Analysis and Verification Tools for Solidity

Dr. Raju Halder,
Dept. of Comp. Sc. & Engg., IIT Patna
halder@iitp.ac.in

Oyente
https://oyente.tech/

Oyente was created by Loi Luu and his team from National University of Singapore in 2016. Oyente is currently maintained by the public community like any open source project.

Smart contracts are fundamentally full-fledged programs that run on blockchains. These "contracts" are automatically executed when specified requirements are met. With the steady rise in the adoption of smart contracts - Ethereum's system supports tens of thousands of contracts-, these contracts hold millions of dollars (ever increasing) worth of virtual coins.

The projected rise in the adoption of smart contracts means that the value being processed by these contracts will also increase. As with anything of value, the relationship between risk and reward is directly proportional. An increase in reward (access to value) results in greater risks (whether inherent or external). We have

Making Smart Contracts Smarter

Loi Luu
National University of Singapore
loiluu@comp.nus.edu.sg

Duc-Hiep Chu
National University of Singapore
hiepcd@comp.nus.edu.sg

Hrishi Olickel
Yale-NUS College
hrishi.olickel@yale-nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

Aquinas Hobor
Yale-NUS College&
National University of Singapore
hobor@comp.nus.edu.sg

ABSTRACT

Cryptocurrencies record transactions in a decentralized data structure called a blockchain. Two of the most popular cryptocurrencies, Bitcoin and Ethereum, support the feature to encode rules or scripts for processing transactions. This feature has evolved to give practical shape to the ideas of smart contracts, or full-fledged programs that are run on blockchains. Recently, Ethereum's smart contract system has seen steady adoption, supporting tens of thousands of contracts, holding millions dollars worth of virtual coins.

In this paper, we investigate the security of running smart contracts based on Ethereum in an open distributed network like those of cryptocurrencies. We introduce several new security problems in which an adversary can manipulate smart contract execution to gain profit. These bugs suggest subtle gaps in the understanding of the distributed semantics of the

been used more broadly [2–4]. One prominent new use for blockchains is to enable *smart contracts*.

A smart contract is a program that runs on the blockchain and has its correct execution enforced by the consensus protocol [5]. A contract can encode any set of rules represented in its programming language—for instance, a contract can execute transfers when certain events happen (e.g. payment of security deposits in an escrow system). Accordingly, smart contracts can implement a wide range of applications, including financial instruments (*e.g.*, sub-currencies, financial derivatives, savings wallets, wills) and self-enforcing or autonomous governance applications (*e.g.*, outsourced computation [6], decentralized gambling [7]).

A smart contract is identified by an address (a 160-bit identifier) and its code resides on the blockchain. Users invoke a smart contract in present cryptocurrencies by sending

Proposing/Accepting Blocks Semantics

- A blockchain state σ is a mapping from addresses to accounts.
- $\sigma \xrightarrow{T} \sigma'$: Valid transition via transaction T.

$$\textsc{Propose}\ \frac{\begin{array}{c}\texttt{TXs} \leftarrow \text{ Some transaction sequence } (T_1 \ldots T_n) \text{ from } \Gamma \\ B \leftarrow \langle \texttt{blockid} \,;\, \texttt{timestamp} \,;\, \texttt{TXs} \,;\, \ldots \rangle \\ \texttt{proof-of-work}(B, BC) \\ \forall\, i, 1 \leq i \leq n : \sigma_{i-1} \xrightarrow{T_i} \sigma_i \end{array}}{\begin{array}{c}\langle BC, \sigma_0 \rangle \Downarrow \langle B \cdot BC, \sigma_n \rangle \\ \text{Remove } T_1 \ldots T_n \text{ from } \Gamma \text{ and broadcast } B\end{array}}$$

$$\textsc{Accept}\ \frac{\begin{array}{c}\text{Receive } B \equiv \langle \texttt{blockid} \,;\, \texttt{timestamp} \,;\, \texttt{TXs} \,;\, \ldots \rangle \\ \texttt{TXs} \equiv (T_1 \ldots T_n) \\ \forall\, i, 1 \leq i \leq n : \sigma_{i-1} \xrightarrow{T_i} \sigma_i \end{array}}{\begin{array}{c}\langle BC, \sigma_0 \rangle \Downarrow \langle B \cdot BC, \sigma_n \rangle \\ \text{Remove } T_1 \ldots T_n \text{ from } \Gamma \text{ and broadcast } B\end{array}}$$

Transaction Execution Semantics

VM's execution state: $\mu = \langle A, \sigma \rangle$

- σ: mapping between addresses and account states.
- A: Call stack of Actiation Records

$$
\begin{aligned}
A &\triangleq A_{normal} \mid \langle e \rangle_{exc} \cdot A_{normal} \\
A_{normal} &\triangleq \langle M, pc, l, s \rangle \cdot A_{normal} \mid \epsilon
\end{aligned}
$$

ϵ: empty call stack

$\langle e \rangle_{exc}$: exception is thrown

M : the contract code array

pc : the address of the next instruction to be executed

l : an auxiliary memory (e.g. for inputs , outputs)

s : an operand stack.

Transaction Execution Semantics

Transaction $T = \langle id, v, l \rangle$

- id is the identier of the to-be-invoked contract,
- v is the value to be deposited to the contract, and
- l is an data array capturing the values of input parameters.

$$\text{TX-SUCCESS}\ \frac{\begin{array}{c} T \equiv \langle id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \epsilon, \sigma'' \rangle \end{array}}{\sigma \xrightarrow{T} \sigma''}$$

$$\text{TX-EXCEPTION}\ \frac{\begin{array}{c} T \equiv \langle id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \langle e \rangle_{exc} \cdot \epsilon, \bullet \rangle \end{array}}{\sigma \xrightarrow{T} \sigma}$$

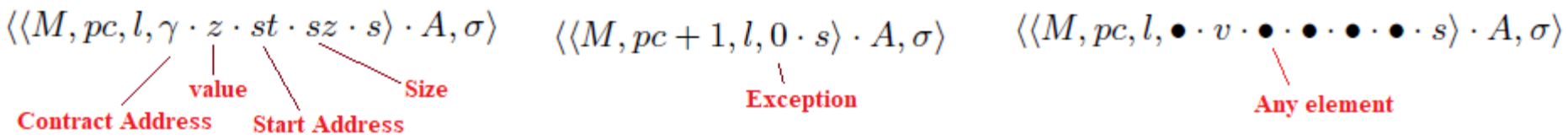
Arbitrary Element

EVM Instructions & Semantics

- ETHERLITE Language:

$$ins \triangleq \texttt{push}\ v \mid \texttt{pop} \mid \texttt{op} \mid \texttt{bne} \mid \texttt{mload} \mid \texttt{mstore} \mid \texttt{sload} \mid \texttt{sstore} \mid \texttt{call} \mid \texttt{return} \mid \texttt{suicide} \mid \texttt{create} \mid \texttt{getstate}$$

$M[pc]$	Conditions	μ	μ'
`call`	$id \leftarrow$ address of the executing contract $a' \leftarrow \langle M, pc, l, s\rangle$ $M' \leftarrow \texttt{Lookup}(\sigma, \gamma)$ $\sigma' \leftarrow \sigma[id][bal \mapsto \sigma[id][bal] - z]$ $\sigma'' \leftarrow \sigma'[\gamma][bal \mapsto \sigma[id][bal] + z]$	$\langle\langle M, pc, l, \gamma \cdot z \cdot st \cdot sz \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M', 0, l', \epsilon\rangle \cdot a' \cdot A, \sigma''\rangle$
`call`	$id \leftarrow$ address of the executing contract $\sigma[id][bal] < v$ or $\lvert A\rvert = 1023$	$\langle\langle M, pc, l, \bullet \cdot v \cdot \bullet \cdot \bullet \cdot \bullet \cdot \bullet \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M, pc+1, l, 0 \cdot s\rangle \cdot A, \sigma\rangle$
`return`		$\langle\langle M, pc, \bullet, \bullet\rangle \cdot \epsilon, \sigma\rangle$	$\langle \epsilon, \sigma\rangle$
`return`	$a' \equiv \langle M', pc', l'_0, st' \cdot sz' \cdot s'\rangle$ $n \leftarrow min(sz', sz)$ $0 \le i < n : l'_{i+1} \leftarrow l'_i[st' + i \mapsto l[st + i]]$	$\langle\langle M, pc, l, st \cdot sz \cdot s\rangle \cdot a' \cdot A, \sigma\rangle$	$\langle\langle M', pc'+1, l'_n, 1 \cdot s'\rangle \cdot A, \sigma\rangle$
EXC	exceptional halting of callee	$\langle\langle e\rangle_{exc} \cdot \langle M, pc, l, st \cdot sz \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M, pc+1, l, 0 \cdot s\rangle \cdot A, \sigma\rangle$



Semantics Improvements for Transaction-Ordering Security

- Introducing Guards in the transaction:

$$\text{TX-STALE}\ \frac{T \equiv \langle g, \bullet, \bullet, \bullet \rangle \qquad \sigma \not\vdash g}{\sigma \xrightarrow{T} \sigma}$$

$$\text{TX-SUCCESS}\ \frac{\begin{array}{c} T \equiv \langle g, id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma \vdash g \qquad \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \epsilon, \sigma'' \rangle \end{array}}{\sigma \xrightarrow{T} \sigma''}$$

$$\text{TX-EXCEPTION}\ \frac{\begin{array}{c} T \equiv \langle g, id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma \vdash g \qquad \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \langle e \rangle_{exc} \cdot \epsilon, \bullet \rangle \end{array}}{\sigma \xrightarrow{T} \sigma}$$

Transaction Reordering Issue

- T_o (transction by owner changing reward value) vs. T_u (transaction by other to submit valid puzzle solution)
- Guard g: (**reward==R)**

