Learning to Fuzz from Symbolic Execution with Application to Smart Contracts

COMS4507

Jack McPherson and Dong Bao

UQ

2020-04-29

Outline

1. Background
2. ILF System
3. Evaluation
4. Conclusion
5. Q&A

Background

Random Fuzzing vs. Symbolic Execution

- Random Fuzzing
  - Strengths
    - Fast
    - Scalable
  - Weaknesses
    - Ineffective input
    - Low code coverage
- Symbolic Execution
  - Strengths
    - Effective input
    - High code coverage
  - Weaknesses
    - Slow

```
contract Crowdsale {
  uint256 goal = 100000 * (10**18);
  uint256 phase = 0;
          // 0: Active, 1: Success, 2: Refund
  uint256 raised, end;
  address owner;
  mapping(address => uint256) investments;

  constructor() public {
    end = now + 60 days;
    owner = msg.sender;
  }

  function invest() public payable {
    require(phase == 0 && raised < goal);
    investments[msg.sender] += msg.value;
    raised += msg.value;
  }

  function setPhase(uint256 newPhase) public {
    require(
      (newPhase == 1 && raised >= goal) ||
      (newPhase == 2 && raised < goal && now > end)
    );
    phase = newPhase;
  }

  function setOwner(address newOwner) public {
    // Fix: require(msg.sender == owner);
    owner = newOwner;
  }

  function withdraw() public {
    require(phase == 1);
    owner.transfer(raised);
  }

  function refund() public {
    require(phase == 2);
    msg.sender.transfer(investments[msg.sender]);
    investments[msg.sender] = 0;
  }
}
```

1. User calls invest() with msg.value $>$ goal
2. User calls setPhase(newPhase) with newPhase $= 1$
3. Attacker (with address A) calls setOwner(A)
4. Attacker calls withdraw()

Challenges of Fuzzing Smart Contracts

- Stateful nature of smart contracts
- Limited coverage of existing fuzzers (e.g. ECHIDNA)
- Limited scalability of existing symbolic execution tools

Goal

Learn a fast and effective fuzzer from symbolic execution expert by using imitation learning, then generate sequences of transactions to reveal vulnerabilities of smart contracts.

Transactions and Blocks

- Transactions
  - Model a transaction, t, as a 3-tuple,

$$t = (f(\bar{x}), sender, amount)$$

  - $f(\bar{x})$, a public function of the target smart contract (with arguments $\bar{x}$)
  - $sender$, the address of the transaction sender
  - $amount$, the amount of Ether sent to the contract
- Blocks
  - Executing a transaction, t, against a block state, b, is denoted by

$$b \xrightarrow{t} b'$$

  - Sequence of transactions denoted

$$\bar{t} = \{t_1, t_2, \ldots, t_n\} \in \mathcal{T}^*$$

  - Block state trace is denoted by

$$b_{init} \xrightarrow{t_1} \ldots \xrightarrow{t_n} b_n$$

Markov Decision Processes (MDPs)

- A Markov Decision Process (MDP) is a mathematical framework for modelling a sequential decision-making problem probabilistically.
- A MDP is defined as a 4-tuple,

$$(\mathcal{S}, \mathcal{A}, \mathcal{E}, \mathcal{R})$$

  - $\mathcal{S}$, the set of states
  - $\mathcal{A}$, the set of actions
  - $\mathcal{E} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, the state transition function
  - $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, the reward function
- At each step, i,
  1. Observes the current state, $s_i \in \mathcal{S}$
  2. Performs an action, $a_i \in \mathcal{A}$
  3. Receives a reward, $r_i = \mathcal{R}(s_i, a_i)$
  4. Advances to the next state via the state transition function, i.e. $s_{i+1} = \mathcal{E}(s_i, a_i)$

$$\pi_{opt} = \arg\max_{\pi} \mathbb{E}_{a_i \sim \pi(s_i)} \left[\sum_{i=0}^{n} \mathcal{R}(s_i, a_i) \right]$$

Markov Decision Processes (MDPs) (cont.)

