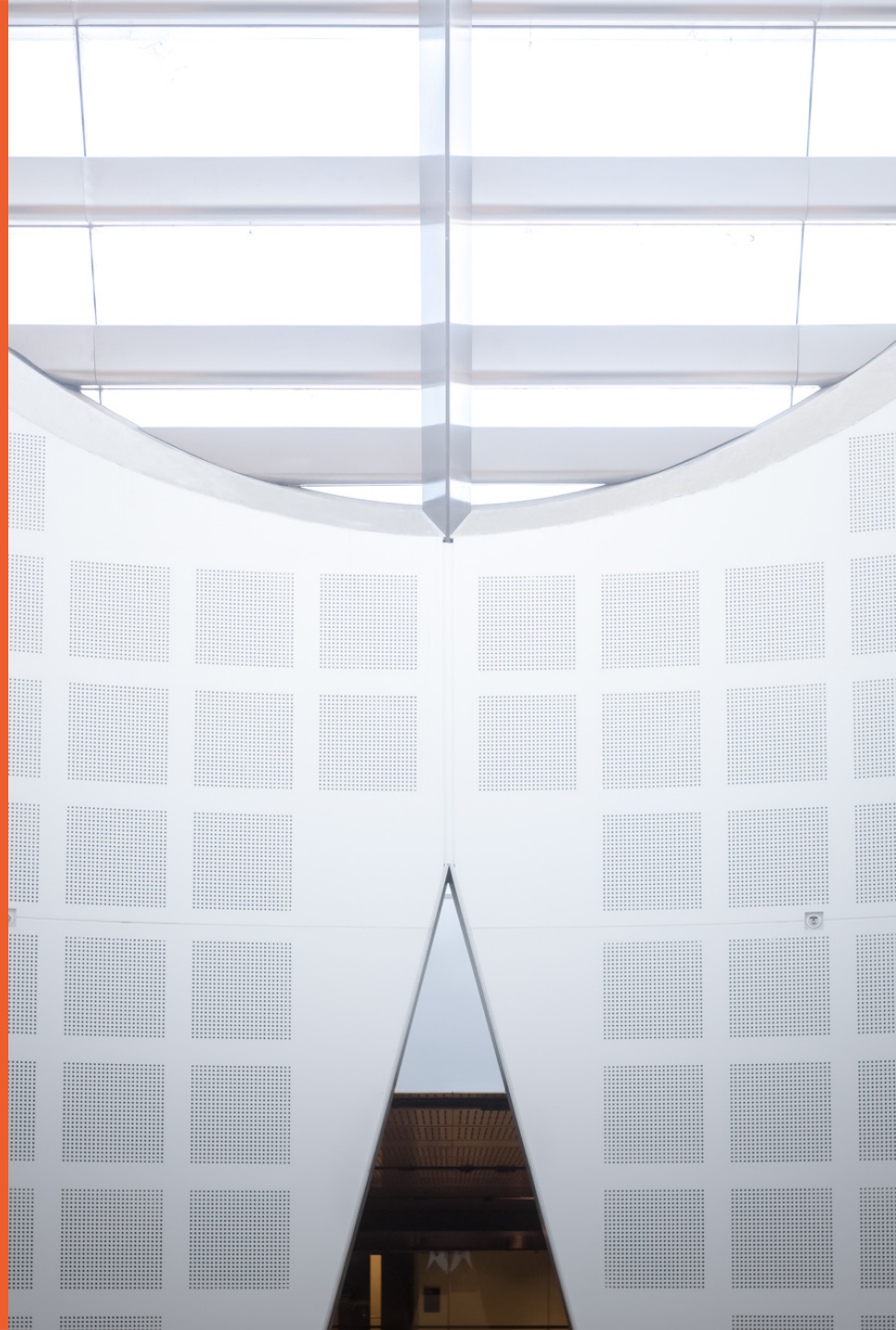
A Scalable Bug Checking Framework for Smart Contracts

