PROJECT ITERATION - 1

CSE 6324

Christoph Csallner

University of Texas at Arlington

Project Title: Conkas 2.0 – Static Analysis Tool

Team - 6

Apoorva Siri Mattewada - 1002026609

Harshith Kannalli Sachidananda Murthy -1001949874

Manisha Sohanlal Jain - 1001869817

Vinayak Neemkar - 1002022438

Vignaan Erram - 1002032121

Abstract

Modern blockchains like Ethereum, enable any user to create a program called smart contract. Ethereum also has a cryptocurrency called Ether. These smart contracts can send and receive ether between each other via transactions. In order for a transaction to occur, a smart contract must be deployed. Once it is deployed, the owner of that contract can no longer change it. This may lead to generating some vulnerabilities. These vulnerabilities make the smart contract prone to attacks, that lead to financial losses. In order to overcome this challenge, and reduce the probability of new future attacks happening, several smart contract analysis tools were created. These tools help detect and solve vulnerabilities before deployment, reducing the risk of attacks. Conkas was one such tool. It is a modular static analysis tool that uses symbolic execution to find traces that lead to vulnerabilities and uses an intermediate representation. 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 versions of Solidity, but the analysis is done at the bytecode level. [1]

Conkas supports 5 modules that detect vulnerabilities: Arithmetic, Front-Running, Reentrancy, Time Manipulation and Unchecked Low-Level Calls. Conkas is easy to extend, meaning that you can add your custom modules to detect other types of vulnerabilities. We plan to improve Conkas by adding module which can detect bad randomness and short address attack vulnerability.

GitHub Repository

- Link: https://github.com/apoorvamattewada/Conkas-2.0

Architecture

Conkas is a modular static analysis tool that uses the symbolic execution model. It utilizes a Command-Line Interface (CLI), which aids in interaction. The users deploy the byte code associated with contract or solidity source code (.sol files). The architecture of the Conkas tool is explained in the figure 1.

The users deploy the bytecode which is passed to the rattle module. This module will elevate the bytecode to an IR and translate the instruction's SSA form and stack form to register form. The Control Flow Graph will also be constructed. This will be passed to the next module which is Symbolic Execution Engine. The Symbolic Execution Engine is responsible to iterate over control flow graphs and generate traces. These are provided to the Detectors module, which contains sub modules, each responsible to detect a category of vulnerabilities. The Symbolic execution engine and the Detectors 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 its possibility and to determine whether a vulnerability exists or not.

Below figure 1. is a transition diagram that explains the actual architecture of the Conkas tool.

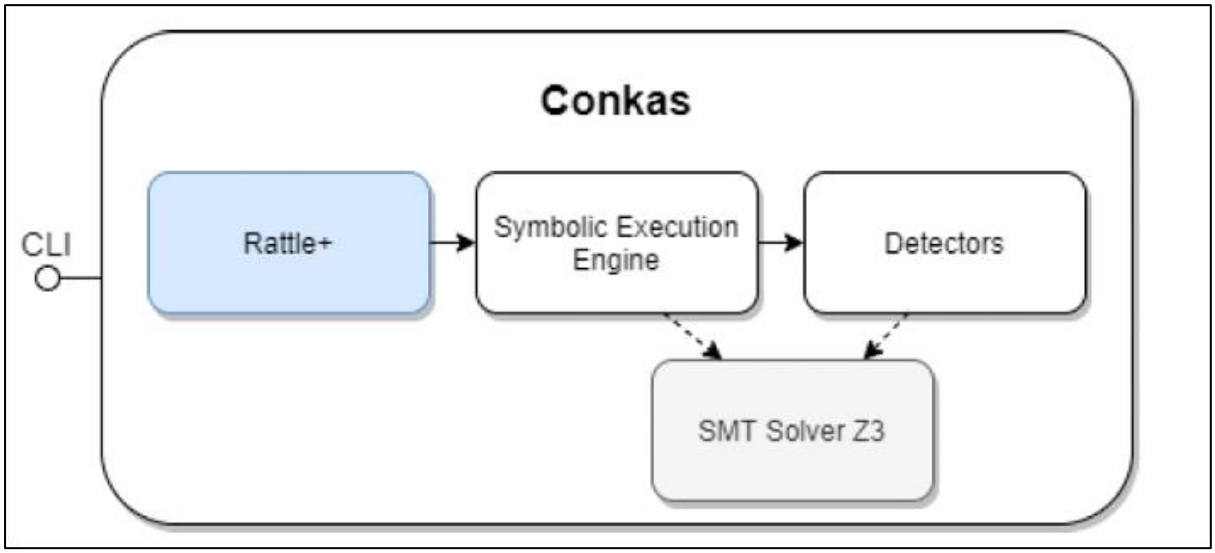

Figure 1: Architecture of Conkas tool [1]

