	0/7 0%	0/7 0%	1/7 14%	0/7 0%	2/7 29%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	30/34 88%	0/34 0%	0/34 0%	2/34 6%	15/34 44%	21/34 62%	28/34 82%	14/34 41%	33/34 97%	30/34 88%	33/34 97%
Time Manipulation	5/7 71%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	1/7 14%	0/7 0%	0/7 0%	2/7 29%	1/7 14%	6/ 7 86%
Unchecked Low Calls	61/75 81%	0/75 0%	0/75 0%	0/75 0%	27/75 36%	0/75 0%	0/75 0%	49/75 65%	48/75 64%	60/75 80%	70/75 93%
Other	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/ 5 0%
Total	117/224 52%	0/224 0%	0/224 0%	5/224 2%	54/224 24%	35/224 16%	46/224 21%	65/224 29%	89/224 40%	94/224 42%	141/224 63%

Table 1: Results with criterion that must indicate the correct vulnerability category and line number in source code

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/24 0%	0/24 0%	0/24 0%	5/24 21%	4/24 17%	0/24 0%	0/24 0%	1/24 4%	6/24 25%	2/24 8%	8/24 33%
Arithmetic	19/23 83%	0/23 0%	0/23 0%	13/23 57%	16/23 70%	13/23 57%	18/23 78%	0/23 0%	0/23 0%	1/23 4%	22/23 96%
Denial Service	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	1/14 7%	1/14 7%
Front Running	2/7 29%	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	2/7 29%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	30/34 88%	19/34 56%	0/34 0%	15/34 44%	25/34 74%	21/34 62%	28/34 82%	14/34 41%	33/34 97%	30/34 88%	33/34 97%
Time Manipulation	7/7 100%	0/7 0%	0/7 0%	4/7 57%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	3/7 43%	2/7 29%	7/ 7 100%
Unchecked Low Calls	62/75 83%	0/75 0%	0/75 0%	9/75 12%	60/75 80%	0/75 0%	0/75 0%	50/75 67%	51/75 68%	61/75 81%	70/75 93%
Other	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/ 5 0%
Total	120/224 54%	19/224 8%	0/224 0%	46/224 21%	107/224 48%	36/224 16%	48/224 21%	67/224 30%	93/224 42%	97/224 43%	143/224 64%
Delta	2%	8%	0%	19%	24%	0%	0%	1%	2%	1%	1%

Table 2: Results with criterion that must indicate the correct vulnerability category but with a line number in source code with an interval of -5 to +5

that the hash terminates in 1,2 or 3 because we only want to analyse about 20% of them for each tool. However, if the false positives that will be manually analysed in each category are below than 10 we increased the percentage by checking if the hash terminates with the next character, which is 4, until we have at least 10 false positives to manually analyse in each category.

We then proceed to update the Table 3 with the new numbers based on our manual verification and the results can be seen in Table 4.

The false positives of Conkas decrease in almost every category, however, the arithmetic category is the most punished. We have less false positives than Mythril and in Unchecked Low Calls, we have the lowest number of false positives. In our analyses we observed that Slither's report has a lot of line numbers, it reports almost every line number in the file. The script used in our analysis check if there is any line number where exists a true positive and the other are not counted to false positives. With this in mind, we can conclude that our tool, in practice, have lower false positives than Slither as well, being Smartcheck the tool with less false positives.

5.3. Performance

The performance of the tools is a point to be in account because if a tool lingers a lot to analyse a contract the user will be unsatisfied. The setup used was a virtual machine running Ubuntu 18.04.5 LTS with 4 cores of the CPU Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz and with 16GB of RAM. In Table 5 is shown, in each line, the average time needed and the total time execution for each tool.

Manticore is by far the most slower tool followed by Maian. Slither has the best performance with just 4 seconds on average. Conkas is behind Oyente but ahead of Mythril. However, Oyente does not have the extra step to elevate bytecode to an IR. Slither and Smartcheck analyse the source code and does not have the step of compiling nor execute each instruction symbolically.

5.4. Tools Combined

In Table 6 are shown the possible combinations of each tool based on the true positive rate of each one, using as a metric the union of the detected vulnerabilities.

With these results, the best choice goes to Conkas with Mythril or Conkas with Slither, with the best percentage of 58% in both choices. In the other hand, Slither in each report is very verbose showing a lot of lines where the vulnerabilities can happen, being almost all of them false positives, so we recommend Conkas and Mythril. Conkas and Smartcheck can also be a good choice, with less 1% but Smartcheck is the tool that shows the lowest number of false positives and is faster than Mythril.

5.5. Analysis Limitations

One limitation of our analysis is that Conkas were not tested with contracts written in Solidity with versions higher or equal to 0.6.0. The SmartBugs dataset used does not provide any contract written in these conditions. We tested only two contracts with reentrancy vulnerability, each of them written in 3 different versions, 0.4.25, 0.5.17 and 0.6.7. Conkas does not throw any error on both contracts in all versions and report the expected vulnerabilities.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	0	11	60	43	0	0	5	34	2	155
Arithmetic	306	0	0	9	203	200	377	0	0	24	1119
Denial Service	0	0	0	0	0	41	83	0	13	29	166
Front Running	24	0	0	0	53	0	60	202	0	0	339
Reentrancy	221	0	0	17	119	30	26	80	48	21	562
Time Manipulation	71	0	0	6	0	6	10	0	2	0	95
Unchecked Low Calls	35	0	0	5	30	0	0	96	45	13	224
Other	0	11	4	47	67	0	0	0	31	17	177
Total	657	11	15	144	515	277	556	383	173	106	2837