  where R is currently observed reward value

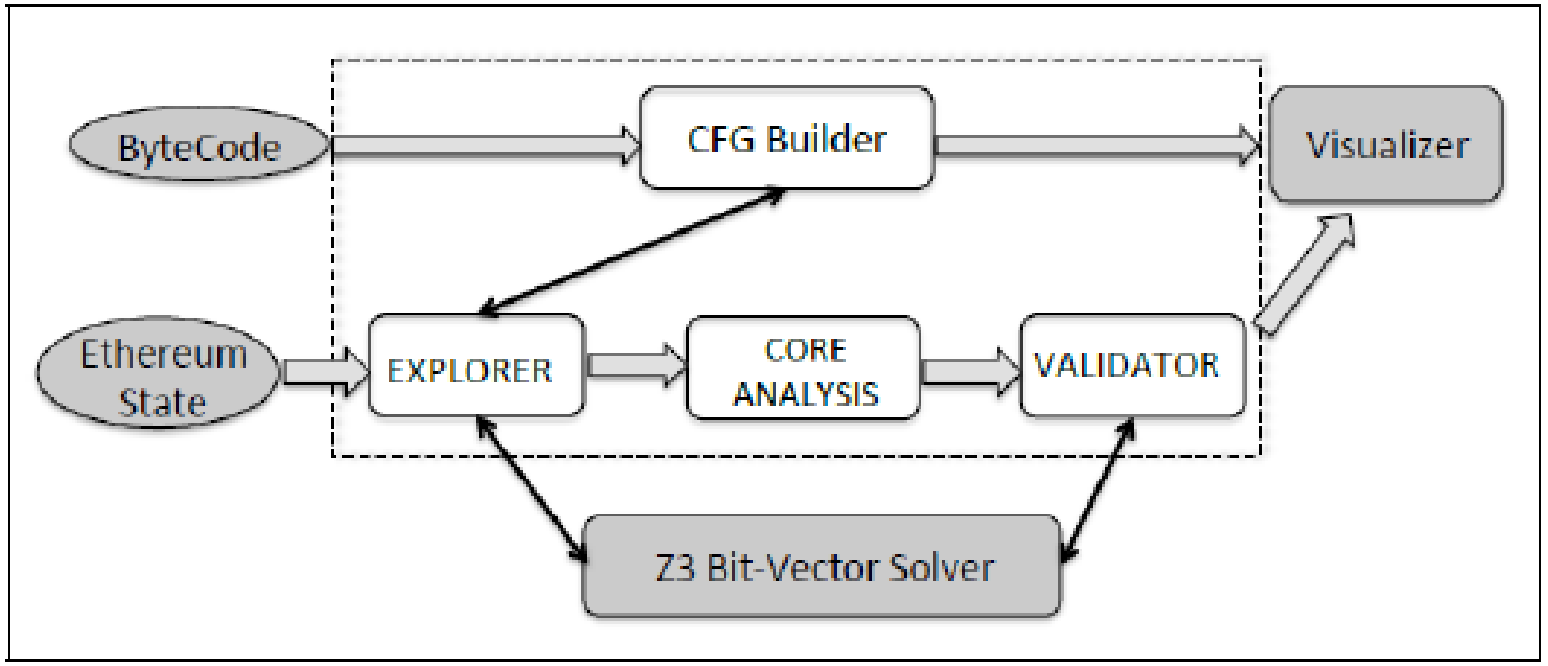
```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

A contract that rewards users who solve a computational puzzle.

Improvement Suggestions

- Deterministic Timestamp
  - Change “timestamp” by “blockNumber”
  - Example, “timestamp – lastTime > 24 hours” can be rewritten as “blockNumber - lastBlock > 7,200”
- Better exception handling
  - Use of throw-catch

Oyente Architecure



Oyente Components

- **CFG Builder:** Builds a skeletal control flow graph containing basic blocks as nodes, and jumps as edges.
- **Explorer:**
  - Performs Symbolic Execution.
  - Produces a set of symbolic traces.
  - Eliminate provably infeasible traces according to constraint solver Z3.
- **Core Analysis:** Identifies the presence of different security vulnerabilities:

Core Analysis

- **TOD detection.**
  - Explorer returns a set of traces and the corresponding Ether flow for each trace.
  - The analysis checks if two different traces have different Ether flows.
  - If a contract has such pairs of traces, Oyente reports it as a TOD contract.
- **Timestamp dependence detection.**
  - Use a special symbolic variable to represent the block timestamp.
  - Given a path condition of a trace, it checks if this symbolic variable is included.
  - A contract is tagged as timestamp-dependent if any of its traces depends on this symbolic variable.
- **Mishandled exceptions.**
  - Detecting a mishandled exception is straightforward.
  - Recall that if a callee yields an exception, it pushes 0 to the caller's operand stack. Thus it needs to check if the contract executes the ISZERO instruction (which checks if the top value of the stack is 0) after every call.
  - If it does not, any exception occurred in the callee is ignored. Thus, it flags such contract as a contract that mishandles exceptions.
- **Reentrancy Detection.**
  - At each CALL that is encountered, it obtains the path condition for the execution before the CALL is executed.
  - It then checks if such condition with updated variables (e.g., storage values) still holds (i.e., if the call can be executed again).
  - If so, we consider this a vulnerability, since it is possible for the callee to re-execute the call before finishing it.

Oyente Components

- Attempts to remove false positives.
- For instance, given a contract tagged as TOD by Core Analysis and its two traces t1 and t2 exhibiting different Ether flows.
- Validator queries Z3 to check if both ordering (t1; t2) and (t2; t1) are feasible.
- If no such t1 and t2 exist, the case is considered as a false positive.

Experimental Setup

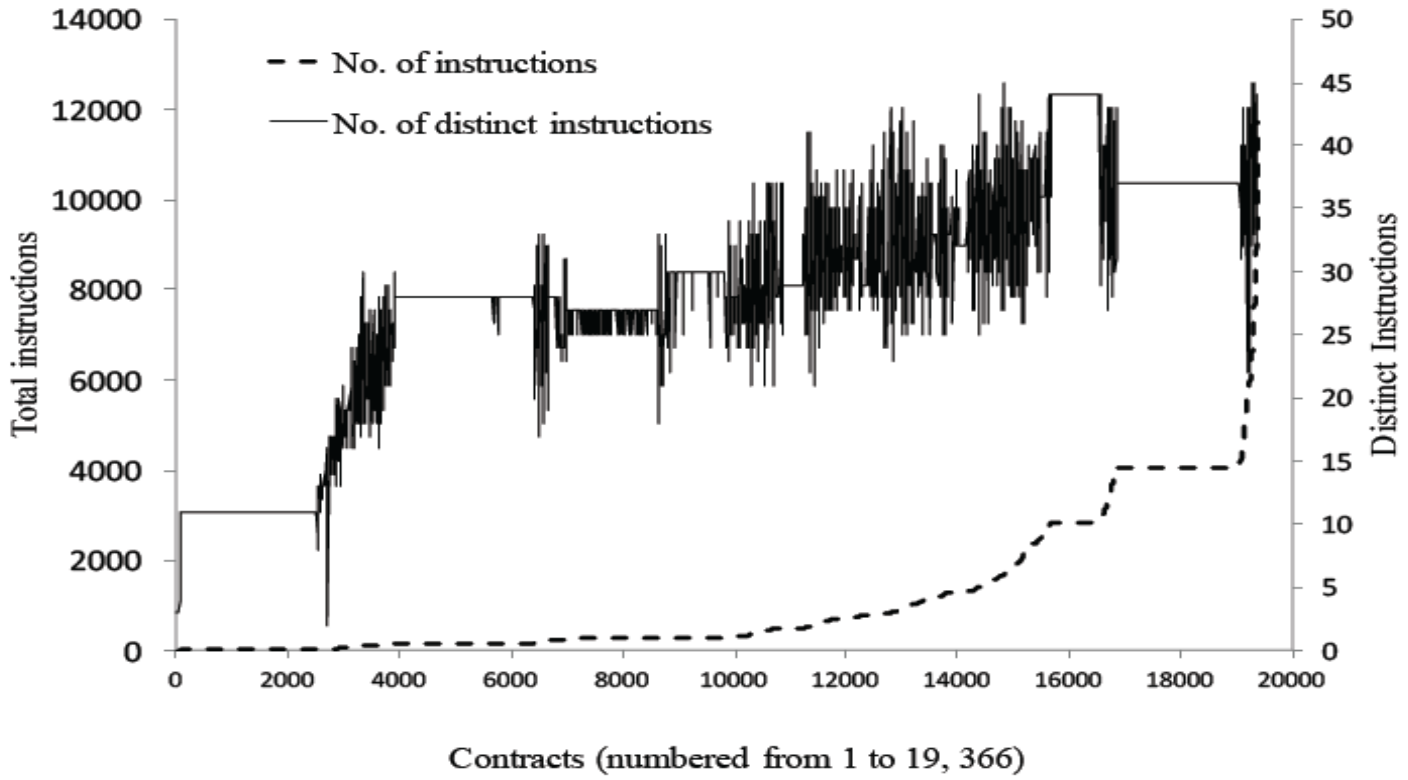
- Oyente in Python with roughly 4000 LoC.
- Oyente faithfully simulates Ethereum Virtual Machine (EVM) code which has 64 distinct instructions in its language.
- Experiment is conducted on 19,366 smart contracts from the blockchain as of May 5, 2016.
- These contracts are found in the first 1,459,999 blocks of Ethereum.
- Run on Amazon EC2 (each has 40 Amazon vCPU with 160 GB of memory and runs 64-bit Ubuntu 14.04).

Financially Motivated Attackers

- These contracts currently hold a total balance of 3,068,654 Ether, or 30 Million US dollars at the time of writing.
  - most of contracts do not hold any Ether (e.g., balance is zero)
  - 10% of them have at least 1 Ether
  - On an average, a contract has 318.5 Ether, or equivalently 4523 US dollars.

Number of SC instructions

- Ethereum contracts vary from being simple to fairly complex.
- The number of instructions in a contract ranges from 18 to 23,609, with an average of 2,505 and a median of 838.

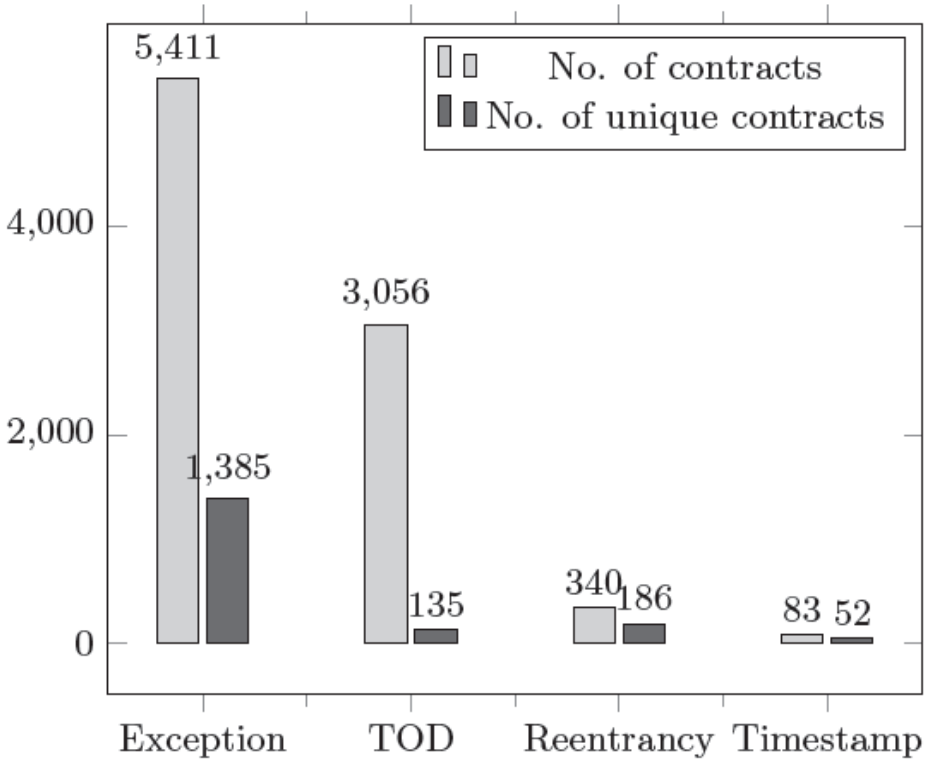


Experimental Evaluation

- Timeout for symbolic execution: 30 min
- Timeout for Z3: 1 sec
- Oyente finds a total number of 366,213 feasible execution paths which took a total analysis time of roughly 3,000 hours
- The number of paths explored by Oyente ranges from 1 to 4613 with an average of 19 per contract and a median of 6.
- On average, Oyente takes 350 seconds to analyze a contract.
- 267 contracts require more than 30 minutes to analyze.

Experimental Evaluation

- We observe that the running time depends near linearly on the number of explored paths, i.e., the complexity of contracts.



- flags 8,833 contracts which have at least one security issue.
- Out of these, 1,682 are distinct.
- Manual check: Oyente has a low false positive rate of 6.4%, i.e., only 10 cases out of 175.

Search projects

Help Sponsors Log in Register

slither-analyzer 0.8.1

```
pip install slither-analyzer
```

Latest version

Released: Aug 16, 2021

Slither is a Solidity static analysis framework written in Python 3.

Navigation

- Project description
- Release history
- Download files

Project description

Slither, the Solidity source analyzer

[![Build Status](https://img.shields.io/github/workflow/status/crytic/slither/CI/master)](https://github.com/crytic/slither/actions?query=workflow%3ACI) [![Slack Status](https://empireslacking.herokuapp.com/badge.svg)](https://empireslacking.herokuapp.com) [![PyPI version](https://badge.fury.io/py/slither-analyzer.svg)](https://badge.fury.io/py/slither-analyzer)

Slither is a Solidity static analysis framework written in Python 3. It runs a suite of vulnerability detectors, prints visual

Slither: A Static Analysis Framework For Smart Contracts

Josselin Feist
Trail of Bits
New York, New York
josselin@trailofbits.com

Gustavo Greico
Trail of Bits
New York, New York
gustavo.grieco@trailofbits.com

Alex Groce
Northern Arizona University
Flagstaff, Arizona
agroce@gmail.com

***Abstract*—This paper describes Slither, a static analysis framework designed to provide rich information about Ethereum smart contracts. It works by converting Solidity smart contracts into an intermediate representation called SlithIR. SlithIR uses Static Single Assignment (SSA) form and a reduced instruction set to ease implementation of analyses while preserving semantic information that would be lost in transforming Solidity to bytecode. Slither allows for the application of commonly used program analysis techniques like dataflow and taint tracking. Our framework has four main use cases: (1) automated detection of vulnerabilities, (2) automated detection of code optimization opportunities, (3) improvement of the user's understanding of the contracts, and (4) assistance with code review.**

In this paper, we present an overview of Slither, detail the design of its intermediate representation, and evaluate its capabilities on real-world contracts. We show that Slither's bug detection is fast, accurate, and outperforms other static analysis tools at finding issues in Ethereum smart contracts in terms of speed, robustness, and balance of detection and false positives. We compared tools using a large dataset of smart contracts and

contract code into an internal representation, more suitable for the analysis and detection of common security issues.

While modern compilers, such as clang, offer various APIs on top of which third-party analyzers can be built, the Solidity compiler fails to offer the same features. Ideally, a static analysis framework for Ethereum smart contracts should have the following properties:

1) **Correct level of abstraction**: if the framework is too abstract, it can be hard to introduce accurate semantics that capture common usage patterns. Conversely, if the framework is too narrowly focused on the detection of certain issues only, it can be difficult to add new detectors or analyses.
2) **Robustness**: it should parse and analyze real-world code without crashing.
3) **Performance**: analysis should be fast, even for large

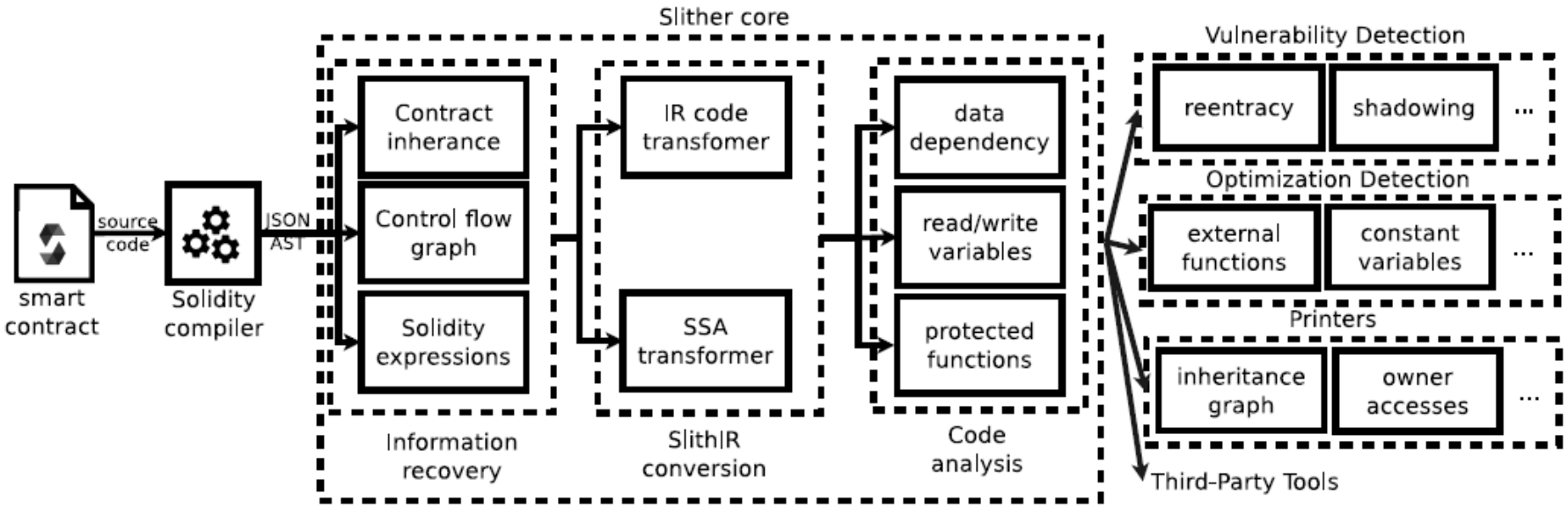
Properties of Good Static Analyzer

- **Correct level of abstraction:**
  - **I**f the framework is too abstract, it can be hard to introduce accurate semantics that capture common usage patterns.
  - Conversely, if the framework is too narrowly focused on the detection of certain issues only, it can be difficult to add new detectors or analyses.
- **Robustness:** It should parse and analyze real-world code without crashing.
- **Performance:** Analysis should be fast, even for large contracts.
- **Accuracy:** It should find most potential issues while maintaining a low false positive rate.
- **Batteries included:**
  - It should include a set of common analyses and issue detectors that are useful for most contracts.
  - This will appeal both to security engineers looking to extend the framework and to code auditors looking for issues to report.

Slither Architecure

The framework is currently used for the following:

- **Automated vulnerability detection**
- **Automated optimization detection**
- **Code understanding**
- **Assisted code review**



Slither Tasks

- Slither takes as initial input the Solidity Abstract Syntax Tree (AST) generated by the Solidity compiler.
- In the first stage, Slither recovers important information such as the contract's inheritance graph, the control flow graph (CFG), and the list of expressions.
- Next, Slither transforms the entire code of the contract to SlithIR, its internal representation language.
- SlithIR uses static single assessment (SSA) to facilitate the computation of a variety of code analyses.

Built-in Code Analyses

- **Read/Write:** Slither identifies the reads and the writes of variables and their types (local or state).
- **Protected functions:** Modeling the protection of functions (use of ownership) lowers the number of false positives. Slither checks for following cases:
  - the function is not the constructor
  - the address of the caller, msg.sender, is not directly used in a comparison.
  - This heuristic can give false positives and false negatives, but our experience shows that it is effective in practice.
- **Data dependency analysis:** Slither computes the data dependencies of all variables as a pre-stage using the SSA representation.
  - Taint Analysis.

Automated Vulnerability Detection

The open source version of Slither contains more than twenty bug detectors, including:

- **Shadowing:** Solidity enables the shadowing of most of the contract's elements, such as local and state variables, functions, or events. Shadowed elements may not refer to the objects expected by the contract author.
- **Uninitialized variables:** Uninitialized state or local variables are a common source of errors in programming languages.
- **Reentrancy:**
- A variety of other known security issues, such as suicidal contracts, locked ether, or arbitrary sending of ether.

Automated Optimization Detection

- Slither can detect code patterns that lead to costly code execution and deployment
- Detect variables that should be declared as constants, aiming to optimize as a way to consume less gas, and do not take space in storage.
- Functions that should be declared as externals. External functions allow the compiler to optimize the code.

Code Understanding

- Slither includes printers, which allow users to quickly understand what a contract does and how it is structured.
- Open source printers include:
  - The exportation of different graph-based representations, including the inheritance graph, the control flow graph, and the call graph of each contract.
  - A human-readable summary of the contracts, including the number of issues found and information about the code quality, such as cyclomatic complexity, or a minting restriction for ERC20 tokens.
  - A summary of the authorization accesses and the variables that can be changed by the contract’s owner.

Assisted Code Review

- Users can build third-party scripts and tools using Slither Python API3.
- A custom script can target contract-specific needs.
  - a user can ensure that a specific variable is never tainted by the parameter of a given function, or
  - that a function is reachable only via legitimate entry points.
- Third-party tools can use Slither internals to build more advanced analyses
  - symbolic execution on top of SlithIR, or
  - a conversion from SlithIR to another IR such as LLVM.

SlithIR

- SlithIR uses fewer than 40 instructions
- Arithmetic Operations:
  - `LV = RV BINARY RV`
  - `LV = UNARY RV`
- Mappings and Arrays:
  - `REF ← Variable [Index]`
- Structures:
  - `REF ← Variable · Member`

SlithIR

- Calls
  - `LV = L_CALL Destination Function [ARG..]` (low-level Solidity call)
  - `LV = H_CALL Destination Function [ARG..]` (high-level Solidity call)
  - `LV = LIB_CALL Destination Function [ARG..]` (library call)
  - `LV = S_CALL Function [ARG..]` (call to a inbuilt-Solidity function)
  - `LV = I_CALL Function [ARG..]` (call to an internal function)
  - `LV = DYN_CALL Variable [ARG..]` (call to an internal dynamic function)
  - `LV = E_CALL Event [ARG..]` (event call)
  - `LV = Send Destination` (Solidity *send*)
  - `Transfer Destination` (Solidity *transfer*)
- Additional Instructions: PUSH, CONVERT