The upgraded version is our tool, Conkas 2.0, as we are planning to add two modules to this smart contract tool. This is represented in figure 2, as mentioned below.

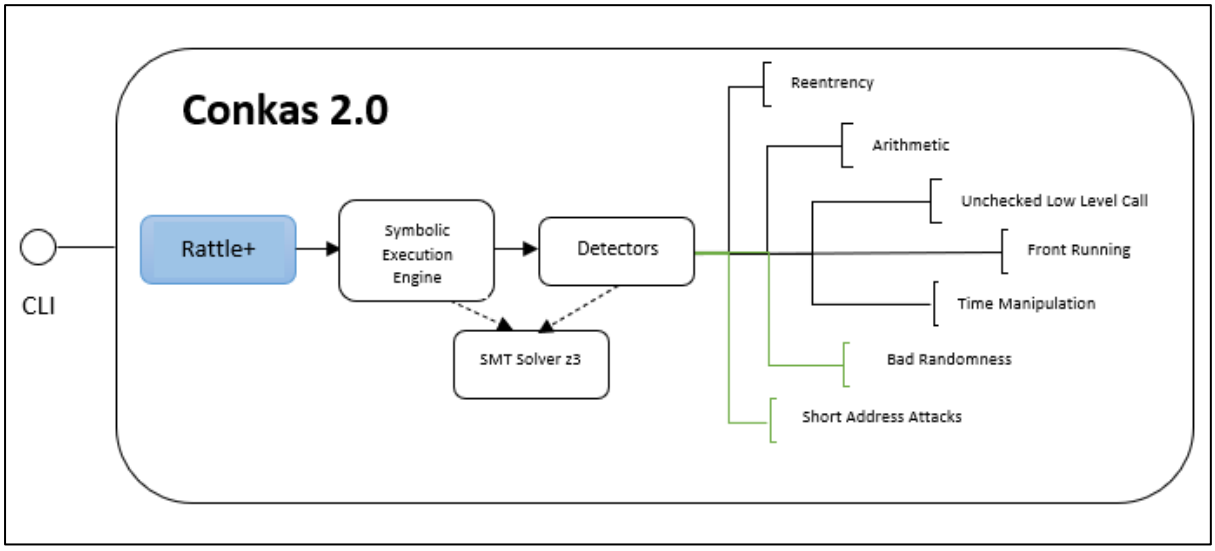

Figure 2: Architecture of Conkas 2.0 tool

Project Plan

Iteration	Goals	Achieved Goals
1	To run the Conkas tool successfully and test it with some sol files. And understand what the tool is doing. And fix any dependency issues if there are any.	Successfully installed the tool and fixed the versions of the libraries, solved some compatibility and dependency issues; run the sample .sol files successfully. Working on bad randomness vulnerability.
2	Come up with suitable code for the "bad randomness" vulnerability and integrate it in our tool.	
3	Work on the "short address attack" vulnerability. Finally, test all the modules and run it on the tool successfully.	

Comparison with Competitors:

The following table explains the comparison of Conkas with its top two competitors, Slither and Mythril. [6] These competitors, among the many ones in market are selected as they are both static analysis and have integrated bad randomness and short address attack vulnerabilities.

We see that Slither has good execution time, but only accepting only .sol files. This does not allow its users to have the freedom of sharing their bytecode files and hence compromise in security. This is a drawback despite having good execution time when compared to Conkas.

Mythril on the other hand has the ability to share bytecode and .sol files. However, when compared to Conkas, it states to have bad execution time. Having a longer execution time is a drawback, which is why we have chosen to work with Conkas.

