Learning to Fuzz from Symbolic Execution with Application to Smart Contracts

Jingxuan He
ETH Zurich, Switzerland
jingxuan.he@inf.ethz.ch

Mislav Balunović
ETH Zurich, Switzerland
mislav.balunovic@inf.ethz.ch

Nodar Ambroladze
ETH Zurich, Switzerland
anodar@ethz.ch

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

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

ABSTRACT

Fuzzing and symbolic execution are two complementary techniques for discovering software vulnerabilities. Fuzzing is fast and scalable, but can be ineffective when it fails to randomly select the right inputs. Symbolic execution is thorough but slow and often does not scale to deep program paths with complex path conditions.

In this work, we propose to learn an effective and fast fuzzer from symbolic execution, by phrasing the learning task in the framework of imitation learning. During learning, a symbolic execution expert generates a large number of quality inputs improving coverage on thousands of programs. Then, a fuzzing policy, represented with a suitable architecture of neural networks, is trained on the generated dataset. The learned policy can then be used to fuzz new programs.

We instantiate our approach to the problem of fuzzing smart contracts, a domain where contracts often implement similar functionality (facilitating learning) and security is of utmost importance. We present an end-to-end system, ILF (for Imitation Learning based Fuzzer), and an extensive evaluation over >18K contracts. Our results show that ILF is effective: (i) it is fast, generating 148 transactions per second, (ii) it outperforms existing fuzzers (*e.g.*, achieving 33% more coverage), and (iii) it detects more vulnerabilities than existing fuzzing and symbolic execution tools for Ethereum.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Fuzzing; Imitation learning; Symbolic execution; Smart contracts

ACM Reference Format:
Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363230

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '19, November 11–15, 2019, London, United Kingdom

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6747-9/19/11...$15.00
https://doi.org/10.1145/3319535.3363230

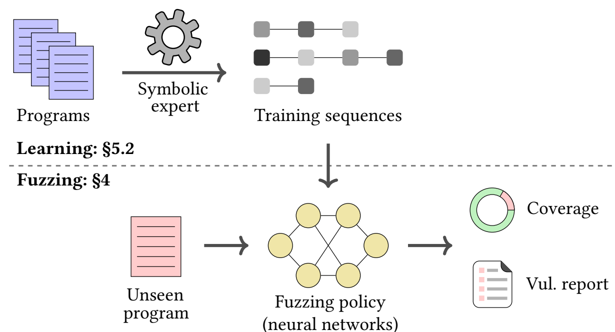

Figure 1: Learning to fuzz from symbolic execution.

1 INTRODUCTION

Fuzzing is a technique for automatically and quickly generating test inputs, and running them against a target program to uncover security vulnerabilities. Because of its simplicity and practical effectiveness, fuzzing has become one of the main approaches for software testing [21, 30, 38, 68].

A key challenge in effective fuzzing of real-world programs is to generate inputs that thoroughly exercise the code of the programs. *Random* fuzzers often get stuck with ineffective inputs that fail to exercise deep program paths and may thus miss bugs [22, 45, 48, 53]. Symbolic execution has been used to generate effective inputs as it can leverage off-the-shelf constraint solvers to solve *path constraints* which steer program execution along specific paths that improve coverage or expose new vulnerabilities [13, 14, 52, 67]. Unfortunately, symbolic execution may fail to explore deep paths and suffers from poor scalability when path constraints are complex and hard to solve [15, 48, 59].

This Work: Learning to Fuzz from a Symbolic Expert. Our core insight is that high-quality inputs generated by symbolic execution form a valuable knowledge base which can be used to learn an effective fuzzer. We propose to learn a fuzzer from inputs generated by a symbolic execution expert using the framework of imitation learning [49]. This approach has been successfully applied in various fields such as autonomous driving [47], robotics [4], and game playing (*e.g.*, where learned models achieved super-human performance in playing Go [57]). In this framework, an *apprentice* (a fuzzer in our setting) learns to imitate the behavior of a powerful, but expensive-to-execute *expert* (a symbolic execution engine). The

learned fuzzer combines strengths of both fuzzing and symbolic execution - it generates effective inputs quickly. We can afford to exhaustively run the symbolic execution expert for learning and then use the learned fuzzer to test *unseen* programs.