```
using SafeMath for uint;
mapping(address => uint) balances;

function transfer(address to, uint val) public{
    balances[msg.sender] = balances[msg.sender].min(
        val);
    balances[to] = balances[to].add(val);
}
```

Solidity code example

```
Function transfer(address,uint256)
  Solidity: balances[msg.sender] = balances[msg.sender].sub
      (val)
  SlithIR:
    REF_0(uint256) -> balances[msg.sender]
    REF_1(uint256) -> balances[msg.sender]
    TMP_1(uint256) = LIB_CALL SafeMath.sub(REF_1, val)
    REF_0 := TMP_1(uint256) // dereferencing
  Solidity: balances[to] = balances[to].add(val)
  SlithIR:
    REF_3(uint256) -> balances[to]
    REF_4(uint256) -> balances[to]
    TMP_3(uint256) = LIB_CALL, dest:SafeMath.add(REF_4, val
        )
    REF_3 := TMP_3(uint256) // dereferencing
```

SlithIR code

Other IRs in the literature

- Scilla is the intermediate representation used by the Zilliqa blockchain.
- Michelson is the intermediate representation used in the Tezos blockchain.
- IELE is the representation developed for the Cardano blockchain.
- The Solidity authors are working on an intermediate representation called YUL, which is still a work in progress

Slither Results

- For the second experiment, using a dataset of 1000 contracts, the tools were run on each contract with a timeout of 120 seconds, using only reentrancy detectors.

		Slither	Securify	SmartCheck	Solhint
Accuracy	False positives	10.9%	25%	73.6%	91.3%
	Flagged contracts	112	8	793	81
	Detections per contract	3.17	2.12	10.22	2.16
Performance	Average execution time	0.79 ± 1	41.4 ± 46.3	10.9 ± 7.14	0.95 ± 0.35
	Timed out analyses	0%	20.4%	4%	0%
Robustness	Failed analyses	0.1%	11.2%	10.22%	1.2%
Reentrancy examples	DAO	✓	✗	✓	✗
	Spankchain	✓	✗	✗	✗

SmartCheck

SmartCheck: Static Analysis of Ethereum Smart Contracts

Sergei Tikhomirov
SnT, University of Luxembourg
Esch-sur-Alzette, Luxembourg
sergey.s.tikhomirov@gmail.com

Ekaterina Voskresenskaya
SmartDec
Moscow, Russia
voskresenskaya@smartdec.net

Ivan Ivanitskiy
SmartDec
Moscow, Russia
ivanitskiy@smartdec.net

Ramil Takhaviev
SmartDec
Moscow, Russia
tahaviev@smartdec.net

Evgeny Marchenko
SmartDec
Moscow, Russia
marchenko@smartdec.net

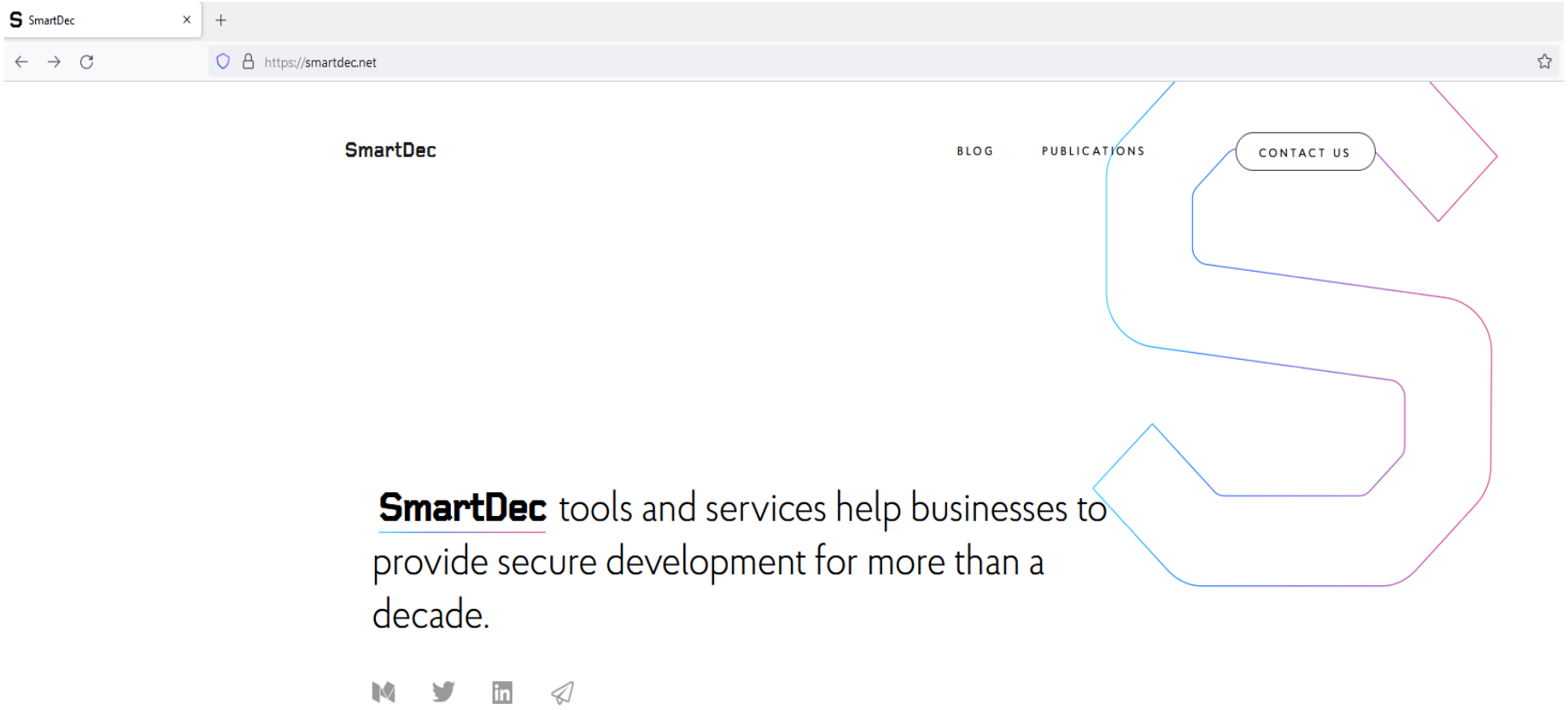
Yaroslav Alexandrov
SmartDec
Moscow, Russia
alexandrov@smartdec.net

ABSTRACT

Ethereum is a major blockchain-based platform for smart contracts – Turing complete programs that are executed in a decentralized network and usually manipulate digital units of value. Solidity is the most mature high-level smart contract language. Ethereum is a hostile execution environment, where anonymous attackers exploit bugs for immediate financial gain. Developers have a very

ACM Reference Format:
Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *WETSEB'18: WETSEB'18:IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194113.3194115

SmartCheck


SmartDec
https://smartdec.net
SmartDec
BLOG
PUBLICATIONS
CONTACT US
SmartDec tools and services help businesses to provide secure development for more than a decade.

Vulnerable Patterns

- Adversary can forcibly send ether to any account by mining or via self-destruct:

```
if (this.balance == 42 ether) { /* ... */} // bad
if (this.balance >= 42 ether) { /* ... */} // good
```

- Should check the return value and handle errors:

```
addr.send(42 ether); // bad
if (!addr.send(42 ether)) revert; // better
addr.transfer(42 ether); // good
```

- Condition should not depend on external function call. Callee may permanently fail, leading caller to fail.

```
function dos(address oracleAddr) public {
  badOracle = Oracle(oracleAddr);
  if (badOracle.answer() < 42) { revert; }
  // ...
}
```

Vulnerable Patterns

- Reentrancy:

```
pragma solidity 0.4.19;
contract Fund {
  mapping(address => uint) balances;
  function withdraw() public {
    if (msg.sender.call.value(balances[msg.sender])())
    balances[msg.sender] = 0;
  }
}
```

- Use “check-effects-interactions” pattern:
  - Check invariants
  - Update internal state
  - Communicate with external entities

```
function withdraw() public {
  uint balance = balances[msg.sender];
  balances[msg.sender] = 0;
  msg.sender.transfer(balance);
  // state reverted, balance restored if transfer fails
}
```

Vulnerable Patterns

- Malicious libraries: The pattern simply detects the library keyword
- Use of "*Tx.origin"* for authentication
- Transfer forwards all gas: addr.call.value(x)() transfers x ether and forwards all gas to addr, potentially leading to vulnerabilities like reentrancy

Functional & Operational issues

- Timestamp dependence.

```
if (now % 2 == 0) winner = pl1; else winner = pl2;
```

- Unsafe type inference.

```
for (var i = 0; i < array.length; i++) { /*...*/ }
```

```
for (uint256 i = 0; i < array.length; i++) { /*...*/ }
```

- Costly loop.

```
for (uint256 i = 0; i < array.length; i++) { costlyF(); }
```

Development Issue

- Compiler version not fixed.

```
pragma solidity ^0.4.19;  // bad: 0.4.19 and above
pragma solidity 0.4.19;   // good: 0.4.19 only
```

- Redundant Fallback Function

```
function () payable { throw; }
```

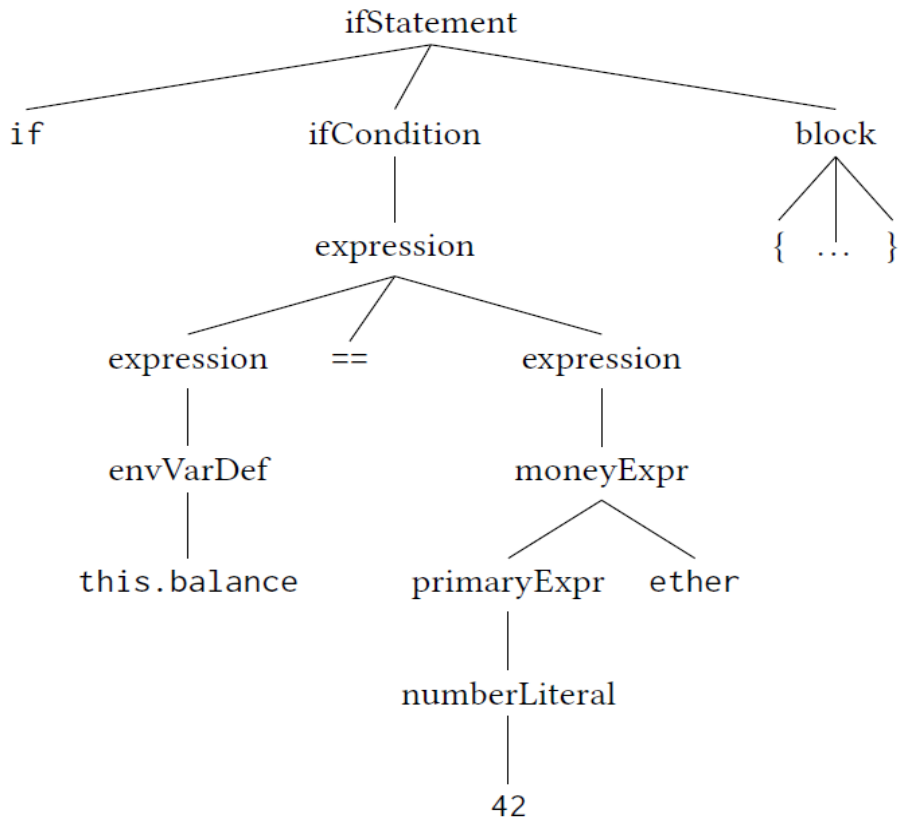
- Misconception about private modifier
- Style guide violation
- Implicit visibility level

SmartCheck Components

- SmartCheck runs lexical and syntactical analysis on Solidity source code.
- It uses ANTLR and a custom Solidity grammar to generate an XML parse tree as an intermediate representation (IR).
- We detected vulnerability patterns by using XPath queries on the IR.
- Line numbers are stored as XML attributes and help localize findings in source code.
- IR attributes can be enriched with additional information when new analysis methods are implemented.

Example