Presented by

Michael Kong

School of Information Technology

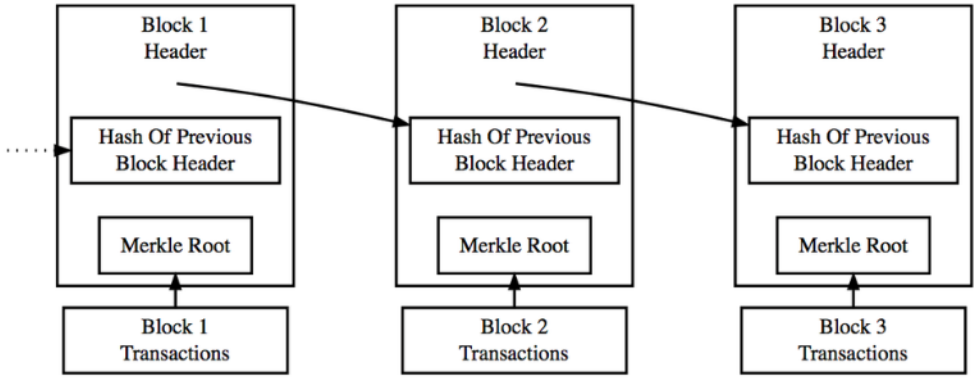
Authors: Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Bernhard Scholz



THE UNIVERSITY OF SYDNEY

What is Blockchain?

- A Data structure
  - Doubly-linked list by hashes
  - To determine hash of a block, depends on all previous hashes.
  - A blockchain => "Single source of truth" compared to other blockchains.
  - Each full node = copy of whole blockchain.
  - Not hype



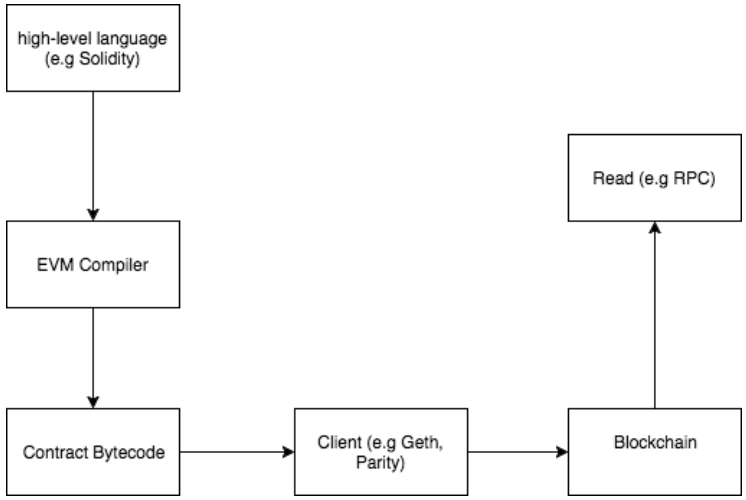
Simplified Bitcoin Block Chain

What are smart contracts?

- Just software programs compiled to bytecode
- Computation executed using virtual machine
- Run on distributed system
- Outcome of programs confirmed by consensus algorithm of the network ("Proof of Work").
- Can represent "digital assets" e.g currencies, financial instruments and real-life assets e.g land registry to validate property agreements.

Background: What is the Ethereum Virtual Machine (EVM)?

- Performs computations of smart contracts
- 1024 Stack machine
- Turing-complete
- Sandboxed and isolated from blockchain
  - No need to run node to do static program analysis



Motivation

- "TheDAO" hack: USD$60m gone.
- Parity multisig hack #1: USD$30m gone.
- Parity multisig hack #2: USD$150m frozen.
- Many More...

Motivation

Objectives

- Detect vulnerabilities and the EVM level
- Build framework
  - Decompiler to create Control-flow graph (CFG)
  - Create accurate method to statically calculating gas costs.
  - Analyze loops by calculating “Maximum Frequencies”.
  - Rapid prototyping of vulnerability detection scripts with minimal lines of logic programming code (Souffle, or others).

Vulnerabilities

1. **Unchecked send**: Not checking for failure of message calls leads to state being changed, on assumption call succeeded!
2. **Reentrancy**: Recursively calling the same function of a contract that has called other contract (e.g "TheDAO" hack).
3. **Use of tx.origin:** should never be used.
4. **Unsecured Balance:** protect the Ether & token balance of a contract from the public.
5. **Dynamically bounded loops:** Identity loops bounded by user input that can grow over time.
6. **"Wallet Griefing":** Message call fails within loop.
7. **Mass clearing of storage:** Resetting array or mapping is very expensive.