We illustrate this high-level idea in Figure 1. During the learning phase, we run a coverage-guided symbolic execution expert on tens of thousands of programs to produce a dataset of training input sequences, where every input improves coverage for the given programs (top part in Figure 1). To learn the behavior of the symbolic execution expert, we train a suitable architecture of neural networks that captures a probabilistic fuzzing policy for generating program inputs. Our neural networks are trained to achieve high accuracy on the obtained dataset (thus successfully imitating the symbolic execution expert). After learning, we use the learned policy to generate input sequences for fuzzing unseen programs (bottom part in Figure 1). At each step, the fuzzer samples an input from the fuzzing policy, executes it against the program, and updates its state to continue the fuzzing process. To the best of our knowledge, this is the first work to apply the framework of imitation learning in the context of program testing.

Application Domain: Fuzzing Smart Contracts. We instantiate our fuzzing approach to the domain of (blockchain) smart contracts, which are programs that run on top of distributed ledgers such as Ethereum [65]. We find this domain particularly suitable for our learning approach because smart contracts often implement similar functionality, such as tokens and decentralized crowdsales [43]. Indeed, recent work demonstrates that there is substantial code similarity among real-world contracts [34]. This makes it possible to learn from a symbolic execution expert how to fuzz new, unseen smart contracts.

Moreover, smart contracts have been proved to be challenging to fuzz effectively. Conceptually, they are stateful programs, processing sequences of transactions that modify the ledger's persistent state. A key obstacle in achieving high coverage for smart contracts is that some functions can only execute at deeper states (*e.g.*, after a crowdsale has finished). Indeed, existing fuzzers for contracts such as Echidna [17] and ContractFuzzer [32] randomly generate sequences of transactions that can fail to cover deeper paths and expose vulnerabilities. Further, existing symbolic execution engines [37, 42] can only symbolically reason limited number of transactions (*e.g.*, 3), failing to reach deeper states. Therefore, high-coverage fuzzing of smart contracts remains an important open problem, especially since contract vulnerabilities have already led to significant financial losses [44, 60].

ILF System and Evaluation. Based on our approach, we present ILF[1] (for Imitation Learning based Fuzzer), a new fuzzer for Ethereum that generates sequences of transactions achieving high coverage on real-world contracts (deployed on Ethereum). We present an extensive evaluation of ILF over 18,000 smart contracts. Our results show that the learned fuzzer is practically effective. First, ILF is fast, generating and executing on average 148 transactions per second. Second, it outperforms existing fuzzing approaches for contracts, achieving 33% more coverage than Echidna on large contracts (with ≥3K instructions). Third, it identifies more vulnerabilities than existing fuzzing and symbolic execution tools for Ethereum, *e.g.*, it discovers roughly 2× more Leaking vulnerabilities than Maian [42], a tool based on symbolic execution.

Main Contributions. To summarize, our main contributions are:

- A new fuzzing approach based on learning to imitate a symbolic execution expert.
- A new neural network architecture that models a fuzzing policy for generating input sequences for smart contracts (Section 4).
- An end-to-end implementation, called ILF, that supports semantic feature representation tailored to fuzzing of smart contracts, a practical symbolic execution expert for smart contracts, and a set of critical vulnerability detectors (Section 5).
- An extensive evaluation over >18K real-world smart contracts, demonstrating that ILF is fast, achieves high coverage (92% on average), and detects significantly more vulnerabilities than existing security tools for smart contracts (Section 6).

2 OVERVIEW

We present a simple contract which highlights the key challenges in generating transactions achieving high coverage. We then discuss the limitations of existing fuzzing and symbolic execution tools and show that they fail to cover relevant functions of the contract. Finally, we present ILF and illustrate how it works on an example.