```
if (this.balance == 42 ether){...}.
```




```
//expression[expression//envVarDef
[matches(text()[1],"^this.balance$")]]
[matches(text()[1],"^==|!=$")]
```

Securify

SECURIFY: Practical Security Analysis of Smart Contracts

Petar Tsankov
ETH Zurich
petar.tsankov@inf.ethz.ch

Andrei Dan
ETH Zurich
andrei.dan@inf.ethz.ch

Dana Drachsler-Cohen
ETH Zurich
dana.drachsler@inf.ethz.ch

Arthur Gervais*
Imperial College London
a.gervais@imperial.ac.uk

Florian Bünzli
ETH Zurich
fbuenzli@student.ethz.ch

Martin Vechev
ETH Zurich
martin.vechev@inf.ethz.ch

ABSTRACT

Permissionless blockchains allow the execution of arbitrary programs (called *smart contracts*), enabling mutually untrusted entities to interact without relying on trusted third parties. Despite their potential, repeated security concerns have shaken the trust in handling billions of USD by smart contracts.

To address this problem, we present SECURIFY, a security analyzer for Ethereum smart contracts that is scalable, fully automated, and able to prove contract behaviors as safe/unsafe with respect to a given property. SECURIFY's analysis consists of two steps. First, it symbolically analyzes the contract's dependency graph to extract precise semantic information from the code. Then, it checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not. To enable extensibility, all patterns are specified in a designated domain-specific language.

SECURIFY is publicly released, it has analyzed > 18K contracts submitted by its users, and is regularly used to conduct security

July 2017 [10], and few months later 280M were frozen due to a bug in the very same wallet [13]. It is apparent that effective security checkers for smart contracts are urgently needed.

Key Challenges. The main challenge in creating an effective security analyzer for smart contracts is the Turing-completeness of the programming language, which renders automated verification of arbitrary properties undecidable. To address this issue, current automated solutions tend to rely on fairly generic testing and symbolic execution methods (e.g., Oyente [39] and Mythril [16]). While useful in some settings, these approaches come with several drawbacks: *(i)* they can miss critical violations (due to under-approximation), *(ii)* yet, can also produce false positives (due to imprecise modeling of domain-specific elements [30]), and *(iii)* they can fail to achieve sufficient code coverage on realistic contracts (Oyente achieves only 20.2% coverage on the popular Parity wallet [17]). Overall, these drawbacks place a significant burden on their users, who must inspect all reports for false alarms and worry about unreported vul-

Securify

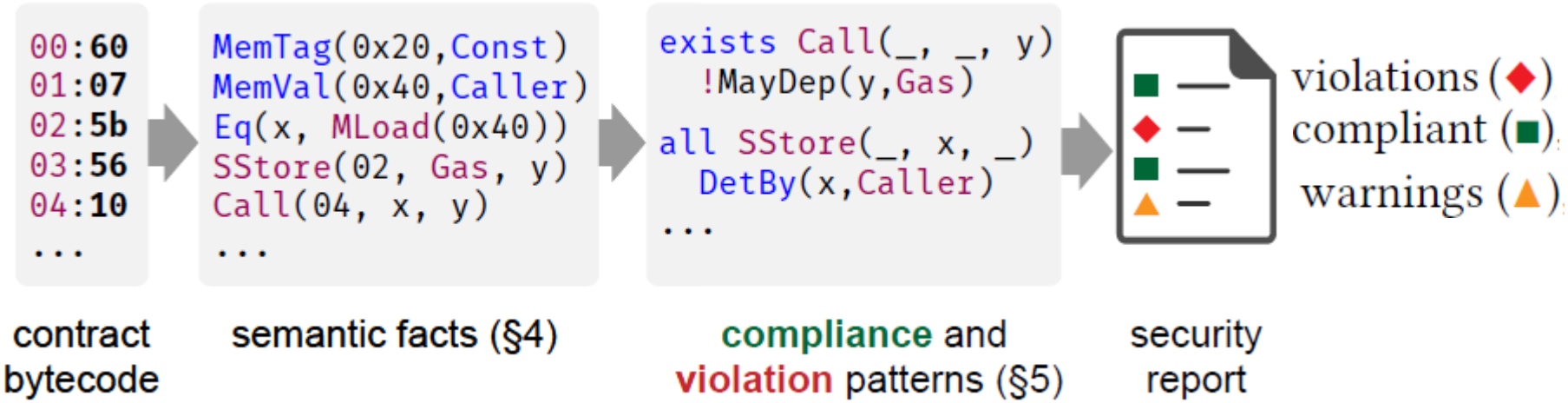
The key idea is to define two kinds of patterns that mirror a given security property:

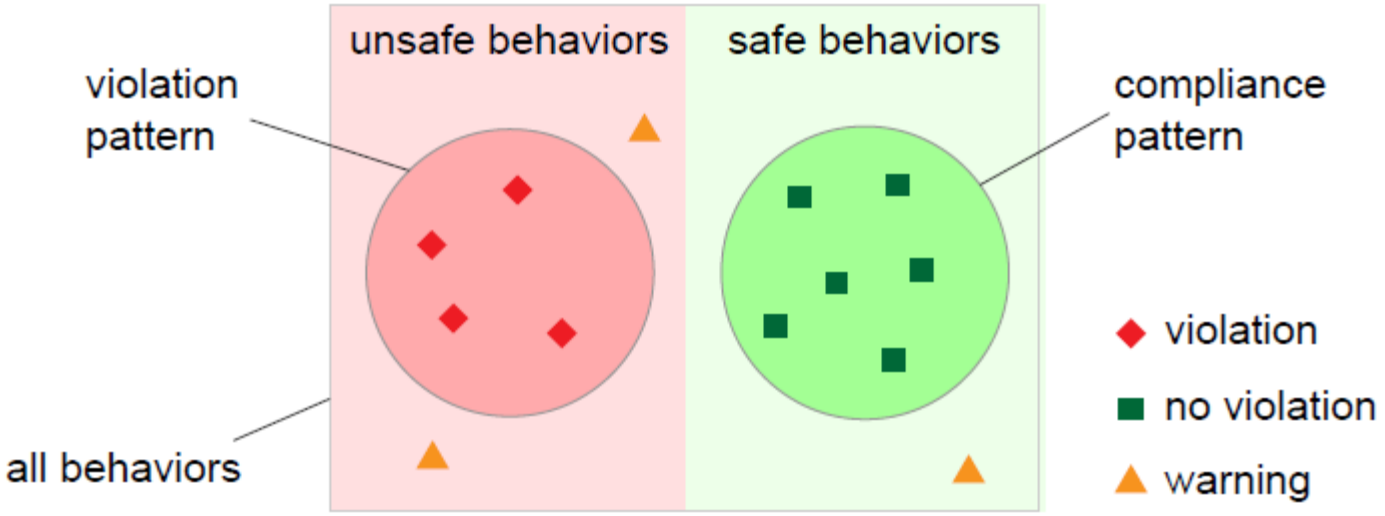
- Compliance patterns, which imply the satisfaction of the property
- Violation patterns, which imply its negation.

To check these patterns, Securify does the following:

- Symbolically encodes the dependence graph of the contract in stratified Datalog.
- Uses off-the-shelf Datalog solvers to efficiently analyze the code.

Securify Approach





Securify System

- Input to Securify:
  - EVM Bytecode
  - A set of Security Patterns (Compliance and Violations in designated DSL)
- Step 1: Decompiling EVM Bytecode.
- Step 2: Inferring Semantic Facts.
  - Analyzes the contract to infer semantic facts, including data and control-flow dependencies, which hold over all behaviors of the contract.
  - For example, the fact MayDepOn(b, dataload) captures that the value of variable b may depend on the value returned by the instruction dataload.
  - Further, the fact Eq(c, 0) captures that variable c equals the constant 0.
  - Semantic facts is specified declaratively in stratified Datalog (declarative logic lang.) and is fully automated using existing scalable engines.
- Step 3: Checking Security Patterns.
  - Set of compliance and violation security patterns expressed in a specialized DSL (fragment of logical formulas over the semantic facts)

Example Violation Pattern

- In sstore(c, b), c is the storage offset of the owner field, and b is the value to store.

- The violation pattern matches if there exists some “sstore” instruction for which both the storage offset, denoted X, and the execution of this instruction, identified by its label L, do not depend on the result of the caller instruction in any possible execution of the contract.

- Since the instruction caller retrieves the address of the transaction sender, matching this pattern implies that any user can reach this sstore and change the value of owner.

$$some\ \mathrm{sstore}(L, X, \_).\ \neg MayDepOn(X, \mathrm{caller}) \wedge \neg MayDepOn(L, \mathrm{caller})$$

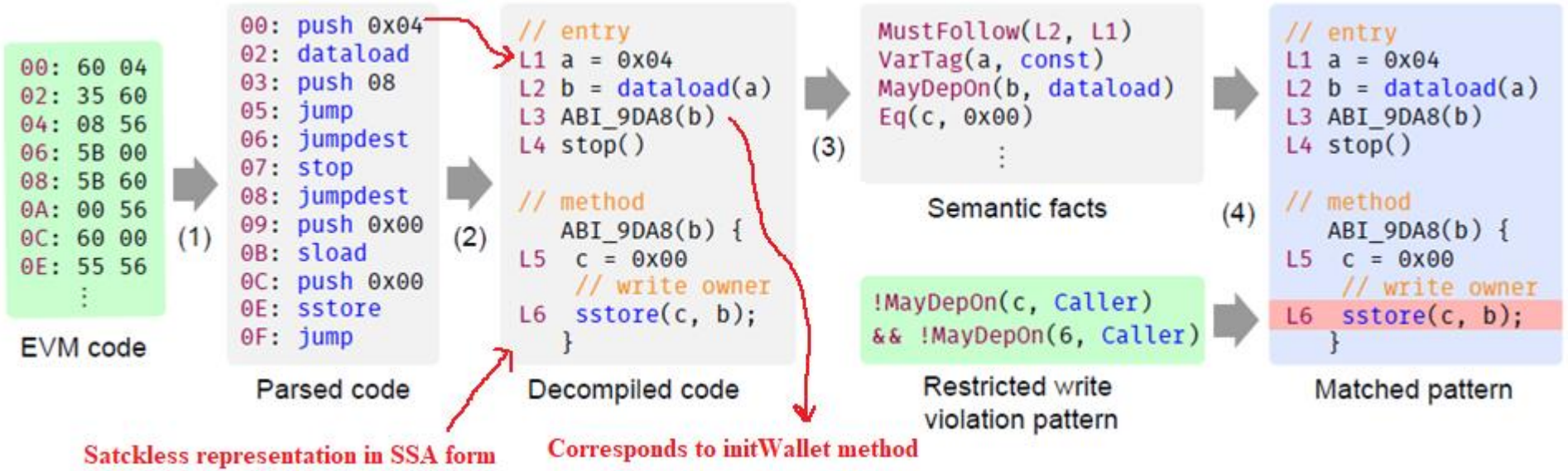
Securify System

- Output of Securify:
  - Violation
  - No Violation
  - Warning
- Limitations:
  - Can not reason numerical properties, e.g. overflow
  - Can not reason reachability properties

```
contract OwnableWallet {
  address owner;

  // called by the constructor
  function initWallet(address _owner) {
    owner = _owner; // any user can change owner
    // more setup
  }