Tool Name	Execution Time	Accepted Input
Slither	Good Execution Time	Only .sol file
Conkas	Good Execution Time	bytecode/.sol file
Mythril	Bad Execution time	bytecode/.sol file

Table: Comparison with Competitors

Implementing the new vulnerability - Bad Randomness:

What is Bad Randomness?

Randomness is hard to get in Ethereum. This means, it is hard to achieve hard-to-predict values, that are generally either more public than they seem or subject to miner's influence.

As the source of randomness are predictable to an extent, malicious users can replicate it and attack a function that uses this unpredictability. [4]

Loss: more than 400 ETH [4]

Real World Impact: [4]

- SmartBillions Lottery
- TheRun

Example:

1. A smart contract uses the block number as a source of randomness for a game. [4]
2. An attacker creates a malicious contract that checks if the current block number is a winner. If so, it calls the first smart contract to win; since the call will be part of the same transaction, the block number will remain the same on both contracts. [4]
3. The attacker only must call her malicious contract until it wins. [4]

The figure below explains our attempt to implement bad randomness vulnerability detection to our tool. This is formulated as a general idea, as we plan on improving the current algorithm to achieve optimization.

Fig: Current test for bad randomness (yet to be implemented)

Risks and Plan to mitigate risks:

Risks	Explanation	Resolution	Severity
Process Knowledge Risk	Difficulty in understanding the tool and working around its complexities.	All team members are trying their best to learn about the tool and share their knowledge in the meetings.	High
Integration Risk	Encountering issues while integrating bad	Generating a proper understanding of code	Medium

	randomness vulnerability in the current version of code.	and figuring out where the bad randomness module needs to be placed, via custom modules. Parallel check on dependencies is done.	
Missing Risk	Conkas project does not seem very active (few commits/issues to git), may be hard to reach the community.	We are trying to find more information about the tool by contacting people how worked on smart contracts, GitHub stargazers, etc.	Low
Logical Risk	Instances where code for bad randomness cannot detect said vulnerabilities. It may generate false positives and false negatives.	This risk can be handled by testing and debugging code for issues.	High
Increased Complexity	Adding a new vulnerability detection rule can increase complexity of the analysis tool, potentially making it difficult to maintain and update overtime.	Creating a separate file/module for bad randomness vulnerability to allow easier management and create efficient and readable code.	Medium
Reduced Performance	Adding new rules to the smart contracts can increase the performance time.	Using algorithms and data structures that are specifically designed for detecting bad randomness in smart contracts, via., efficient time and space complexity.	Medium

Table: Risks and plans to mitigate risks

Other generic risks for bad randomness include (some of them may pertain to the current vulnerability being worked on – bad randomness):

- Financial Risk
  If a smart contract uses bad randomness, the outcomes may not be fair or unpredictable. This can lead to financial losses for users who participate in games, lotteries, or other contracts that rely on randomization.
- Unfair Competition
  If a smart contract uses bad randomness, it may give some users an unfair advantage over others, leading to an unlevel playing field. This can be particularly harmful in contracts related to online gaming, where fairness is critical to maintain the integrity of the game.
- Mistrust

If a smart contract uses bad randomness, it can erode the trust of users and customers. People may be less likely to use the contract or the platform that hosts it if they perceive it as unreliable or prone to manipulation.

- Security Risk
  Bad randomness can also increase security risks for users. For example, if a smart contract uses predictable randomness, attackers can potentially exploit this vulnerability to cheat or manipulate the contract.

Specifications and Design:

The specification and design goals for our tool are:

- Conkas should be able to analyze Solidity code, which is the programming language used for Ethereum smart contracts.
- Conkas should be able to identify potential security vulnerabilities in the code, such as re-entrancy attacks, integer overflows, and other common issues.
- Conkas should be able to analyze the control flow of the contract to identify potential issues with the logic or execution of the code.
- Conkas should be able to identify potential issues with data storage and management in the contract.
- Conkas should be able to generate reports or other outputs that highlight the issues identified in the code.
- Conkas may use a combination of parsing, control flow analysis, and other techniques to analyze the code.

- The tool may be designed to work as a standalone command-line tool or as a plugin for an integrated development environment (IDE) or other development tool.

- Conkas may use a rule-based system to identify potential vulnerabilities or issues in the code, or it may use machine learning or other advanced techniques to improve its analysis.