2.1 Motivating Example

As a motivating example, we present a crowdsale, one of the most common paradigms of smart contracts on Ethereum [65]. The goal of the crowdsale is to collect 100K ether within 60 days. The crowdsale is successful if this goal is reached within 60 days, in which case the crowdsale's owner can withdraw the funds. Otherwise, the crowdsale is considered failed, and the users who have participated can refund their investments.

In Figure 2, we present the contract's Solidity code [3] adapted from the widely-used OpenZeppelin library [43]. The crowdsale's constructor initializes the crowdsale's end to 60 days from the time of deployment (returned by `now`) and assigns the crowdsale's creator (returned by `msg.sender`) as its owner.

Users can invest in the crowdsale using the function `invest`. This function executes only if the crowdsale is active (`phase = 0`) and the goal has not been reached (`raised < goal`). The crowdsale's phase (stored at variable `phase`) can be changed by calling function `setPhase`. This function ensures that the phase is changed to 1 (indicating a successful crowdsale) only if the amount of raised ether equals or exceeds the goal, and to 2 (indicating a failed crowdsale) only if the current time (`now`) exceeds the crowdsale's end and the amount of raised funds is less than the goal. If the crowdsale is in phase 1, then the invested ether can be transferred to its owner with function `withdraw`. Alternatively, if the crowdsale transitions to phase 2, then it allows users to refund their investments (via calls to `refund`).

Vulnerability. This crowdsale contract contains a vulnerability called *Leaking*, defined in [42]. This vulnerability occurs when the contract sends ether to an *untrusted* user who has never sent ether to the contract. Here, the set of *trusted* users consists of the contract's creator and all users whose addresses have appeared as arguments in transactions sent by trusted users; all remaining users

[1] ILF is publicly available at https://github.com/eth-sri/ilf.

```
1  contract Crowdsale {
2    uint256 goal = 100000 * (10**18);
3    uint256 phase = 0;
4            // 0: Active, 1: Success, 2: Refund
5    uint256 raised, end;
6    address owner;
7    mapping(address => uint256) investments;
8
9    constructor() public {
10     end = now + 60 days;
11     owner = msg.sender;
12   }
13
14   function invest() public payable {
15     require(phase == 0 && raised < goal);
16     investments[msg.sender] += msg.value;
17     raised += msg.value;
18   }
19
20   function setPhase(uint256 newPhase) public {
21     require(
22       (newPhase == 1 && raised >= goal) ||
23       (newPhase == 2 && raised < goal && now > end)
24     );
25     phase = newPhase;
26   }
27
28   function setOwner(address newOwner) public {
29     // Fix: require(msg.sender == owner);
30     owner = newOwner;
31   }
32
33   function withdraw() public {
34     require(phase == 1);
35     owner.transfer(raised);
36   }
37
38   function refund() public {
39     require(phase == 2);
40     msg.sender.transfer(investments[msg.sender]);
41     investments[msg.sender] = 0;
42   }
43 }
```

Figure 2: Crowdsale contract containing a vulnerability that allows an arbitrary user to steal the funds invested by users.

are considered as untrusted. This definition ensures that the contract's ownership or administrative permission is not transferred to untrusted users. The vulnerability in our crowdsale example is triggered by the following sequence of transactions:

t_1: A user calls `invest()` with `msg.value` $\geq$ `goal`;
t_2: A user calls `setPhase(newPhase)` with `newPhase` $= 1$;
t_3: An attacker with address A calls `setOwner(`A`)`;
t_4: The attacker calls `withdraw()`.

This sequence exposes the vulnerability because (i) the attacker does not send ether to the contract (the attacker's transactions t_3 and t_4 have `msg.value` set to 0 by default), (ii) the attacker is untrusted as her address A is not used as arguments to transactions sent by possibly trusted users (in t_1 and t_2), yet (iii) the attacker receives ether (in t_4). The vulnerability indicates a missing pre-condition of function `setOwner`, which must restrict calls to `owner` (Line 29).