  // function that allows the owner to withdraw ether
  function withdraw(uint _amount) {
    if (msg.sender == owner) {
      owner.transfer(_amount);
    }
  }
  // ...
}
```



Facts and Inference Rules

Semantic fact	Intuitive meaning
	Flow Dependencies
MayFollow(L_1, L_2)	Instruction at label L_2 may follow that at label L_1.
MustFollow(L_1, L_2)	Instruction at label L_2 must follow that at label L_1.
	Data Dependencies
MayDepOn(Y, T)	The value of Y may depend on tag T.
Eq(Y, T)	The values of Y and T are equal.
DetBy(Y, T)	For different values of T the value of Y is different.

Figure : The semantic facts: L_1 and L_2 are labels, Y is a variable, and T is a tag (a variable or a label).

Data-dependency may-analysis

$$
\begin{aligned}
\mathit{MayDepOn}(Y, X) &\Leftarrow \mathrm{assign}(\_, Y, X)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{assign}(\_, Y, X), \mathit{MayDepOn}(X, T)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{op}(\_, Y, \ldots, X, \ldots), \mathit{MayDepOn}(X, T)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{mload}(L, Y, O), \mathit{isConst}(O), \mathit{MemTag}(L, O, T)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{mload}(L, Y, O), \neg\mathit{isConst}(O), \mathit{MemTag}(L, \_, T)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{mload}(L, Y, O), \mathit{MemTag}(L, \top, T)\\
\mathit{MayDepOn}(Y, T) &\Leftarrow \mathrm{assign}(L, Y, \_), \mathit{Taint}(\_, L, X), \mathit{MayDepOn}(X, T)
\end{aligned}
$$

Memory analysis inference

$$
\begin{aligned}
\mathit{MemTag}(L, O, T) &\Leftarrow \mathrm{mstore}(L, O, X), \mathit{isConst}(O), \mathit{MayDepOn}(X, T)\\
\mathit{MemTag}(L, \top, T) &\Leftarrow \mathrm{mstore}(L, O, X), \neg\mathit{isConst}(O), \mathit{MayDepOn}(X, T)\\
\mathit{MemTag}(L, O, T) &\Leftarrow \mathit{Follow}(L_1, L), \mathit{MemTag}(L_1, O, T),\\
&\qquad \neg\mathit{ReassignMem}(L, O)\\
\mathit{ReassignMem}(L, O) &\Leftarrow \mathrm{mstore}(L, O, \_), \mathit{isConst}(O)
\end{aligned}
$$

Implicit control-flow analysis

$$
\begin{aligned}
\mathit{Taint}(L_1, L_2, X) &\Leftarrow \mathrm{goto}(L_1, X, L_2)\\
\mathit{Taint}(L_1, L_2, X) &\Leftarrow \mathrm{goto}(L_1, X, \_), \mathit{Follow}(L_1, L_2)\\
\mathit{Taint}(L_1, L_2, X) &\Leftarrow \mathit{Taint}(L_1, L_3, X), \mathit{Follow}(L_3, L_2), \neg\mathit{Join}(L_1, L_2)
\end{aligned}
$$

Figure : Partial inference rules for *MayDepOn*: the Datalog variable X ranges over contract variables, L ranges over instruction labels, $\top$ represents an unknown offset, and T ranges over tags.

Security Language

$$
\begin{array}{lll}
\varphi & ::= & \mathrm{instr}(L, Y, X, \ldots, X) \mid \mathit{Eq}(X, T) \mid \mathit{DetBy}(X, T) \\
& \mid & \mathit{MayDepOn}(X, T) \mid \mathit{MayFollow}(L, L) \mid \mathit{MustFollow}(L, L) \\
& \mid & \mathit{Follow}(L, L) \mid \exists X.\varphi \mid \exists L.\varphi \mid \exists T.\varphi \mid \neg\varphi \mid \varphi \wedge \varphi
\end{array}
$$

Semantics. Patterns are interpreted by checking the inferred semantic facts:

- Quantifiers and connectors are interpreted as usual.
- Flow- and data-dependency predicates are interpreted as defined in Section 4; i.e., a semantic fact holds if and only if it is contained in the Datalog fixed-point.

VeriSolid: Correct-by-Design Smart Contracts for Ethereum

Anastasia Mavridou[1], Aron Laszka[2],
Emmanouela Stachtiari[3], and Abhishek Dubey[1]

[1] Vanderbilt University
[2] University of Houston
[3] Aristotle University of Thessaloniki

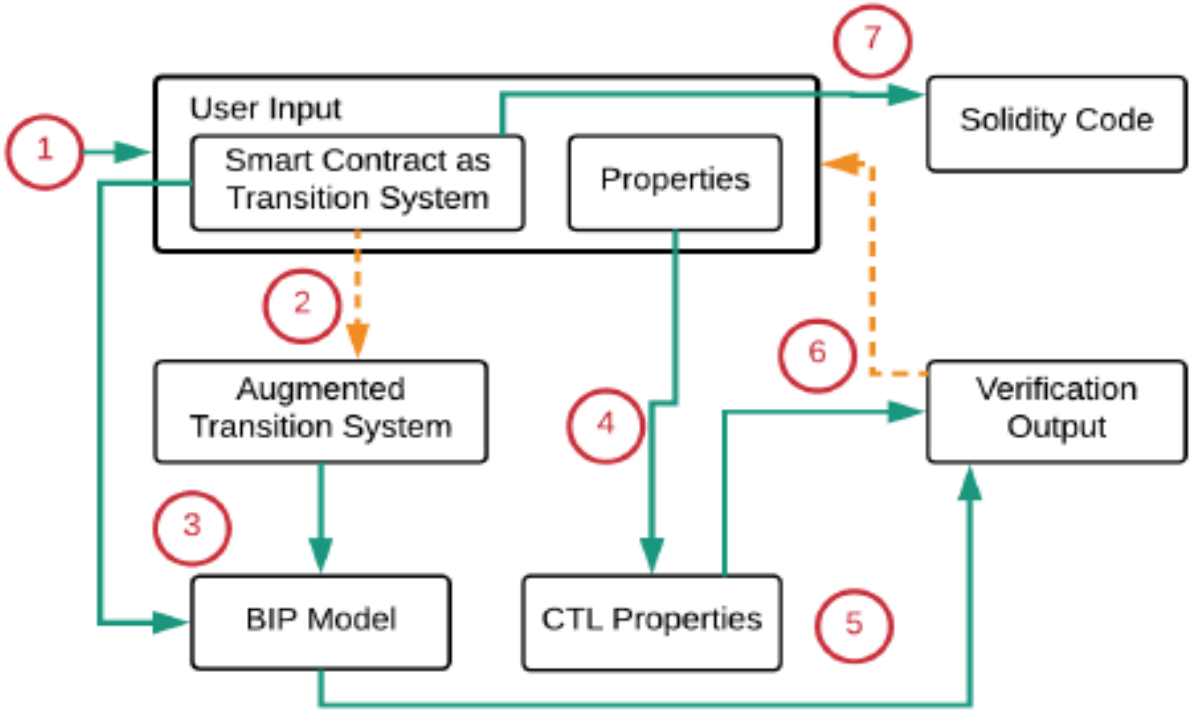
Accepted for publication in the proceedings of the
23rd International Conference on Financial Cryptography and Data Security
(FC 2019).

Abstract. The adoption of blockchain based distributed ledgers is growing fast due to their ability to provide reliability, integrity, and auditability without trusted entities. One of the key capabilities of these emerging platforms is the ability to create self-enforcing smart contracts. However, the development of smart contracts has proven to be error-prone in practice, and as a result, contracts deployed on public platforms are often riddled with security vulnerabilities. This issue is exacerbated by the de-

VeriSolid Architecture

(An extension of FSolidM)

- Allows developers to reason about and verify contract behavior at a high level of abstraction.
- Allows the generation of Solidity code from the veried models, which enables the correct-by-design development of smart contracts.

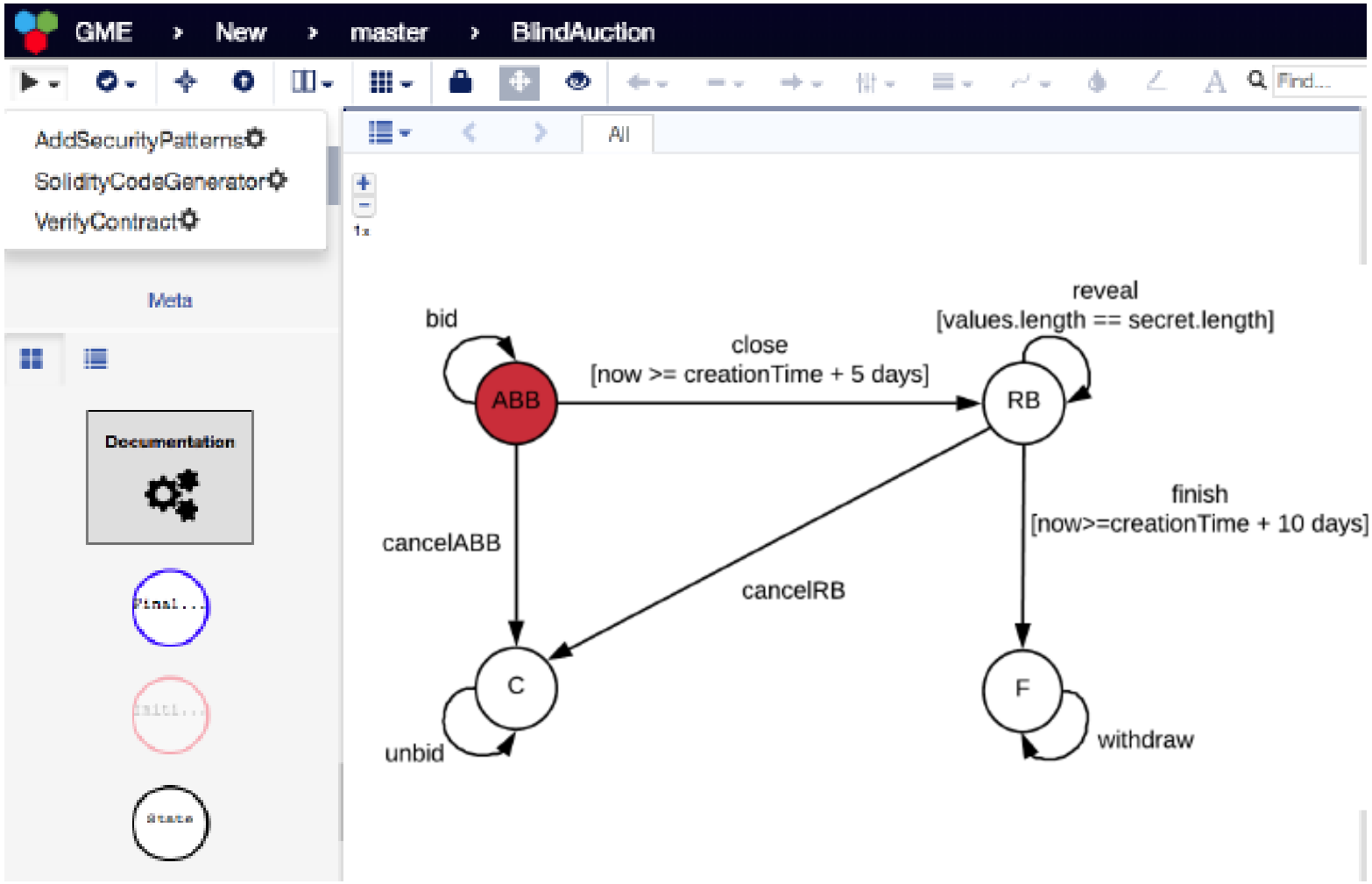


VeriSolid Workflow

Step 1: Inputs:

- A contract specification containing
  - a graphically specified transition system
  - variable declarations, actions, and guards specified in Solidity.
- A list of properties to be verified, which can be expressed using predefined natural-language like templates.

VeriSolid Editor



Example Properties

– **bid** cannot happen after **close**.
– **cancelABB; cancelRB** cannot happen after **finish**,
where **cancelABB; cancelRB** means **cancelABB** ∪ **cancelRB**.

– if **withdraw**`.msg.sender.transfer(amount);` happens,
withdraw`.msg.sender.transfer(amount);` can happen only after
withdraw`.pendingReturns[msg.sender]=0;` happens.

Transition System

Table 1. Summary of Notation for Solidity Code

Symbol	Meaning
$\mathbb{T}$	set of Solidity types
$\mathbb{I}$	set of valid Solidity identifiers
$\mathbb{D}$	set of Solidity event and custom-type definitions
$\mathbb{E}$	set of Solidity expressions
$\mathbb{C}$	set of Solidity expressions without side effects
$\mathbb{S}$	set of supported Solidity statements

Transition System

Definition 1. *A transition-system* initial smart contract *is a tuple* $(D, S, S_F, s_0, a_0, a_F, V, T)$, *where*

- $D \subset \mathbb{D}$ *is a set of custom event and type definitions;*
- $S \subset \mathbb{I}$ *is a finite set of states;*
- $S_F \subset S$ *is a set of final states;*
- $s_0 \in S$, $a_0 \in \mathbb{S}$ *are the initial state and action;*
- $a_F \in \mathbb{S}$ *is the fallback action;*
- $V \subset \mathbb{I} \times \mathbb{T}$ *contract variables (i.e., variable names and types);*
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I} \times \mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ *is a transition relation, where each transition* $\in T$ *includes:*
  - *transition name* $t^{name} \in \mathbb{I}$;
  - *source state* $t^{from} \in S$;
  - *parameter variables (i.e., arguments)* $t^{input} \subseteq \mathbb{I} \times \mathbb{T}$;
  - *transition guard* $g_t \in \mathbb{C}$;
  - *return type* $t^{output} \in (\mathbb{T} \cup \emptyset)$;
  - *action* $a_t \in \mathbb{S}$;
  - *destination state* $t^{to} \in S$.

Transition System Semantics

$$\text{TRANSITION}\quad \frac{\begin{array}{c} t \in T, \quad name = t^{name}, \quad s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M) \\ \text{Eval}(\sigma, g_t) \rightarrow \langle (\hat{\sigma}, N), \texttt{true} \rangle \\ \langle (\hat{\sigma}, N), a_t \rangle \rightarrow \langle (\hat{\sigma}', N), \cdot \rangle \\ \hat{\sigma}' = (\Psi', M'), \quad s' = t^{to} \end{array}}{\langle (\Psi, s), name\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi', s', \cdot) \rangle}$$

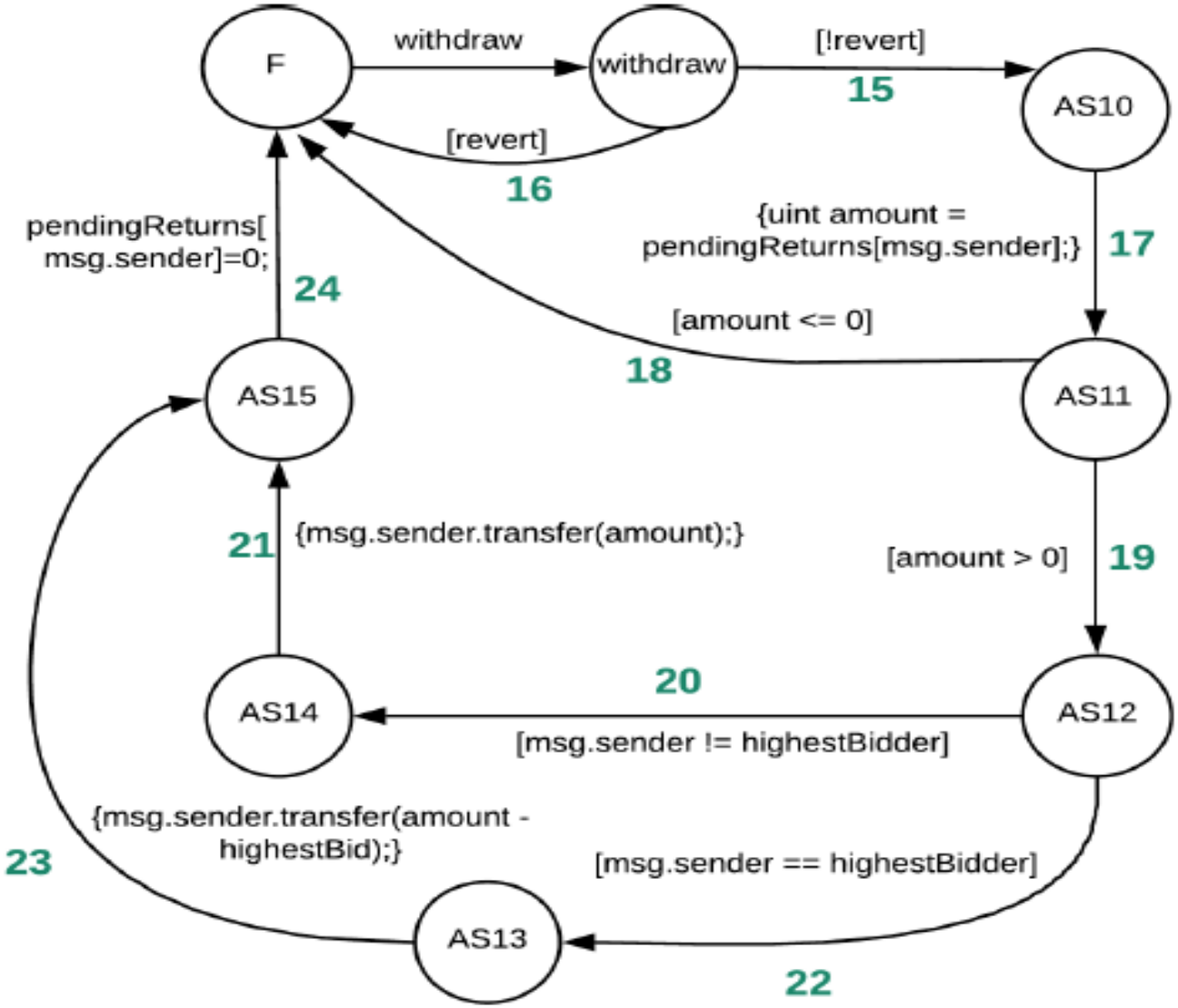
VeriSolid Workflow

- **Step 2:** The verification of the specified properties requires the generation of an augmented contract model.
  - Replace the initial action a_0 and the fallback action a_F with transitions.
  - Replace transitions that have complex statements as actions with a series of transitions that have only simple statements.

Definition 2. *An* augmented contract *is a tuple* (D, S, S_F, s_0, V, T), *where*

- $D \subset \mathbb{D}$ *is a set of custom event and type definitions;*
- $S \subset \mathbb{I}$ *is a finite set of states;*
- $S_F \subset S$ *is a set of final states;*
- $s_0 \in S$, *is the initial state;*