List of Features:

Here is a list of features that Conkas offers:

- Solidity Code and Bytecode input support.
- Command line interface
- Support for custom vulnerability scanning (specify the type of vulnerability to search).
- Support for custom modules.
- Support for EVM instructions.
- Unit test support.

The system requirements to run the tool are:

- Python3 (or the latest version).

- Modules to install:
    - cobr2
    - py-solc-x
    - pycryptodome
    - pyevmasm
    - solidity_parser
    - z3-solver
    - solc-select

Key Data Structures:

- Regular Expressions
- List

Code and Test:

User will need the “requirements.txt” file from the “GITHUB LINK” to install all the dependencies required or by setting up a virtual environment with the following steps:

1. python3 -m venv venv

2. source venv/bin/activate

3. pip install -r requirements.txt

Expected Input for Application:

If users wish to look for a certain kind of vulnerability in a smart contract written in Solidity, they can enter the following commands:

$ python conkas.py -vt reentrancy -s some_file.sol

If the user has the built runtime bytecode that Conkas provided, they can type the following to search for all of Conkas’s vulnerabilities:

$ python3 conkas.py some_file.bin

```
(venv) PS C:\Users\vignan\Desktop\ase project\conkas> python conkas.py --help
usage: conkas.py [-h] [--solidity-file] [--verbosity VERBOSITY]
                 [--vuln-type VULN_TYPE] [--max-depth MAX_DEPTH]
                 [--find-all-vulnerabilities] [--timeout TIMEOUT]
                 file

Symbolic execution tool for EVM

positional arguments:
  file                  File with EVM bytecode hex string to analyse

optional arguments:
  -h, --help            show this help message and exit
  --solidity-file, -s   Use this option when file is a solidity file instead
                        of EVM bytecode hex string. By default it is unset
  --verbosity VERBOSITY, -v VERBOSITY
                        Log output verbosity (NotSet, Debug, Info, Warning,
                        Error, Critical). Default = Error
  --vuln-type VULN_TYPE, -vt VULN_TYPE
                        VULN_TYPE can be ['arithmetic', 'reentrancy',
                        'time_manipulation',
                        'transaction_ordering_dependence',
                        'unchecked_ll_calls']. Default = ['arithmetic',
                        'reentrancy', 'time_manipulation',
                        'transaction_ordering_dependence',
                        'unchecked_ll_calls']
  --max-depth MAX_DEPTH, -md MAX_DEPTH
                        Max recursion depth. The counting is how many basic
                        blocks should be analysed. Default = 25
  --find-all-vulnerabilities, -fav
                        When set it will try to find all possible
                        vulnerabilities. It will take some time. By default it
                        is unset
  --timeout TIMEOUT, -t TIMEOUT
                        Timeout to Z3 Solver. Default = 100
```

Fig: Input code

Here are the arguments we can use:

1. [--solidity-file]: This is used when the file is solidity file instead of the bytecode.
2. [--verbosity]: With this argument we can know the log output such as NotSet, Debug, Info, Warning, Error and Critical. Default: Error
3. [--vuln-type]: We can find the specific type of vulnerability which can arithmetic, reentrancy, time- manipulation, transaction ordering dependence, unchecked low calls.
4. [--max-depth]: The file is analysed up to a maximum depth. Default is 25.
5. [--find-all-vulnerabilities]: When we set this it will find all the vulnerabilities in the sol file by default it is unset.
6. [--timeout]: The execution stops when we give this argument.

Output (Code):

The output will be shown in accordance to the information the user enters for the command-line argument's vulnerability name along with the line number of the solidity code where this vulnerability is. An example is shown below.

Here we are testing all the vulnerabilities which Conkas can detect in kotete.sol (refer the GitHub link for the sol file) file by using the following command: **$ python conkas.py --find-all-vulnerabilities -s ..\kotete.sol.**

These are all the vulnerabilities found:

1. Reetrancy: Here reentrancy as given in the picture is found in functions such as sweepCommission(uint256), _fallthrough at different lines as shown in the picture.
2. Integer Overflow/Underflow: These are the arithmetic vulnerabilities which are found in the function '_fallthrough' and 'claimthrone(string)'.