MDP Concept	Fuzzing Concept
State $s \in \mathcal{S}$	Transaction history $\bar{t} \in \mathcal{T}^*$
Action $a \in \mathcal{A}$	Transaction $t \in \mathcal{T}$
Transition $\mathcal{E}$	Concatenation $\bar{t} \cdot t$
Reward $\mathcal{R}$	Code coverage improvement
Policy π	Policy for generating t
Agent	Fuzzer with policy π

Imitation Learning

- Goal is to imitate behaviour of given expert, π^*, which achieves high reward for given task
- π^* usually has high time complexity
- Imitation learning uses expert to provide demonstrations which are then used to train an apprentice, π, which has lower time complexity

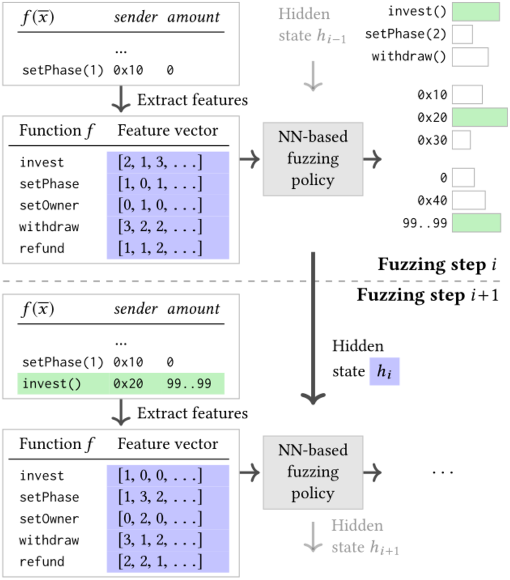
Imitation Learning (cont.)

- Run expert, π^*, on training set to construct dataset, $\mathcal{D}$

$$\mathcal{D} = \{[(s_i, a_i)]_d\}_{d=1}^{|\mathcal{D}|}$$

- Consists of samples, $[(s_i, a_i)]_d \in (\mathcal{S} \times \mathcal{A})^*$
- Aim to learn a classifier, $\mathcal{C}$, on training dataset, $\mathcal{D}$
  - $\mathcal{C}$ will output a probability vector over the set of possible actions, $\mathcal{A}$
  - Aim to assign the highest probability to the actions taken by expert, π^*

Fuzzing Process



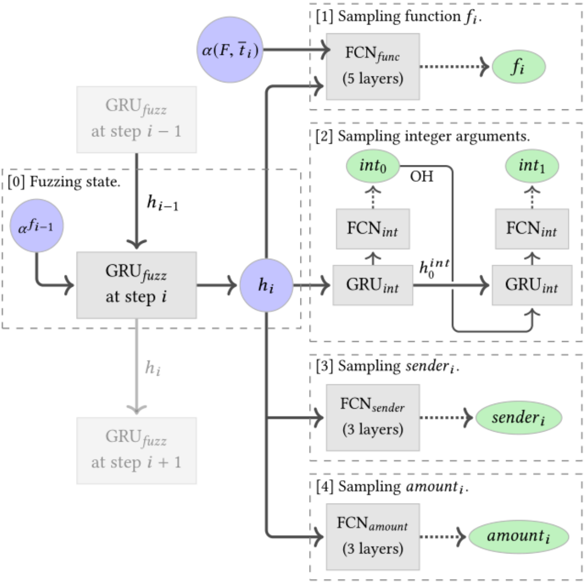
Fuzzing Policy

The fuzzing policy is denoted by the function

$$\pi : \mathcal{T}^* \times \mathcal{T} \rightarrow [0, 1]$$

1. $\pi_{func} : \mathcal{T}^* \times F \rightarrow [0, 1]$
   Selects a function, $f \in F = \left\{ f_1, f_2, \ldots, f_{|F|} \right\}$, from the set of public functions of the target contract
2. $\pi_{args} : \mathcal{T}^* \times F \times \mathcal{X}^* \rightarrow [0, 1]$
   Selects the arguments, $\bar{x}$ to f
3. $\pi_{sender} : \mathcal{T}^* \times SND \rightarrow [0, 1]$
   Selects a sender address from a predefined set of Ethereum addresses
4. $\pi_{amount} : \mathcal{T}^* \times F \times AMT \rightarrow [0, 1]$
   Selects an amount of Ether to send to the smart contract from a predefined set of Ether quantities

Fuzzing Policy (cont.)



Fuzzing Policy (cont.)