- $V \subset \mathbb{I} \times \mathbb{T}$ *contract variables (i.e., variable names and types);*
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I} \times \mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ *is a transition relation (i.e., transition name, source state, parameter variables, guard, return type, action, and destination state).*

Augmented Model for Withdraw



VeriSolid Workflow

- **Step 3:** The **Behavior-Interaction-Priority (BIP) model** of the contract (augmented or not) is automatically generated.

- **Step 4:** Properties are automatically translated to Computational Tree Logic (CTL).

VeriSolid Temporal Properties

(i) `bid` cannot happen after `close`:

$$\mathtt{AG}(close \rightarrow \mathtt{AG}\neg bid)$$

(ii) `cancelABB` or `cancelRB` cannot happen after `finish`:

$$\mathtt{AG}(finish \rightarrow \mathtt{AG}\neg(cancelRB \vee cancelABB))$$

(iii) `withdraw` can happen only after `finish`:

$$\mathtt{A}[\neg withdraw\ \mathtt{W}\ finish]$$

(iv) `finish` can happen only after `close`:

$$\mathtt{A}[\neg finish\ \mathtt{W}\ close]$$

VeriSolid Workflow

- **Step 5:** Model is verified for deadlock freedom or other properties using tools from the BIP tool-chain or nuXmv.
- **Step 6:** Refinement if the properties are not satisfied.
- **Step 7:** Equivalent Solidity code of the contract is automatically generated.

A Study of Static Analysis Tools for Ethereum Smart Contracts

António Pedro Cruz Monteiro*
Instituto Superior Técnico, Universidade de Lisboa
apedrocruz@tecnico.ulisboa.pt

***Abstract*—Blockchain technology has been receiving considerable attention from industry and academia, for it promises to disrupt the digital online world by enabling a democratic, open, and scalable digital economy based on decentralized distributed consensus without the intervention of third-party trusted authorities. Smart contracts, computer programs executed on top of a blockchain, are at the core of this technology, for they allow the creation of new distributed applications. Millions of smart contracts have been deployed on Ethereum, a major blockchain platform for smart contracts, and although they are seen with great potential, smart contracts have been known to have several security problems. Recent attacks on smart contracts and critical vulnerabilities discovered show us the huge financial loss they can impose on the users and warn us for the necessity of having methods that lead to secure and reliable implementations. Over the last few years several static analysis tools have been developed specifically targeting Ethereum smart contracts. In this dissertation, we present a review of state-of-the-art static analysis tools and introduce SmartBugs, a new extendable execution framework, created to facilitate the integration and comparison**

are automatically enforced by the consensus mechanism of the blockchain, using incentives and cryptography. These immutable pieces of code, when deployed on the blockchain, allow parties to manifest contract terms in the form of program code.

Ethereum [3], which was introduced with the proclaimed intention of being a platform aimed to support smart contracts, is the most famous network to deploy smart contracts, where they written in a Turing-complete language. Solidity is the most used language by developers to create smart contracts on Ethereum. While smart contracts can represent a great improvement to our society model, as they do not rely on gained trust, they have shown several major security vulnerabilities. In a study performed on nearly one million Ethereum smart contracts, *34,200* of them were flagged as vulnerable [4]. A different study showed that of *19,366* smart contracts analysed, also in Ethereum, *8,833* (around 46%)

Research Questions

- **RQ1.** What are the state-of-the-art analysis tools for Ethereum smart contracts?
- **RQ2.** What is the precision of the state-of-the-art analysis tools in detecting vulnerabilities on Ethereum smart contracts?
- **RQ3.** Which categories of vulnerabilities are most detected?
- **RQ4.** How many vulnerable contracts are present in the Ethereum blockchain?
- **RQ5.** How long do the tools require to analyse the smart contracts?
- **RQ6.** How can we improve a state-of-the-art analysis tool?

Table : Tools identified as potential candidates for this study.

#	Tools	Tool URLs
1	contractLarva [7]	https://github.com/gordonpace/contractLarva
2	E-EVM [8]	https://github.com/pisocrob/E-EVM
3	Echidna	https://github.com/crytic/echidna
4	Erays [9]	https://github.com/teamnsrg/erays
5	Ether [10]	N/A
6	Ethersplay	https://github.com/crytic/ethersplay
7	EtherTrust [11]	https://www.netidee.at/ethertrust
8	EthIR [12]	https://github.com/costa-group/EthIR
9	FSolidM [13]	https://github.com/anmavrid/smart-contracts
10	Gasper [14]	N/A
11	HoneyBadger [15]	https://github.com/christoftorres/HoneyBadger
12	KEVM [16]	https://github.com/kframework/evm-semantics
13	MadMax [17]	https://github.com/nevillegrech/MadMax
14	Maian [4]	https://github.com/MAIAN-tool/MAIAN
15	Manticore [18]	https://github.com/trailofbits/manticore/
16	Mythril [19]	https://github.com/ConsenSys/mythril-classic

17	Octopus	https://github.com/quoscient/octopus
18	Osiris [20]	https://github.com/christoftorres/Osiris
19	Oyente [5]	https://github.com/melonproject/oyente
20	Porosity [21]	https://github.com/comaeio/porosity
21	rattle	https://github.com/crytic/rattle
22	ReGuard [22]	N/A
23	Remix	https://github.com/ethereum/remix
24	SASC [23]	N/A
25	sCompile [24]	N/A
26	Securify [25]	https://github.com/eth-sri/securify
27	Slither [26]	https://github.com/crytic/slither
28	Smartcheck [27]	https://github.com/smartdec/smartcheck
29	Solgraph	https://github.com/raineorshine/solgraph
30	Solhint	https://github.com/protofire/solhint
31	SolMet [28]	https://github.com/chicxurug/SolMet-Solidity-parser
32	teEther [29]	https://github.com/nescio007/teether
33	Vandal [30]	https://github.com/usyd-blockchain/vandal
34	VeriSol [31]	https://github.com/microsoft/verisol
35	Zeus [32]	N/A

Selection Criteria

- *Criterion #1.* [Available and CLI] The tool is publicly available and supports a command-line interface (CLI).
- *Criterion #2.* [Compatible Input] The tool takes as input a Solidity contract. This excludes tools that only consider EVM bytecode.
- *Criterion #3.* [Only Source] The tool requires only the source code of the contract to be able to run the analysis. This excludes tools that require a test suite or contracts annotated with assertions.
- *Criterion #4.* [Vulnerability Finding] The tool identifies vulnerabilities or bad practices in contracts. This excludes tools that are described as analysis tools, but only construct artifacts such as control flow graphs.
- *Criterion #5.* [Static Analysis] The tool only executes static analysis techniques to find security issues.

Inclusion criteria	Tools that violate criteria
Available and CLI (C1)	Ether, Gasper, ReGuard, Remix, SASC, sCompile, teEther, Zeus
Compatible Input (C2)	MadMax, Vandal
Only Source (C3)	Echidna, VeriSol
Vulnerability Finding (C4)	contractLarva, E-EVM, Erays, Ethersplay, EtherTrust, EthIR, FSolidM, KEVM, Octopus, Porosity, rattle, Solgraph, SolMet, Solhint
Static Analysis (C5)	contractLarva, Echidna, MAIAN, Manticore, ReGuard

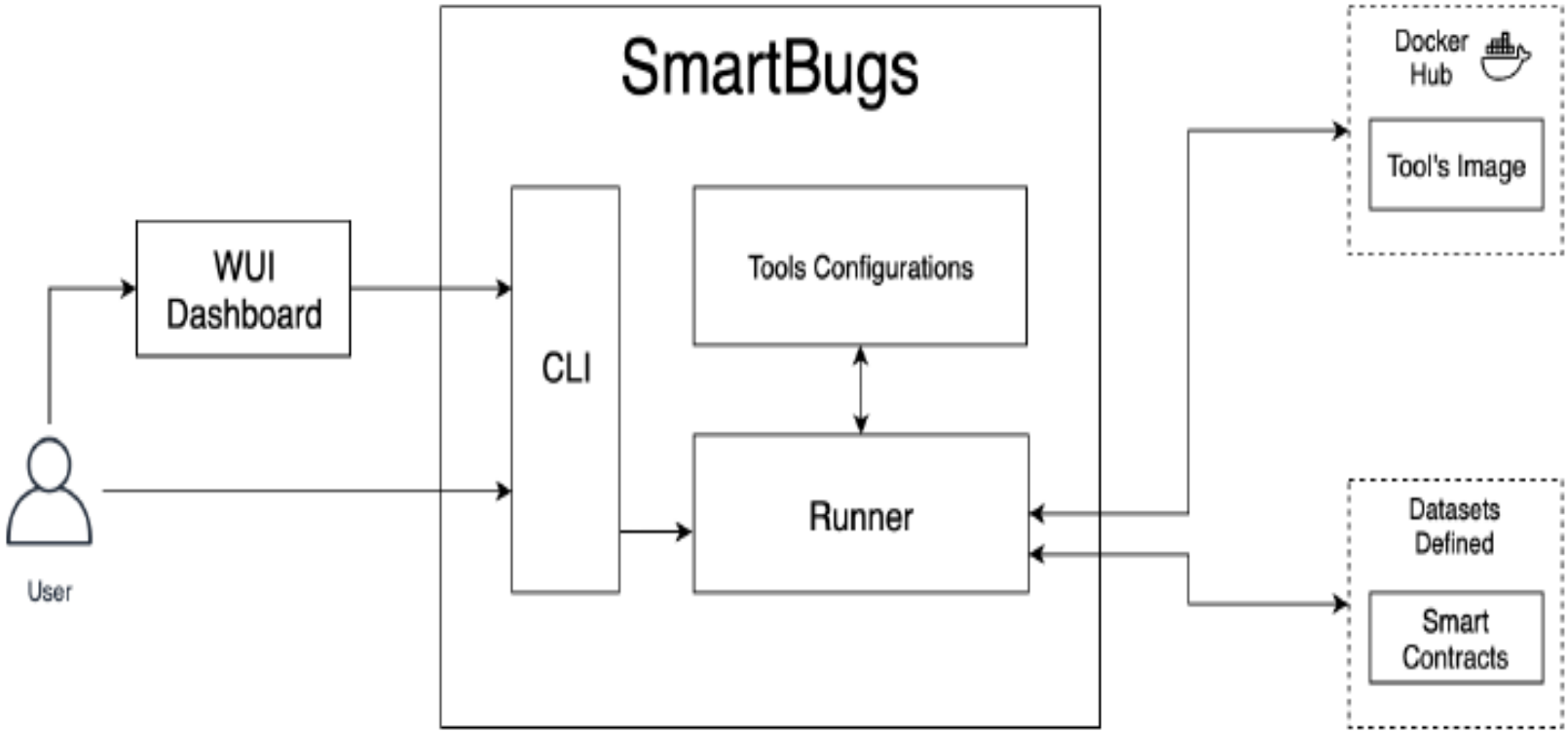
Table : Excluded analysis tools based on our inclusion criteria.

Tools that meet criteria	HoneyBadger, Mythril, Osiris, Oyente, Securify, Slither, Smartcheck

Table : Included analysis tools based on our inclusion criteria.

SmartBugs Architecture

- New tool can be added in the form of docker image of the tool.



SmartBugs

SmartBugs has the following features:

- A plugin system to easily add new analysis tools, based on Docker images;
- A plugin system to easily add new named datasets;
- Bulk analysis with any number of tools available;