2.2 Challenges of Fuzzing Smart Contracts

Smart contracts are stateful programs. Each contract has a persistent storage, where it keeps relevant state across transactions, such as the amount of raised funds `raised` in our crowdsale example (Figure 2). Each transaction processed by the contract may change the contract's state. For example, if a user invokes function `invest` of our crowdsale contract with a positive amount of ether (i.e., with `msg.value` > 0), then the amount of raised funds increases. More formally, each transaction is identified by (i) a contract function to execute, together with any arguments required by the function, (ii) the address of the user who initiates the transaction, and (iii) the amount of ether transferred to the contract.

The stateful nature of smart contracts makes them difficult to test automatically. Smart contracts often have *deep* states which can only be reached after executing a specific sequence of transactions. For example, our crowdsale can reach a state where `phase` $= 1$ holds only after users have invested a sufficient amount of ether into the contract (via calls to function `invest`). This challenge directly affects the code coverage one can achieve when testing the contract. The contract's functions often have pre-conditions defined over the state that must be satisfied for the functions to execute. For example, function `withdraw` in the crowdsale example can be executed only if `phase = 1` holds. Therefore, to achieve high coverage when testing a smart contract, one needs to generate *sequences* of transactions that thoroughly explore the contract's state space.

Next, we illustrate that existing fuzzers and symbolic execution tools indeed struggle to achieve high code coverage and, in turn, to trigger vulnerabilities hiding deeper in the state space.

Limited Coverage of Existing Fuzzers. Existing fuzzers for smart contracts, such as Echidna [17], generate transactions to test contracts. Our experiments in Section 6 indicate that these fuzzers fail to exercise functions that can only execute at deeper states. For our crowdsale contract, Echidna achieves 57% code coverage. Inspecting closely the achieved coverage reveals that Echidna fails to cover functions `setPhase`, `withdraw`, and `refund`, all of which have pre-conditions that require reaching a specific state. In addition, function `setPhase` requires generating an argument `newPhase` $\in \{1, 2\}$.

Limited Scalability of Existing Symbolic Execution Tools. While smart contracts are often small, analyzing them thoroughly using symbolic execution is challenging. This is because to uncover bugs that are revealed only at deeper states, one needs to reason about *sequences* of transaction symbolically, resulting in block state explosion and complex constraints. Concretely, the number of symbolic block states grows exponentially in the depth k of analyzed symbolic transactions. Further, the time complexity of solving the generated constraints is exponential in k (the number of symbolic variables grows linearly with k), burdening the constraint solver. In Section 5.2, we present experimental results that demonstrate the limited scalability of existing symbolic execution tools.

To cope with this scalability issue, existing symbolic execution tools [37, 42] bound the depth (*e.g.*, typically to 2-3 transactions) and analyze a subset of the feasible paths (using heuristics). Indeed, Maian [42], a symbolic tool that supports the Leaking vulnerability, fails to reveal the vulnerability in our crowdsale example, even when increasing the analysis depth to 4 (its default depth is 3).

2.3 The ILF Fuzzer

To address the above limitations, we developed ILF, a new fuzzer that uses a *probabilistic fuzzing policy* for generating sequences of transactions. The fuzzer is learned from tens of thousands of sequences of quality transactions generated by running a scalable symbolic execution expert on real-world contracts. Our expert operates on *concrete block states* and employs symbolic execution to generate *concrete sequences of transactions*. It can generate sufficiently long transaction sequences (of length 30 on average) that reach reasonably deep states. We present more details on our expert in Section 5.2. Below, we describe how the learned fuzzing policy generates transactions.