1. Sampling function, f_i

$$f_i \sim \pi^{nn}_{func}(\bar{t}_i) = FCN_{func}(h_i, \alpha(F, \bar{t}_i))$$

2. Sampling function arguments, $\bar{x}$

$$h^{int}_j = GRU_{int} = (OH(int_{j-1}), h^{int}_{j-1})$$

$$int_j \sim FCN_{int}(h^{int}_j)$$

3. Sampling sender, *sender*

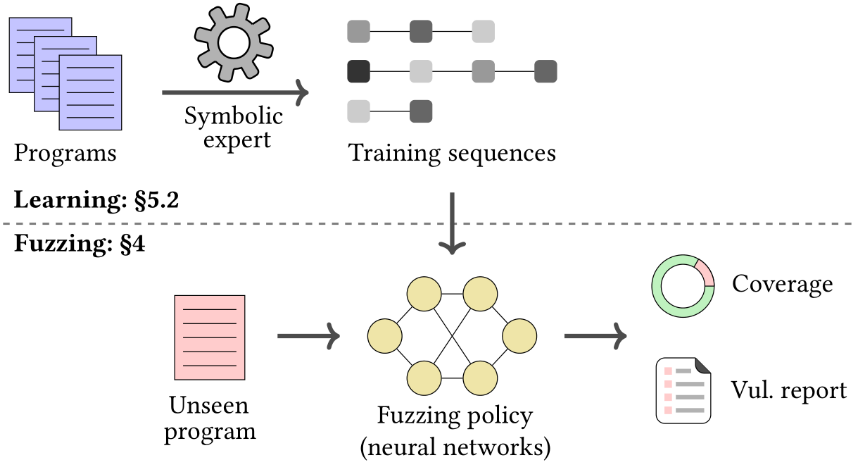
$$sender_i \sim \pi^{nn}_{sender}(\bar{t}_i) = FCN_{sender}(h_i)$$

4. Sampling amount, *amount*

$$amount_i \sim \pi^{nn}_{amount}(\bar{t}_i, f_i) = \begin{cases} FCN_{amount}(h_i) & f \text{ payable} \\ \{0 \rightarrow 1\} & \text{otherwise} \end{cases}$$

ILF System

Structure



Programs
Symbolic expert
Training sequences
Learning: §5.2
Fuzzing: §4
Unseen program
Fuzzing policy (neural networks)
Coverage
Vul. report

Structure (cont.)

- VerX [1] is used as symbolic execution expert
  - Symbolic execution engine
  - Designed for Ethereum smart contracts
- Apprentice (ILF)[2] learns from expert (VerX)
- Once trained, ILF accepts (either seen or unseen) smart contracts as input
- Produces (quasi-)optimal transaction sequences
  - Average length of 30
  - Reference implementation resets after 50 (for practicality)
- Transaction sequences are submitted against target contract
- ILF ultimately produces two key deliverables
  - Code coverage metrics
  - Vulnerability report

Training

1. Extract features from bytecode of input smart contract
2. Infer optimal transaction components from these features
    - Function
    - Arguments
    - Sender
    - Amount
3. Compute cross-entropy loss
4. Back-propogate
5. Repeat from Step 2 for arbitrarily many steps
    - Adjust hidden state weights each time

Features

- Revert
  - Boolean flag
    True if previous call to function reverted
  - Float
    Proportion of transactions that have reverted thus far
- Assert
  - Boolean flag
    True if previous call to function asserts
  - Float
    Proportion of transactions that have raised an assertion thus far
- Return
  - Boolean flag
    True if previous call to function returned
  - Float
    Proportion of transactions that have returned thus far
- Transaction
  - Float
    Proportion of transactions that have called function thus far

Features (cont.)

- Coverage
  - Integer
    Instruction coverage of contract
  - Integer
    Instruction coverage of function
- Arguments
  - Integer
    Number of arguments function accepts
  - Integer
    Number of arguments to function that are addresses
- Opcodes
  - List
    List of 50 most representative opcodes in function's code
    - Ignores arithmetic operations
    - Ignores stack operations
- Name
  - Word embedding of the function's name
    - Via word2vec

Issues with Symbolic Execution

- Ideal world
  - All smart contracts would be executed entirely symbolically
  - Symbolic execution engine (VerX, etc.) would choose sequence of transactions that are maximally covering
- Not practical
  - Symbolic execution is *extremely* computationally expensive
    - ★ Exponential time complexity
    - ★ Linear space complexity

Issues with Symbolic Execution (cont.)