Example Vulnerability: Unchecked send

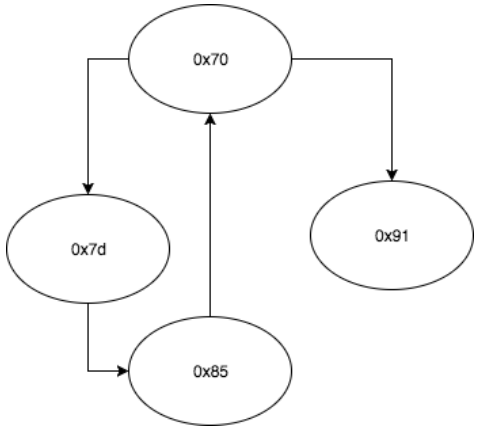
- Message calls may fail (e.g send wei – transfer my not succeed).
  - If fails, may not throw.
- Can lead to unexpected behavior (transaction succeeds, updated state, on assumption that message call suceeded.
- Solution: check for failure, and deal with it.

```
function pay() {
      creditor.send(100);
      paid = true;
}

function pay() {
       if (!creditor.send(100)) throw;
      paid = true;
}
```

Example Vulnerability: Dynamic Loop

- Loops determined by user input could grow dynamically
- Iterating through the loop can exceed block gas limit.
  - “Denial of Service" for all txs attempting to iterate iterate the loop.
  - 0x75: V29 = LT 0x0,0x1 0x100

```
pragma solidity ^0.4.11;
contract dynamicLooping{
  struct Payee {
      address addr;
      uint256 value;
  }
  Payee [] payees;
  function payOut() returns (uint) {
      uint x = 0;
      for (uint i= 0; i<payees.length; i++){
        x++;
      }
      return x;
  }
}
```



Example Vulnerability: Wallet Griefing

- When a message call to an external address is made (CALL, CALLCODE, DELEGATECALL), the transaction's program execution counter is given to that external address.
  - gives rise to potential vulnerabilities
- Occurs when a call to an external function inside a loop leads to an exception being thrown (for any reason).

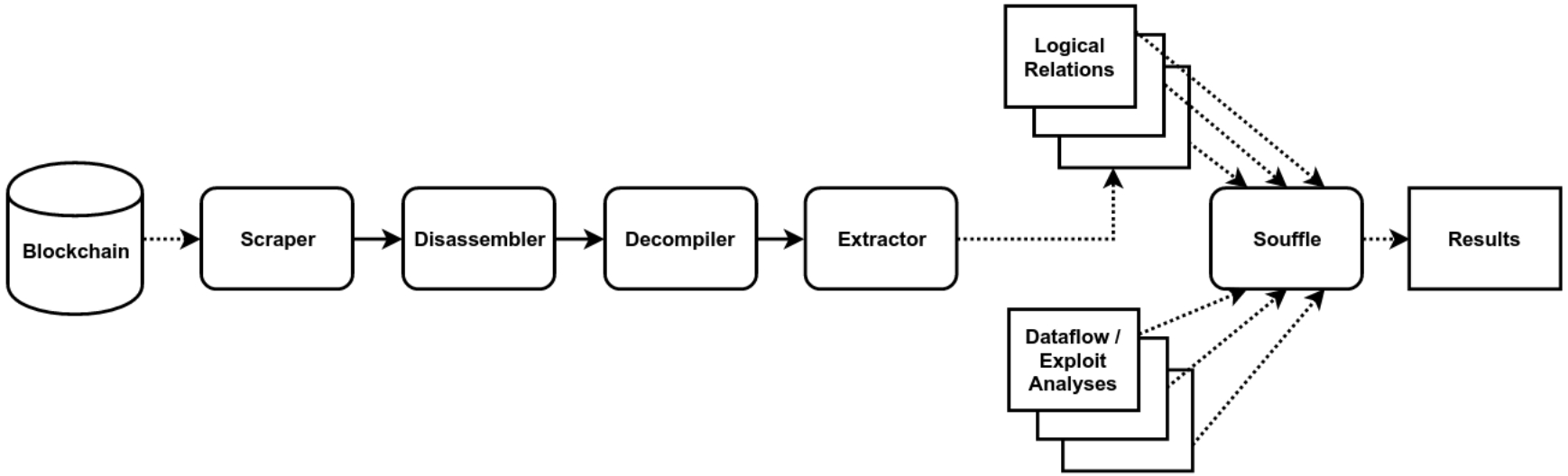
```
for (uint i=0; i<investors.length; i++) {
  if (investors[i].invested == min_investment) {
    // Refund, and check for failure.
    // This code looks benign but
    // will lock the entire contract
    // if attacked by a griefing wallet.
    if (!(investors[i].address
          .send(investors[i].dividendAmount)))
      {
        throw;
      }
    investors[i] = newInvestor;
  }
}
```

```
contract malicious{
    function (){
        throw;
    }
}
```