3. Time manipulation: Time manipulation is found in the function '_fallthrough' as shown in the below picture.
4. Transaction ordering dependency: This is also found in function '_fallthrough'.
5. Unchecked low level calls: This vulnerability is found in sweepCommission(uint256) and _fallthrough.

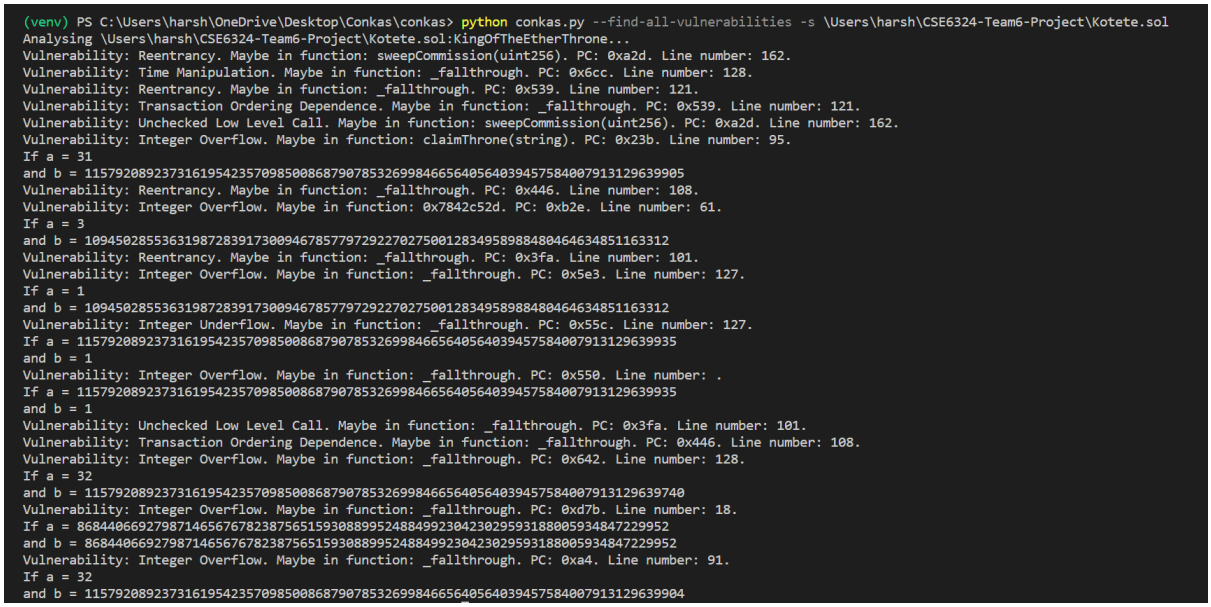

Fig: Output generated

It is to note that the above output shows that Conkas is unable to detect Bad randomness vulnerabilities and report them. It detects the 5 vulnerabilities (Reentrency, Arithmetic, Unchecked Low-Level Calls, Front-Running and Time Manipulation [1]). This output has been generated by deploying a sample solidity smart contract [5].

Target Users/Customers of the Tool:

- All smart contract developers who wish to work on lottery and raffle smart contracts for gambling.
- People who use payment processing smart contracts want to check for short address vulnerability.
- All professors and students who desire to do study on this type of tool are welcome to utilize it.
- People that want to modify it and add new features can adopt it.

Feedback:

- Alexander Norta & Toomas Lepikult - "Conkas's primary known limitation is its inability to detect smart contract dependency files". [2]
- Shovon Pereira - "Good tool to work on. If I were to share my bytecode files instead of .sol files for security reasons, this tool would do the job for me".

References:

1. https://www.semanticscholar.org/paper/Conkas%3A-A-Modular-and-Static-Analysis-Tool-for-Veloso/425e474177885f9ac9e57d44e8e2386d13f9c87d
2. https://digikogu.taltech.ee/et/Download/1c564244-622c-4b84-9af1-743b67f5400e/Nutilepingutehaavatavustetuvastamisetriistade.pdf
3. https://github.com/apoorvamattewada/Conkas-2.0
4. DASP - TOP 10
5. https://github.com/crytic/not-so-smart-contracts/blob/master/unchecked_external_call/KotET_source_code/KingOfTheEtherThrone.sol
6. (PDF) Ethereum Smart Contract Analysis Tools: A Systematic Review (researchgate.net)