Table 3: Number of false positives

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	-	-	-	43	-	-	-	34	2	79
Arithmetic	217	-	-	-	179	-	-	-	0	14	410
Denial Service	0	-	-	-	0	-	-	-	13	29	42
Front Running	13	-	-	-	48	-	-	-	0	0	61
Reentrancy	99	-	-	-	102	-	-	-	44	8	253
Time Manipulation	33	-	-	-	0	-	-	-	2	0	35
Unchecked Low Calls	0	-	-	-	25	-	-	-	45	6	76
Other	0	-	-	-	67	-	-	-	31	17	115
Total	362	-	-	-	464	-	-	-	169	76	1071
Delta	295	-	-	-	51	-	-	-	4	30	1766

Table 4: Number of false positives updated

6. Conclusions

We started describing blockchain, Ethereum, Smart Contracts and Ethereum Virtual Machine. We also describe the vulnerabilities that Conkas detect and study the state of the art. Next, we introduce Conkas, a modular tool of static analysis, easy to add new modules where users can write their own policies and easy to add EVM instructions. We introduce Rattle and made some modifications to adapt to be part of Conkas as an IR. We analyse Conkas, like the performance against other tools using SmartBugs framework. Analyse also the true positive rate between all tools and Conkas had the higher percentage. We show the false positives of each tool and manually verified, for about 20% of the files that have at least one false positive, being Smartcheck the tool that has less false positives followed by Conkas that improved well. We also analyse the combination of tools using as metric the union of vulnerabilities detected by each tool. With Conkas we hope that developers and security engineers can use to automate the analysis of Smart Contracts.

6.1. Future Work

Some suggestion to future work includes improving the Symbolic Execution Engine increasing the level of variable information (e.g. where they were declared). We could also add the possibility to use different strategies in path exploration, improve the arithmetic module, add more modules to detect more vulnerability categories, add newer EVM instructions when they emerge and eliminate the limitations of Rattle.

References

[1] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 513–520. Springer, 05 2018.

[2] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In M. Pinzger, G. Bavota, and A. Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 442–446. IEEE Computer Society, 2017.

[3] Ethereum. Solidity, the contract-oriented programming language. `https://github.com/ethereum/solidity`. (Accessed on 15/09/2020).

[4] Ethereum. White paper · a next-generation smart contract and decentralized application platform. `https://ethereum.org/en/whitepaper/`. (Accessed on 15/09/2020).

[5] S. Falkon. The story of the dao — its history and consequences. `https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee`. (Accessed on 16/09/2020).

[6] J. Feist. Slithir, an intermediate representation of solidity to enable high precision security analysis. `https://github.com/trailofbits/publications/tree/master/presentations/SlithIR,AnIntermediateRepresentationofSoliditytoenableHighPrecisionSecurityAnalysis`, 2019. (Accessed on 17/09/2020).

[7] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

#	Tool	Avg. Execution Time	Total Execution Time
1	Conkas	0:00:32	1:14:37
2	Honeybadger	0:01:12	2:49:03
3	Maian	0:03:47	8:52:25
4	Manticore	0:12:53	1 day, 6:15:28
5	Mythril	0:00:58	2:16:21
6	Osiris	0:00:21	0:50:25
7	Oyente	0:00:05	0:12:35
8	Securify	0:02:06	4:56:13
9	Slither	0:00:04	0:09:56
10	Smartcheck	0:00:15	0:35:23
Total		———	2 days, 4:12:25

Table 5: Execution time of each tool

	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck
Conkas		120/224 54%	120/224 54%	125/224 56%	130/224 58%	120/224 54%	123/224 55%	127/224 57%	131/224 58%	128/224 57%
Honeybadger			19/224 8%	60/224 27%	113/224 50%	41/224 18%	48/224 21%	80/224 36%	93/224 42%	97/224 43%
Maian				46/224 21%	107/224 48%	36/224 16%	48/224 21%	67/224 30%	93/224 42%	97/224 43%
Manticore					114/224 51%	63/224 28%	70/224 31%	91/224 41%	114/224 51%	117/224 52%
Mythril						118/224 53%	117/224 52%	115/224 51%	126/224 56%	123/224 55%
Osiris							53/224 24%	91/224 41%	107/224 48%	111/224 50%
Oyente								99/224 44%	113/224 50%	116/224 52%
Securify									105/224 47%	105/224 47%
Slither										109/224 49%
Smartcheck										

Table 6: Tools' Combination

[8] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. Smartbugs: A framework to analyze solidity smart contracts. *CoRR*, abs/2007.04771, 2020.

[9] N. Group. Decentralized application security project (or dasp) top 10. `https://dasp.co/`. (Accessed on 16/09/2020).

[10] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[11] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD, August 2018. USENIX Association.

[12] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[13] S. Marx. Stop using solidity's transfer() now. `https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/`. (Accessed on 16/09/2020).

[14] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1186–1189. IEEE, 2019.

[15] B. Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.

[16] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. (Accessed on 15/09/2020).

[17] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.

[18] D. Pérez and B. Livshits. Smart contract vulnerabilities: Does anyone care? *CoRR*, abs/1902.06710, 02 2019.

[19] Solidity. Yul. `https://solidity.readthedocs.io/en/latest/yul.html`. (Accessed on 17/09/2020).

[20] R. Stortz. Rattle - an ethereum evm binary analysis framework. In reCON Montreal Conference, `https://www.trailofbits.com/presentations/rattle/`, 2018. (Accessed on 17/09/2020).

[21] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[22] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 664–676. ACM, 12 2018.

[23] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.

[24] G. Wood. Ethereum: A secure decentralised generalised transaction ledger petersburg version 3e2c089 – 2020-09-05. `https://ethereum.github.io/yellowpaper/paper.pdf`. (Accessed on 15/09/2020).