- CLI and WUI Dashboard;
- Parallel execution of the tools to speed up the execution time; *[Contribution]*
- A parser mechanism that normalizes the way the tools are outputting the results; *[Contribution]*

[4]SmartBugs (including $SB^{CURATED}$): https://github.com/smartbugs/smartbugs

[5]SmartBugs WUI: https://github.com/smartbugs/smartbugs-dashboard

SB$^{\text{CURATED}}$: *A Dataset of 143 Vulnerable Smart Contracts*

Category	**Contracts**	**Vulns**	**LoC**
Access Control	17	19	899
Arithmetic	14	24	304
Bad Randomness	8	31	1,079
Denial of service	6	7	177
Front running	4	7	137
Reentrancy	31	32	2,164
Short addresses	1	1	18
Time manipulation	5	7	100
Unchecked low level calls	53	60	4055
Other	3	3	115
Total	143	191	9,048

Evaluation on SB CURATED

Category	HoneyBadger	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck	Total
Access Control	0 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,758 7%
Arithmetic	1 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,590 79%
Denial of Service	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 0%	8,454 17%	496 1%	308 0%	2,033 4%	8,764 18%	847 1%	14,747 31%
Time Manipulation	0 0%	0 0%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,005 8%
Unchecked Low Calls	0 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,655 30%
Other	26 0%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,091 59%
Total	46 0%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,549 93%

Vulnerabilities identified per category by each tool.

- It shows that the tools can accurately detect vulnerabilities of the categories Arithmetic, Reentrancy, Time Manipulation, Unchecked Low Level Calls and Other.
- However, they were not accurate in detecting vulnerabilities of the categories Access Control, Denial of Service, and Front Running.
- Unfortunately, we found out that none of the 7 tools were able to detect vulnerabilities of the categories Bad Randomness and Short Addresses.

Evaluation on $_{SB}^{CURATED}$

Category	HoneyBadger	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck	Total
Access Control	0	24	0	0	6	20	3	53
Arithmetic	0	92	62	69	0	0	23	246
Denial of Service	0	0	27	11	0	2	19	59
Front Running	0	21	0	0	55	0	0	76
Reentrancy	0	16	5	5	32	15	7	80
Time Manipulation	0	0	4	5	0	5	2	16
Unchecked Low Level Calls	0	30	0	0	21	13	14	78
Other	5	32	0	0	0	28	8	73
Total	5	215	98	90	114	83	76	681

Total number of detected vulnerabilities by each tool, including vulnerabilities not tagged in the dataset.

- The tool Mythril has the best accuracy among the 7 tools.
- Mythril detects 27% of all the vulnerabilities when the average of all tools is 12%.
- Moreover, Mythril, Slither, and SmartCheck are the tools that detect the largest number of different categories (5 categories).
- Despite its good results, Mythril is not powerful enough to replace all the tools.
- By combining the detection abilities of all the tools, they succeed to detect 42% of all the vulnerabilities.
- The combination of Mythril and Slither can detect 42 (37%) of all the vulnerabilities (trade-off between performance vs computational cost).

Tool Precision

- False Negative Rate (FNR): FNR = FN / (FN + TP)
- False Discovery Rate (FDR): FDR = FP / (TP + FP)
- Sensitivity = 1 - FNR
- Precision = 1 - FDR

	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck
TP	88	59	46	31	61	49
FP	78	39	44	35	22	27
FN	79	108	121	136	106	118
FDR	46,4%	39,8%	48,9%	53,0%	26,5%	35,5%
FNR	47,3%	64,7%	72,5%	81,4%	63,5%	70,7%
Precision	53,6%	60,2%	51,1%	47%	73,5%	64,5%
Sensitivity	52,7%	35,3%	27,5%	18,6%	36,5%	29,3%

- Mythril shows the best sensitivity, having the most number of true positives.
- Slither is the tool that presents best precision (73,5%), followed by SmartCheck.

Evaluation on SB WILD

Category	HoneyBadger	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck	Total
Access Control	0 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,758 7%
Arithmetic	1 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,590 79%
Denial of Service	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 0%	8,454 17%	496 1%	308 0%	2,033 4%	8,764 18%	847 1%	14,747 31%
Time Manipulation	0 0%	0 0%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,005 8%
Unchecked Low Calls	0 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,655 30%
Other	26 0%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,091 59%
Total	46 0%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,549 93%

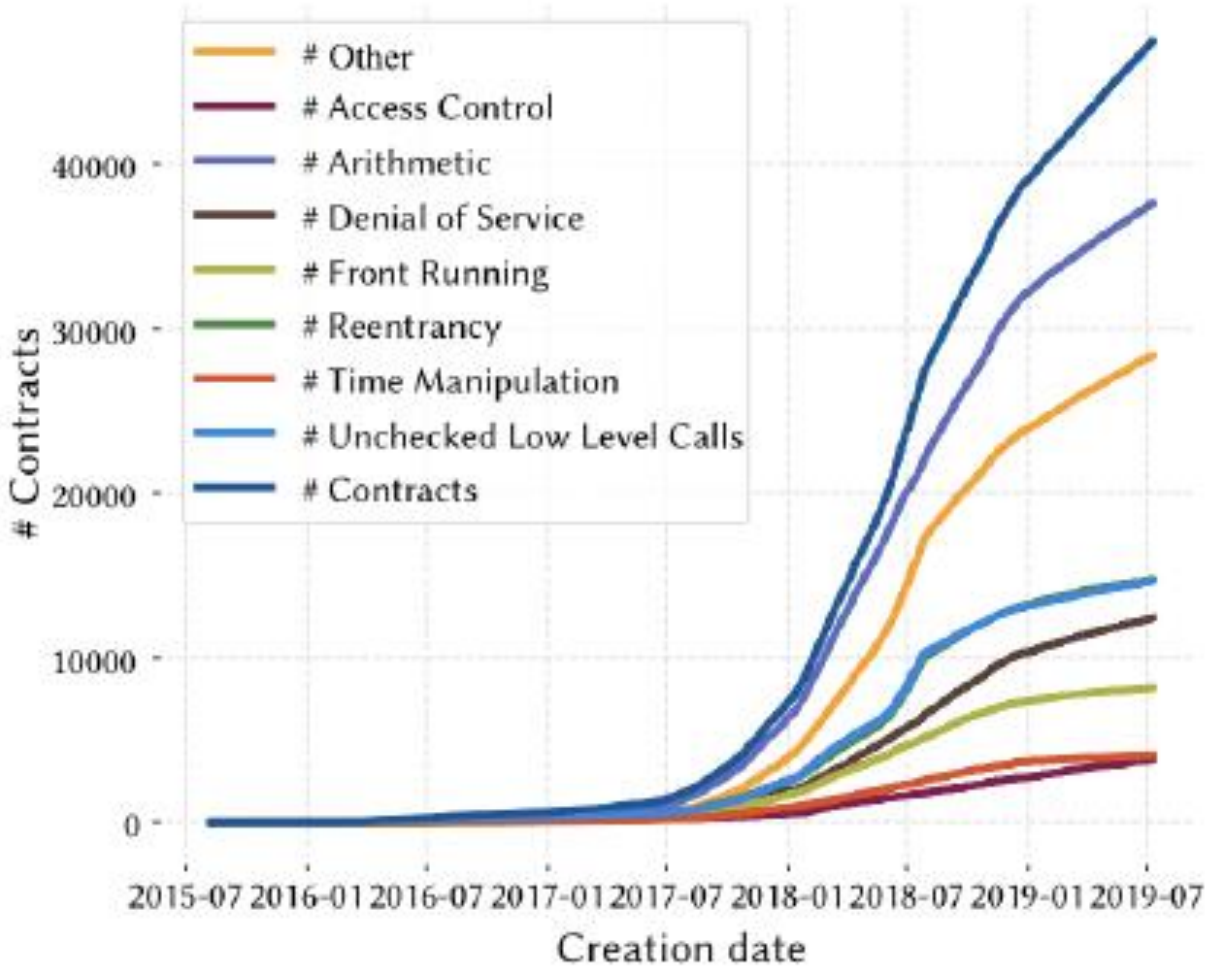
Table : Vulnerabilities identified per category by each tool.

- Dataset collected from Ethereum: 47,518 contracts
- It shows that the 7 tools are able to detect eight different categories of vulnerabilities.
- In total, 44.549 contracts (93%) have at least one vulnerability detected by one of the 7 tools. Such a high number of vulnerable contracts suggests the presence of a considerable number of false positives.
- Oyente is the approach that identifies the highest number of contracts as vulnerable (73%), mostly due to vulnerabilities in the Arithmetic category.

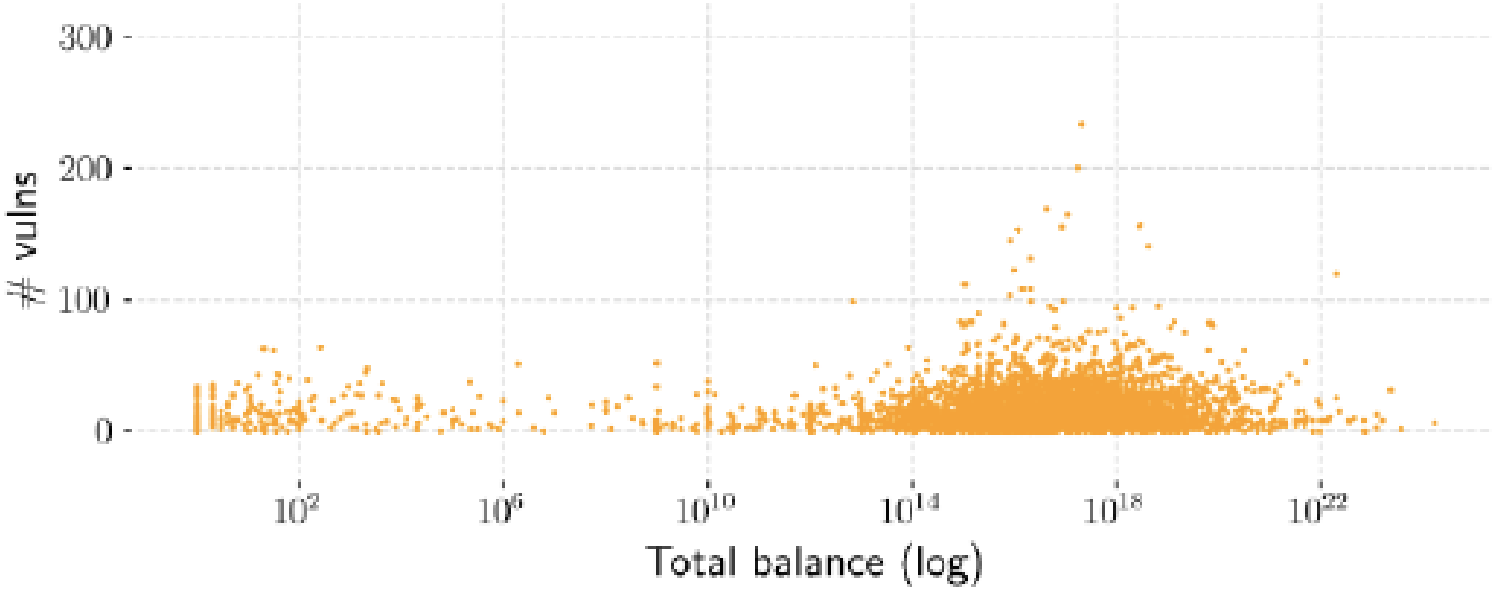
Evaluation on $_{SB}^{WILD}$

- Arithmetic the category with the highest consensus between four and more tools: 937 contracts are flagged as having an Arithmetic vulnerability
- It is followed by the Reentrancy category with 133 contracts receiving a consensus of four tools or more.
- Unchecked Low Level Calls is flagged by four tools or more in 52 contracts,
- Denial Of Service and Other in 50 and 41, respectively.
- These results suggest that combining several of these tools may yield more accurate results, with less false positives and negatives.

Evolution of Number of Vulnerabilites



Correlation between no. of vulnerabilities & balance in wei



Execution Time

#	Tools	Average	Total
1	Honeybadger	0:01:38	23 days, 13:40:00
2	Mythril	0:01:24	46 days, 07:46:55
3	Osiris	0:00:34	18 days, 10:19:01
4	Oyente	0:00:30	16 days, 04:50:11
5	Securify	0:06:37	217 days, 22:46:26
6	Slither	0:00:05	2 days, 15:09:36
7	SmartCheck	0:00:10	5 days, 12:33:14
	Total	0:01:40	330 days and 15 hours

- Oyente, Osiris, Slither, SmartCheck are much faster tools that take between 5 and 34 seconds on average to analyse a smart contract.
- HoneyBadger, Mythril, and Securify are slower and take between 1m24s and 6m37s to execute.

Answers to the Research Questions

Answer to RQ1. In our research we listed 35 state-of-the-art tools currently available to analyse Ethereum smart contracts. Further more we listed all their URLs and identified the research paper if existent, we also reviewed 7 state-of-the-art tools.

Answer to RQ2. In Table IX we established the precision and sensitivity of six state-of-the-art tools. Slither showed to be most precise and Mythril seems to have the most sensitivity.

Answer to RQ3. In Table V are defined vulnerabilities identified per category by each tool. It seems that the categories *Other*, *Reentrancy*, *Arithmetic* and *Unchecked Low Level Calls* are the most detected. Table VII also reiterates it. With this data we can also say that they seem to be the most common type of vulnerabilities.

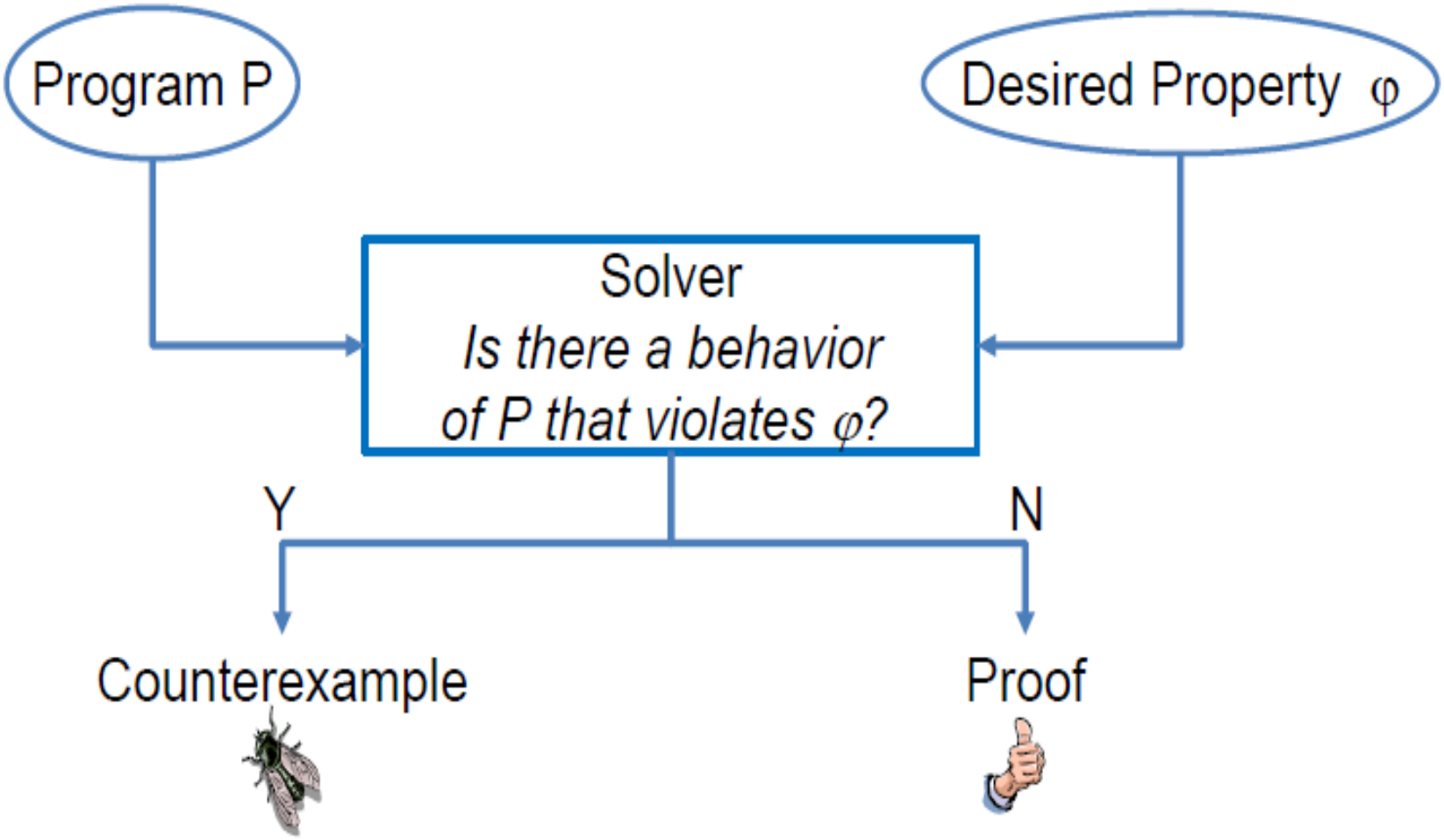
Answers to the Research Questions

Answer to RQ4. The seven tools identify vulnerabilities in 93% of the contracts (in 44,589 contracts), which suggests a high number of false positives.

Answer to RQ5. On average, the tools take 1m40s to analyze one contract. Slither is the fastest tool and takes on average only 5 seconds to analyze a contract. We have not observed a correlation between accuracy and execution time.

Answer to RQ6. By presenting an extension of SmartCheck in Sectiom VI we already contributed to improve a state-of-the-art analysis tool.

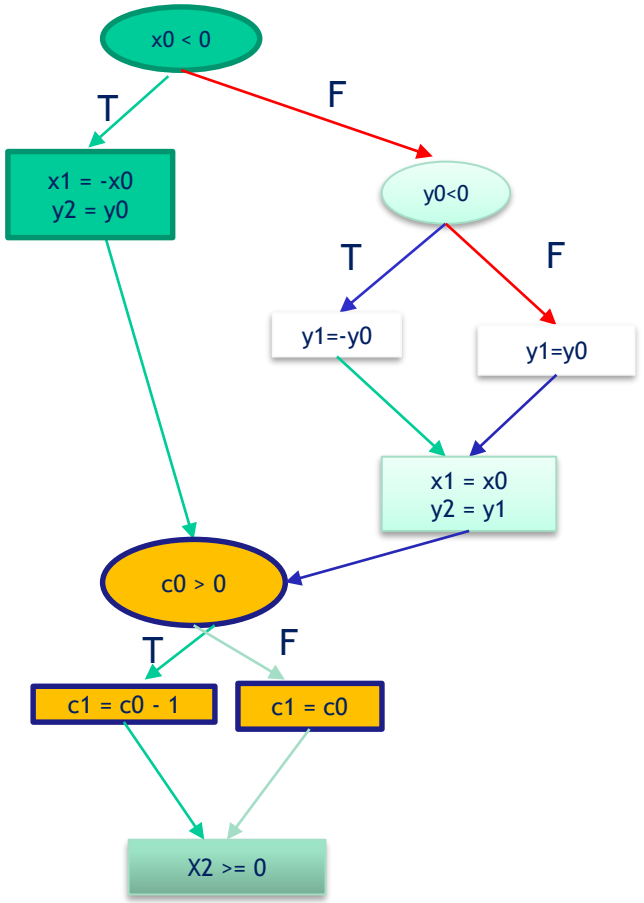
Formal Verification



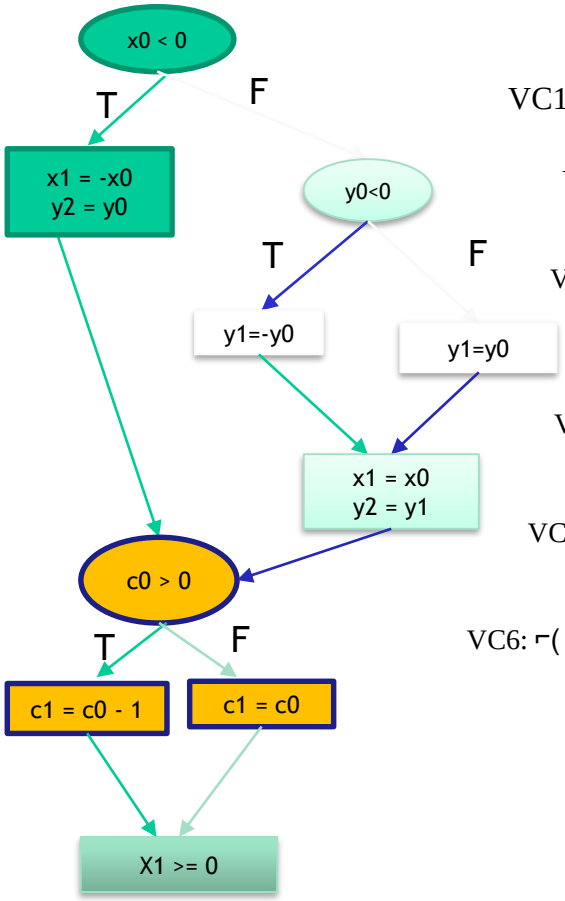
Symbolic Execution: Code and its CFG