Example Vulnerability: Mass Clearing of Storage

- Inexperienced developers may be tricked into thinking that relatively simple statements are safe.
- Example of this is the clearing of arrays or mappings in Ethereum.
  - **addresses = new address[](0);**
- Governmental Contract: 1100 ETH stuck.
  - Lines 36/37:
    - **creditorAddresses = new address[](0);**
    - **creditorAmounts = new uint[](0);**
- SSTORE: set or clear storage value (expensive!).
- **creditorAddresses = new address[](0)**: run SSTORE in a loop, setting each element to zero iteratively.
- **creditorAddresses** becomes too big = tx exceeds block gas limit
  - **Result:** 1100 ETH stuck

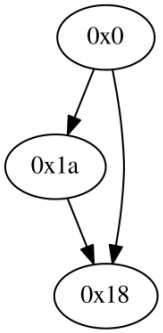
Pipeline & Code Transformation




```
60606040526000357c
0100000000000000000
0000000000000000000
0000000000000000000
000090048063193ddd
2c146037576035565b
005b60426004805050
605a565b6040518082
151581526020019150
5060405180910390f3
5b6000600560006000
5054149050606b565b
9056
```

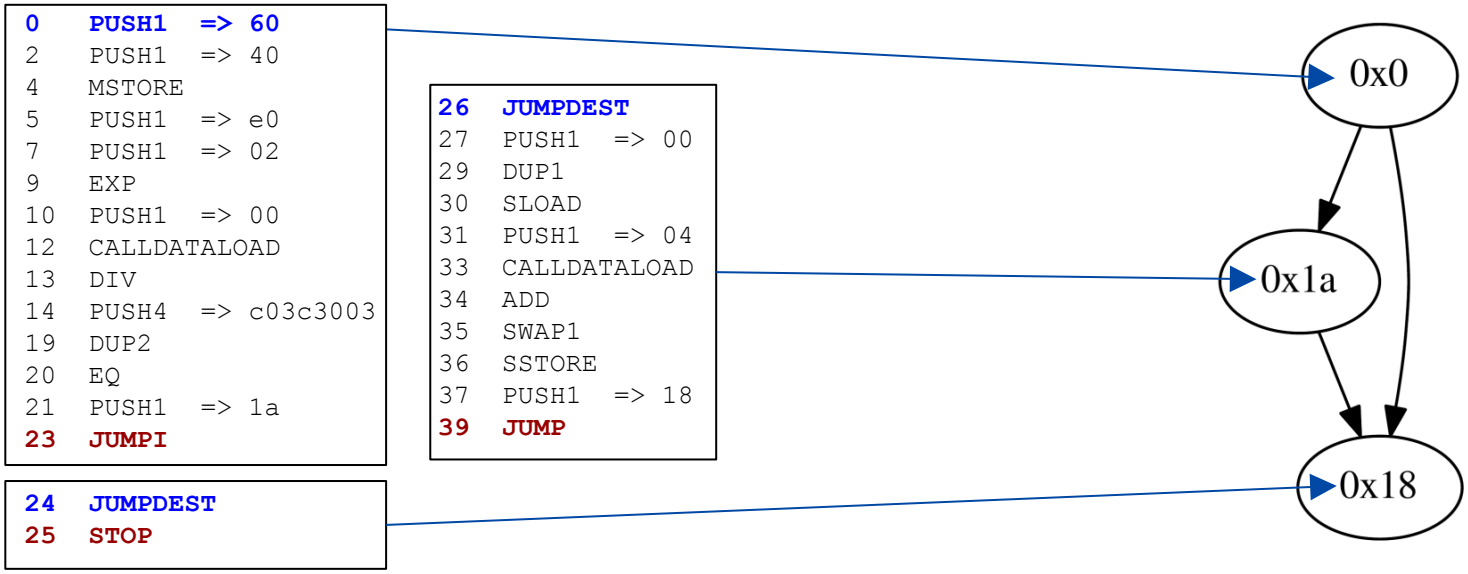
```
0x32 PUSH1   => 0x35
0x34 JUMP

0x35 JUMPDEST
0x36 STOP

0x37 JUMPDEST
0x38 PUSH1   => 0x42
0x3a PUSH1   => 0x04
0x3c DUP1
0x3d POP
0x3e POP
0x3f PUSH1   => 0x5a
0x41 JUMP
```

```
0x264: JUMPDEST
0x26d: V0 = 0x20
0x26f: V1 = 0x40
0x271: V2 = M[0x40]
0x274: V3 = SUB Var V2
0x276: V4 = 0x0
0x279: V5 = 0x61da
0x27c: V6 = GAS
0x27d: V7 = SUB V6 0x6
0x27e: V8 = CALL V7 Var
0x27f: V9 = ISZERO V8
0x280: V10 = 0x2
0x283: THROWI V9
```