Fuzzing Policy. We show the learned fuzzing policy in Figure 3. As an input, the policy receives semantic information derived from the contract under test and the history of executed transactions so far. It outputs a probability distribution over a set of candidate transactions which can be used to extend the sequence of transactions. The transactions generated before selecting t_i are shown in the top-left table in Figure 3. For example, transaction t_{i-1} is a call to function `setPhase(1)`, sent by the user with address `0x10` and 0 ether. Based on this sequence, ILF derives a feature vector for each function f of the contract (`invest`, `setPhase`, etc.). These features (described in Section 5.1) capture domain-specific knowledge relevant for generating a new transaction t_i that aims to improve coverage. Examples of features include current coverage of each function, opcodes found in the function body and function names. Using this feature matrix and the hidden state h_{i-1}, the fuzzing policy outputs several distributions, depicted in the top-right in Figure 3. ILF samples (i) a function, together with a list of arguments, (ii) an address for the transaction sender, and (iii) an amount of ether to be sent with the transaction. These components collectively identify the next transaction t_i to be executed. In our example, ILF selects `invest()`, sent by user `0x20`, sending a large amount of ether (represented by 99..99). To deal with large domain types, we use a set of values (*e.g.*, integers and ether amount) observed during training. Finally, after selecting t_i, ILF updates its hidden state h_i and uses it later to select a transaction in the next fuzzing step.

We remark that the fuzzing policy is non-trivial to represent (and learn) due to the enormous set of transactions ILF can generate. To address this, we present a carefully crafted neural network architecture, which addresses relevant challenges, including handling variable number of arguments and large domain types (*e.g.*, integers). The architecture is described in detail in Section 4.2.

Coverage and Vulnerability Detection. A key benefit of ILF is that its fuzzing policy is fast to compute (ILF fuzzes at rate 148 transactions per second), yet it can effectively achieve high-coverage (92% on average for our dataset). To discover vulnerabilities, ILF supports six relevant detectors, including one for the Leaking vulnerability. For our crowdsale example, ILF achieves 99% coverage and correctly identifies the Leaking vulnerability within 2 seconds. In Section 6, we present an extensive evaluation of ILF and demonstrate its benefits over existing fuzzing and symbolic execution tools in terms of both code coverage and vulnerability detection.

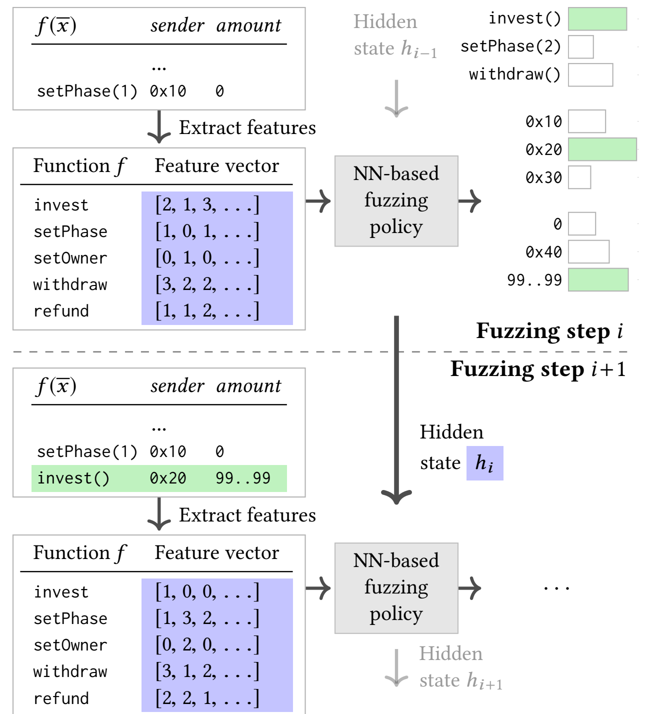

Figure 3: Fuzzing process: At each step *i*, the fuzzer derives *features* based on the transactions executed so far, and outputs distributions over the contract's functions, arguments (shown together with functions), sender addresses, and ether amounts. It samples a transaction, executes it on the contract, and updates fuzzing hidden state.

3 BACKGROUND

In this section, we provide background on Ethereum smart contracts, imitation learning, and relevant neural network models.