Take initial block state, b_{init}. Execute contract symbolically by applying T_1 to b_{init} to yield n resultant block states,

$$\varphi_1^1(T_1), \varphi_2^1(T_1), \ldots, \varphi_n^1(T_1)$$

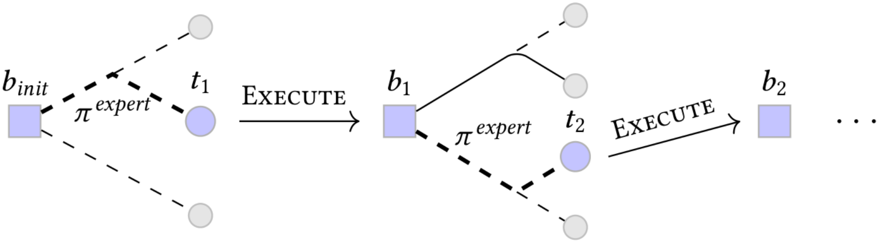
For the next transaction in the sequence, generate new resultant block states *for each current block state*

$$\varphi_1^2(T_1, T_2), \varphi_2^2(T_1, T_2), \ldots, \varphi_m^2(T_1, T_2)$$

...

- Number of constraint variables grows **linearly** in depth of execution, k
  - $\mathcal{O}(k)$
- Due to nature of constraint solvers, overall time complexity grows **exponentially** in depth of execution, k
  - $\mathcal{O}(a^k)$

Issues with Symbolic Execution (cont.)



Executing the Symbolic Expert

- How is the symbolic expert, $\pi^{expert}(\bar{t})$, actually used in ILF?
- Two main procedures
  - RUNEXPERT(c)
    - Accepts target contract, c, as input
    - Maintains priority queue, Q, of all observed block states
    - Code coverage improvement as priority
    - Runs DFSFUZZ(b, Q, c) on each element of Q
  - DFSFUZZ(b, Q, c)
    - Accepts a block state, priority queue (defined previously), and the target contract as inputs
    - Constructs transaction sequence via depth-first search
    - Does this such that each transaction *strictly* improves code coverage

Executing the Symbolic Expert (cont.)

Algorithm 1: Algorithm for running expert policy π^{expert}.

```
Procedure RunExpert(c)
    Input  : Contract c
    Q ← {b_init}
    while Q.size() > 0 do
        b ← Q.pop()
        DfsFuzz(b, Q, c)

Procedure DfsFuzz(b, Q, c)
    Input  : Block state b
             Priority queue Q of block states
             Contract c
    t̄ ← Txs(b)
    t ← π^expert(t̄)
    if t ≠ ⊥ then
        b′ ← Execute(t, b, c)
        DfsFuzz(b′, Q, c)
        Q.push(b)
```

Evaluation

Effectiveness of Imitation Learning

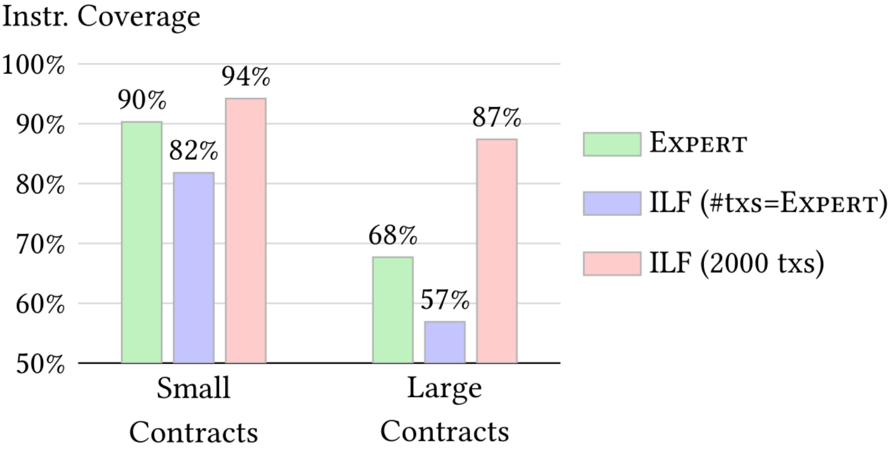
How well did ILF learn?

- We expect the apprentice policy to achieve strictly less coverage than the expert policy
  - Apprentice will never *perfectly* learn from expert
- How close can we get though?

Effectiveness of Imitation Learning (cont.)