Logical Relations

Bytecode to CFG



Bytecode to CFG

- A program's control flow graph is a vital tool for reasoning about its properties.
- First step of creating CFGs is splitting disassembled bytecode into *basic blocks*.
- The end of a basic block is usually an instruction which alters program flow.
- Each basic block is one node in the CFG, with directed edges indicating control flow between blocks.

Calculating Gas Costs

- Decompiler constructs Control Flow Graph (CFG).
- Each basic block represents sequence of executed statements.
  - Has predecessor and successor nodes.
- Each statement has associated gas costs (see Ethereum "Yellow Paper".
- Code in cost of each statement and calculate while building CFG.
  - Result: CFG with gas cost per block.
- Can be used to calculate "maximum frequencies":
  - Given *CFG(V, E)* and *b*
    - Number of times an edge e is visited given an execution path from start node (*s*) and end node (e).

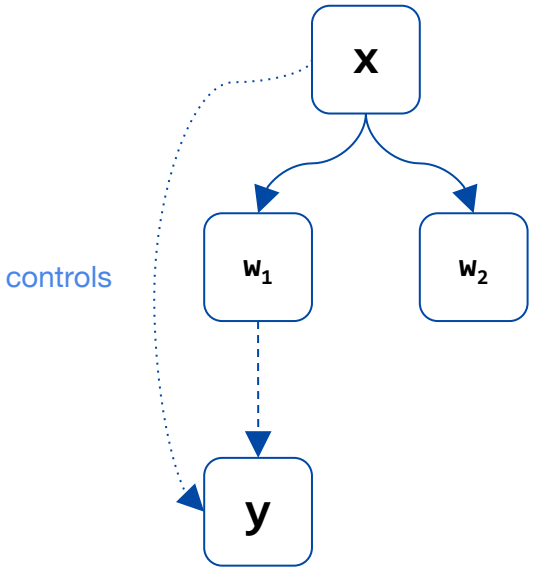
Some Useful Graph Patterns

Controlling Statements

- x controls the execution of y if x has two successors w_1 and w_2 such that w_1 is post-dominated by y and y is not reachable from w_2.

- In terms of program flow:
  The jump condition at x determines whether or not y will be executed.
  One path originating at x must execute y, the other must not.

- We will say that x controls the execution of y *with* variable c if x is a conditional jump whose condition variable is c.