```
if(x0 < 0){
x1 = -x0;
y2 = y0;
}
else {
if(y0 < 0){
        y1 = -y0;
        }
else  {
        y1 = y0;
                }
x1 = x0;
y2 = y1;
}

if(c0 > 0){
c1 = c0 – 1;
}
else {
c1 = c0 ;
}
assert x2 >= 0;
```



Symbolic Execution: Generation of Verification Conditions



VC1: $(x0 < 0) \Lambda (x1 = -x0) \Lambda (y2 = y0) \Lambda (c0 > 0) \Lambda (c1 = c0 - 1) \rightarrow (x1 >= 0)$

VC2: $(x0 < 0) \Lambda (x1 = -x0) \Lambda (y2 = y0) \Lambda \neg(c0 > 0) \Lambda (c1 = c0) \rightarrow (x1 >= 0)$

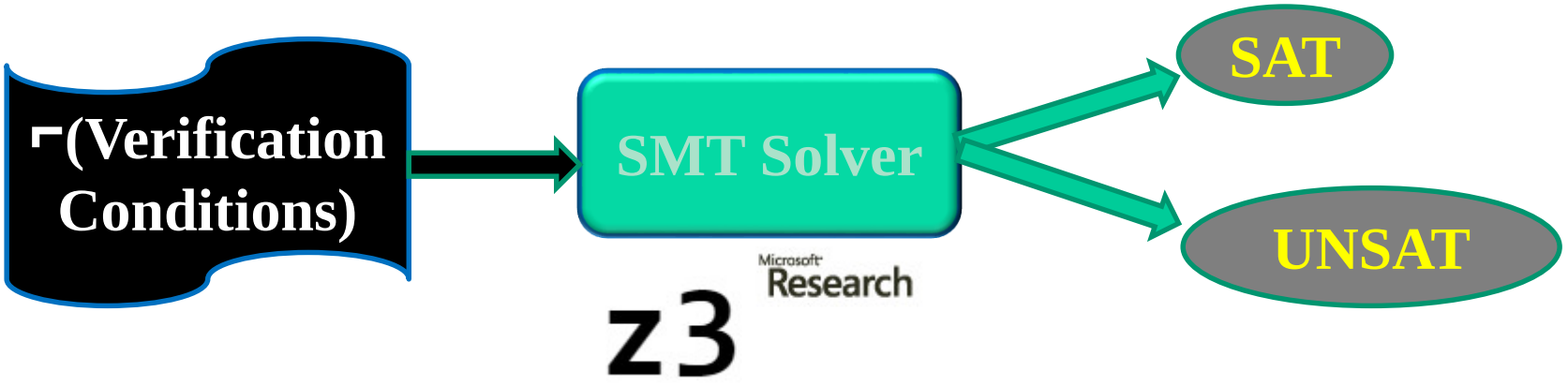
VC3: $\neg(x0 < 0) \Lambda (y0 < 0) \Lambda (y1 = -y0) \Lambda (x1 = x0) \Lambda (y2 = y1) \Lambda (c0 > 0) \Lambda (c1 = c0 - 1) \rightarrow (x1 >= 0)$

VC4: $\neg(x0 < 0) \Lambda (y0 < 0) \Lambda (y1 = -y0) \Lambda (x1 = x0) \Lambda (y2 = y1) \Lambda \neg(c0 > 0) \Lambda (c1 = c0) \rightarrow (x1 >= 0)$

VC5: $\neg(x0 < 0) \Lambda \neg(y0 < 0) \Lambda (y1 = y0) \Lambda (x1 = x0) \Lambda (y2 = y1) \Lambda (c0 > 0) \Lambda (c1 = c0 - 1) \rightarrow (x1 >= 0)$

VC6: $\neg(x0 < 0) \Lambda \neg(y0 < 0) \Lambda (y1 = y0) \Lambda (x1 = x0) \Lambda (y2 = y1) \Lambda \neg(c0 > 0) \Lambda (c1 = c0) \rightarrow (x1 >= 0)$

Symbolic Execution: Validity Checking

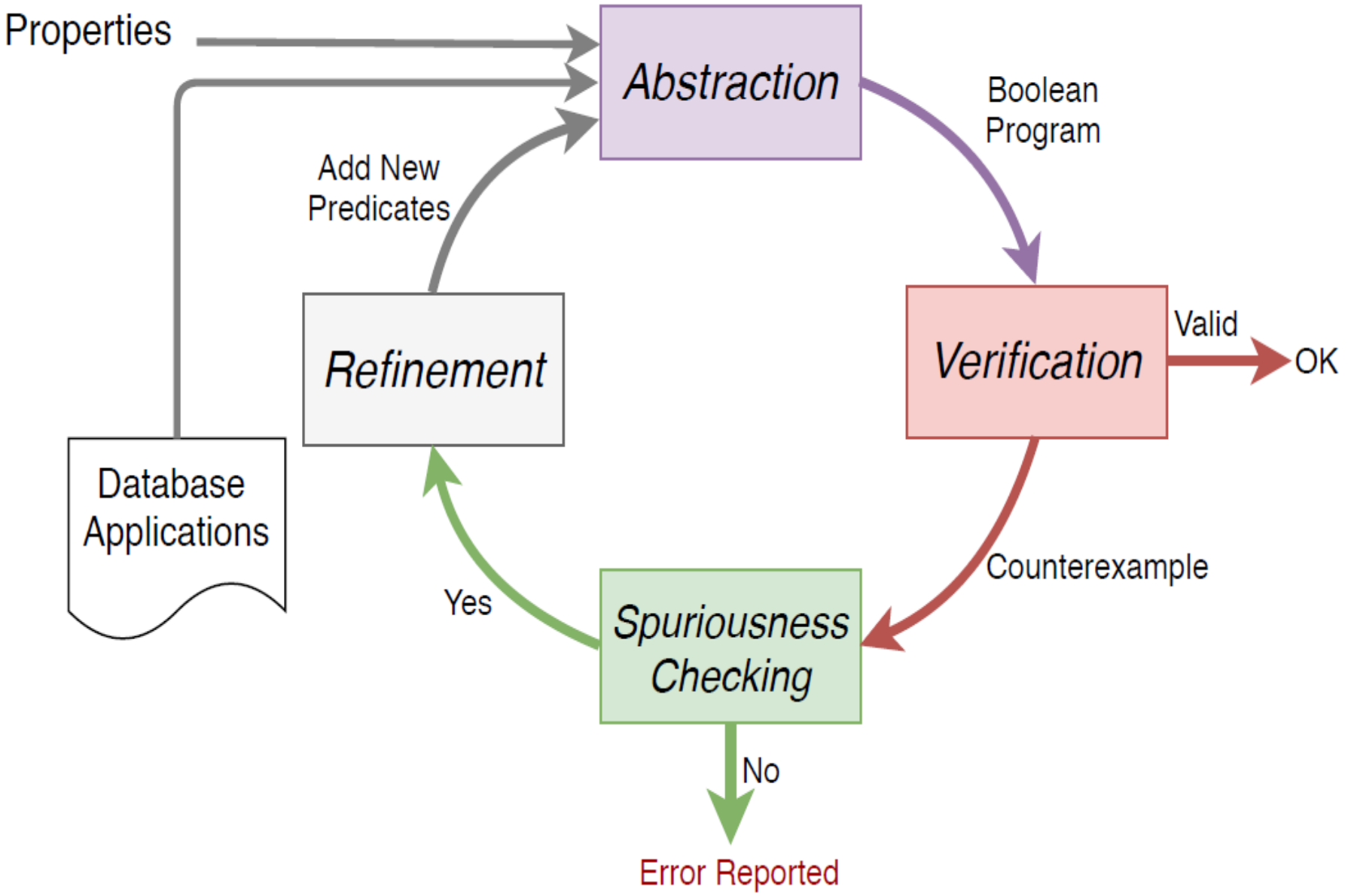


VC1: $\neg(c_0 > 0 \wedge c_1 = c_0 - 1 \wedge x_1 = c_1 \rightarrow x_1 > 0)$

VC2: $\neg(\neg(c_0 > 0) \wedge c_1 = c_0 \wedge x_1 = c_1 \rightarrow x_1 > 0)$

- SAT?
- UNSAT?

Model Checking: Predicate Abstraction

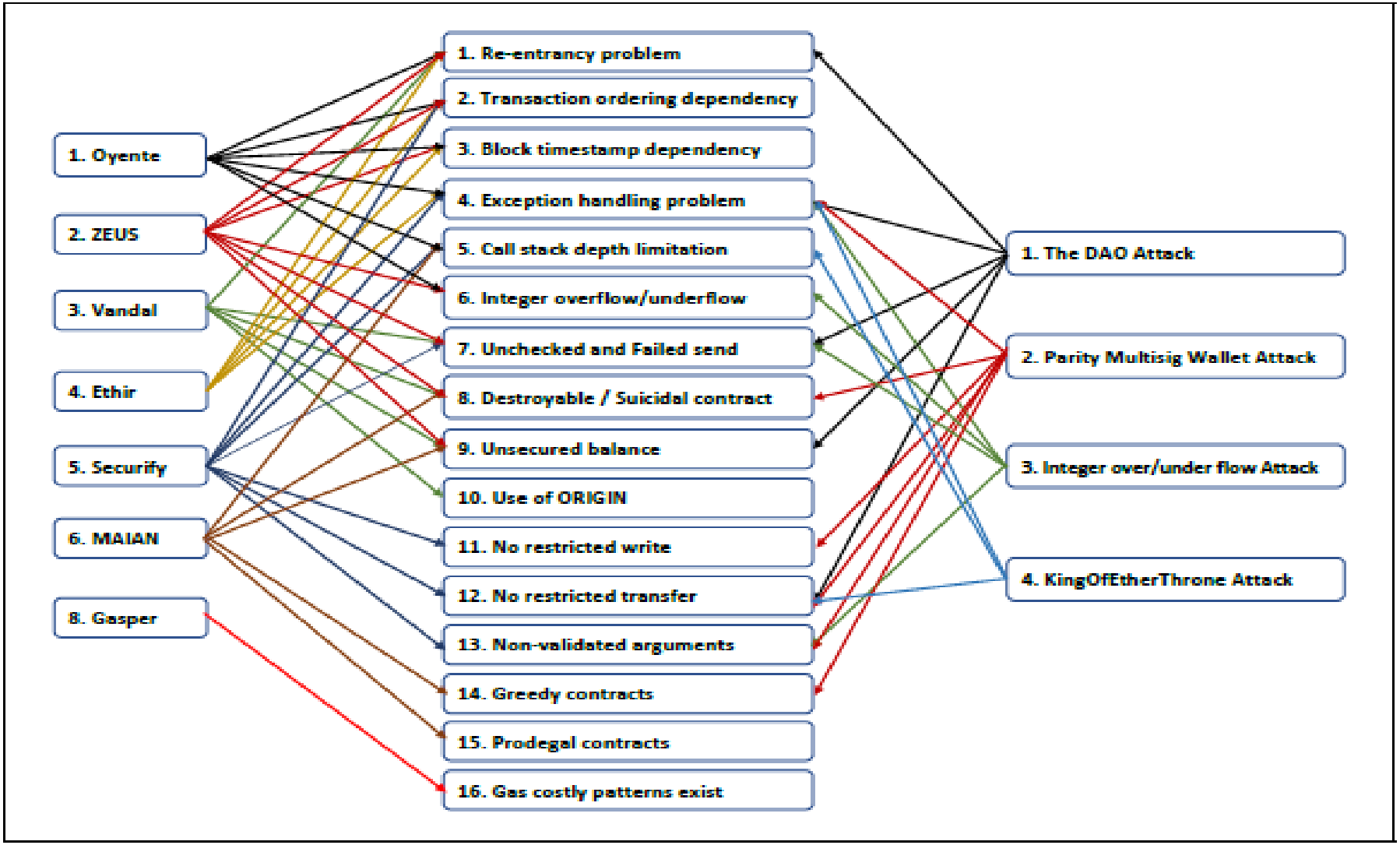


Many Others

- Smartcheck
- Mythril
- Securify
- KEVM

Mapping Between All

Security Analysis Methods on Ethereum Smart Contract Vulnerabilities - A Survey by Purathani Praitheeshan



A (partial) summary of formal verification tools.

TOLMACH et al. A Survey of Smart Contract Formal Specification and Verification, https://arxiv.org/abs/2008.02712

Verification Techniques	Tools	Selected References	Total
Model Checking	NuSMV, nuXmv	[16, 121, 143, 147, 157, 158, 206]	18
	SPIN	[32, 151, 163]	
	CPN	[74, 138]	
	BIP-SMC	[10, 145]	
	PRISM	[43]	
	UPPAAL	[20]	
	MCK	[207]	
	Maude	[28]	
	FDR	[171]	
	Ambient Calculus	[206]	
	LDLf	[180]	
Theorem Proving	Coq	[21, 25, 40, 159, 161, 185, 196, 226]	20
	Isabelle/HOL	[18, 89, 107, 108, 126, 132]	
	Agda	[53]	
Program Verification	Datalog & Soufflé	[49, 50, 95, 166, 205]	42
	Boogie Verifier & Corral	[22, 101, 215]	
	LLVM & SMACK	[118, 213]	
	SeaHorn	[14, 118]	
	F*	[42, 96]	
	KeY	[11, 37, 38]	
	Why3	[68, 156, 231]	
	$\mathbb{K}$ framework	[117, 119, 165]	
	Custom static analyses	[17, 77, 83, 126, 146, 182, 186, 187, 190]	
Symbolic & Concolic Execution	Oyente	[81, 141, 204, 233]	22
	Mythril	[153, 170, 217]	
	teEther	[123]	
	Maian	[160]	
	Manticore	[152]	
Runtime Verification & Testing	ContractLarva	[29, 76]	26
	EVM*	[142]	
	ReGuard	[134]	
	SODA	[57]	
	Solythesis	[130]	
	ECFChecker	[98]	

Thank You!