- Small contracts (≤ 3000 opcodes)
  - Expert takes (on average) **30** transactions to achieve **90%** code coverage
  - ILF takes (on average) **30** transactions to achieve **82%** code coverage
- Large contracts (> 3000 opcodes)
  - Expert takes (on average) **49** transactions to achieve **68%** code coverage
  - ILF takes (on average) **49** transactions to achieve **57%** code coverage
- ILF times out on less contracts than the expert (for both categories)
  - Essentially sacrifices some coverage to achieve any at all

Effectiveness of Imitation Learning (cont.)



Comparison

What value does ILF add?

- Higher performance
- Higher code coverage
  - Closest competitor fuzzer is ECHIDNA
    - Achieves **at most** 70% instruction coverage
    - Even after 1,000 transactions
  - ILF achieves near 95% in the same number of transactions
- Better vulnerability detection
  - Many existing tools lack wide array of detectors
  - Some existing tools even have buggy detectors
  - **No** existing tools have the same suite of vulnerability detectors as ILF

Smart Contract Vulnerabilities

Locking Smart contract is able to be coerced into a state where Ether can be received by the contract, but never sent (essentially burning Ether)

Leaking Smart contract is able to be coerced into a state where Ether can be sent inappropriately

Suicidal Smart contract's destructor can be called by an adversary

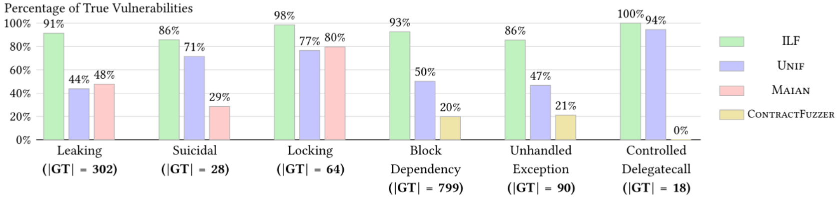
Block Dependency Ether transfers by the smart contract rely on block state variables (e.g. timestamps, etc.)

Unhandled Exception Smart contract may encounter exceptions which are not handled

Controlled Delegatecall Smart contract passes attacker-controlled parameters into a 'delegatecall' operation

Vulnerability Detection of Various Competitors

Baseline	Type	Coverage	Detectors
UNIF (π^{unif})	Fuzzer	✓	All
EXPERT (π^{expert})	Symbolic	✓	None
ECHIDNA[1] [17]	Fuzzer	✓	None
MAIAN [42]	Symbolic	✗	LO, LE, SU
CONTRACTFUZZER[2] [32]	Fuzzer	✗	BD, UE, CD



Weaknesses of ILF

Where does ILF fall short?

- Contracts that have preconditions predicated on block data
  - (Example on next slide)
- Contracts that require interaction with other smart contracts
  - Expert (and thus apprentice) cannot reason about other contracts' behaviour
  - Thus, ILF does not factor other contracts' behaviour when deciding optimal transaction at each fuzzing step

Weaknesses of ILF (cont.)

```
contract ProjectKudos {
  uint256 voteStart;
  uint256 voteEnd;
  function ProjectKudos() {
    voteStart = 1479996000; // GMT: 24-Nov-2016 14:00
    voteEnd = 1482415200; // GMT: 22-Dec-2016 14:00
  }
  function giveKudos(bytes32 projectCode, uint kudos) {
    if (now < voteStart) throw;
    if (now >= voteEnd) throw;
    ... // other operations
  }
}
```

Conclusion

Overall Security Model

- ILF is promising
  - Combination of fuzzing and symbolic execution extremely effective
  - Application of machine learning technologies to smart contract security also effective
- More thorough alternatives exist
  - Formal verification
    - ★ Verify contract implementation against formal specification
    - ★ Automated proof assistants exist to reason about properties of both
    - ★ E.g. Isabelle/HOL used on SeL4 [3]
    - ★ Very costly in terms of development time, effort, and skills
    - ★ Consider risks/rewards

Future Work

- Extending symbolic execution (and by extension fuzzing) to model *interaction* of multiple smart contracts
- Survey of smart contract security on the Ethereum mainnet
- Formal verification of smart contracts
- Integration of tools like ILF into a *de facto* (or real) standard smart contract development workflow
  - E.g. via Truffle, etc.

Bibliography

A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP*, 2020, pp. 18–20.

J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 531–548, ISBN: 9781450367479. DOI: 10.1145/3319535.3363230.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. DOI: 10.1145/1629575.1629596.

Q&A