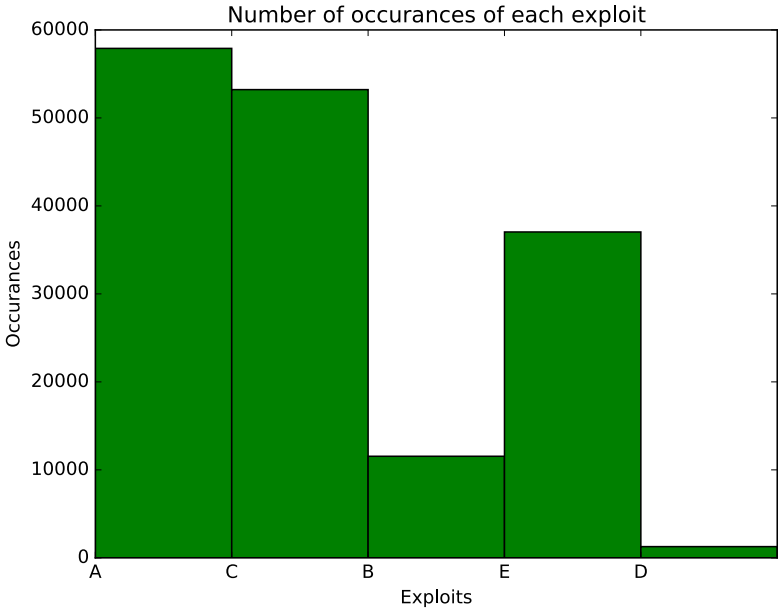
Datalog and Soufflé

- Datalog: logic programing language.
- Soufflé: C++ engine that runs datalog programs.
- Vandal produces .fact files (facts are assumed to be true) including all cyclic statements (max(e) > 1), dominance and post-dominance.

Results

a. uncheckedCall (61.8%)
b. accessibleSelfdestruct (12.34%)
c. reentrantCall (55.50%)
d. checkedCallStateUpdate (1.36%)
e. checkedCallThrows (39.57%)

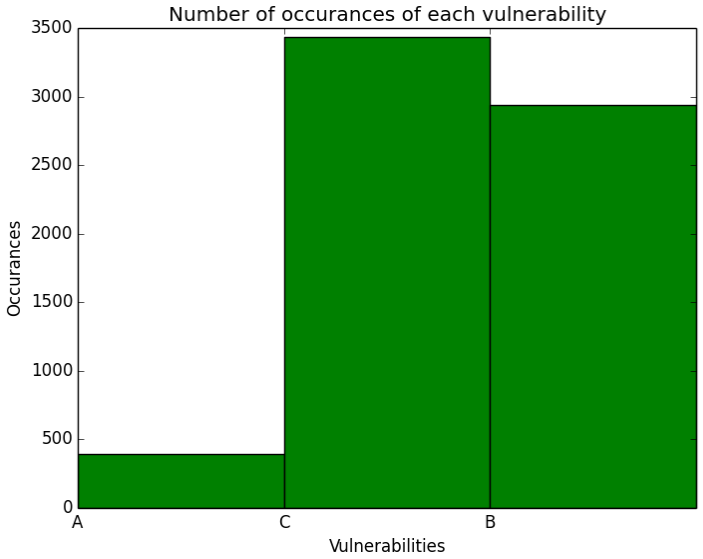
Total: 93592 Contracts (22/03/2017)



Results

A. Wallet Griefing (0.044%)
B. Mass Clearing of Storage (0.33%)
C: Dynamically Bounded Loop (0.388%)

Total: 886,323 contracts (28/07/2017)



Future work

- Incorporate Byzantium changes.
  - REVERT, RETURNDATASIZE, RETURNDATACOPY and STATICCALL op codes.
- Underflow and Overflow detection.
- Analyze Contract-to-contract communication (e.g DELEGATECALL).
- Detect gas inefficient coding patterns:
  - Chen *et al* (2017): Unreachable, opaque predicates, unnecessary looping, taking operations out of a loop, unnecessary number of loops, repeated operations.
- Vandal will be Open-Sourced soon

Related work

- Formal verification
  - Oyente: Uses symbolic execution, written in Python.
  - Bhagavan *et al* (2016): Translates EVM and Solidity into F*.
  - K framework: Translated EVM into 'K'. Checks for arithmetic underflows/overflows.
- Dynamic analysis:
  - SolCover: unit testing for Solidity code (assert() and require() checking).
  - In-build Solidity complier solver: SMT with Z3, also checks assert() and require().

Questions