3.1 Ethereum Smart Contracts

Ethereum [65] is a distributed ledger that supports execution of arbitrary programs, called *smart contracts*. It supports user accounts, which are associated with private keys and provide the interface through which users submit transactions. It also supports contract accounts, which are autonomous agents, whose code implementation and state are stored on the ledger. We denote by $\mathcal{B}$ the set of possible ledger states and call a particular state $b \in \mathcal{B}$ a *block state*. A contract's execution is triggered via transactions (sent by users) or messages (calls received from other contracts). Smart contracts are compiled into Ethereum Virtual Machine (EVM) bytecode. The EVM instruction set supports standard instructions, such as opcodes for reading from and writing to the contract's state (`sstore` and `sload`), reading the timestamp of the current block `timestamp`, reading the transaction sender's address (`caller`), and others.

Transactions and Blocks. We model a transaction t as a tuple

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

where $f(\overline{x})$ identifies a public or external function f of the contract and its arguments $\overline{x}$, *sender* is the transaction sender's address, and *amount* is the ether amount transferred to the contract. A function can be *payable*, which means it can receive ether. Non-payable functions revert if amount is greater than zero. We write $\mathcal{T}$ for the set of all possible transactions. A transaction can modify the storage of the called contract and other invoked contracts. Formally, the effect of executing a transaction t at block state b is captured by $b \xrightarrow{t} b'$. Given an initial block state $b_{init} \in \mathcal{B}$ and a sequence of transactions $\overline{t} = (t_1, \dots, t_n) \in \mathcal{T}^*$, we refer to the sequence of block states $b_{init} \xrightarrow{t_1} \cdots \xrightarrow{t_n} b_n$ induced by $\overline{t}$ as a *block state trace*. For more details on Ethereum, please see [65].

3.2 Imitation Learning

We now describe the relevant background on Markov Decision Process and imitation learning.

Markov Decision Process. A Markov Decision Process (MDP) is a mathematical framework for modeling a sequential decision making problem. It involves an *agent* making a decision in an *environment* and earning a *reward*. Formally, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{E}, \mathcal{R})$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{E}: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition function and $\mathcal{R}: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. At each step i, an agent observes the current state s_i and performs an action a_i. It then receives a reward $r_i = \mathcal{R}(s_i, a_i)$ and the state is advanced according to the transition function: $s_{i+1} = \mathcal{E}(s_i, a_i)$.

The goal of solving an MDP is to learn a probabilistic policy $\pi_{opt}: \mathcal{S} \times \mathcal{A} \to [0, 1]$ which maximizes the expected cumulative reward during n steps. Whenever the agent observes a new state it can then sample an action from the learned policy. Formally, we need to solve the following optimization problem:

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

Learning From an Expert by Example. The key challenge is to design an algorithm to solve the above optimization problem. To address this challenge, we employ the framework of imitation learning [49]. This approach leverages an existing *expert* policy π^* which can achieve high reward at the given task. However, π^* often has high time complexity or requires manual effort (in our case, π^* is a symbolic execution engine). Imitation learning uses π^* to provide demonstrations which can be used to learn an *apprentice* policy $\hat{\pi}$, which is cheaper to compute with and can ideally achieve the same reward as π^*.

For learning, we first run π^* on a training set so to construct a dataset $\mathcal{D} = \left\{[(s_i, a_i)]_d\right\}_{d=1}^{|\mathcal{D}|}$, where each sample $[(s_i, a_i)]_d \in (\mathcal{S} \times \mathcal{A})^*$ is a sequence of state-action pairs meaning that π^* takes action a_i when encountering state s_i. Then, we learn a classifier C on $\mathcal{D}$, which given a state, outputs a probability vector over the possible actions and which aims to assign the highest probability to actions taken by π^*. The learned classifier C represents our apprentice policy $\hat{\pi}$. If C has high prediction accuracy on the dataset $\mathcal{D}$, then $\hat{\pi}$ will closely mimic π^*. Note that learning classifier C is a standard supervised learning problem as training input-output pairs are given in $\mathcal{D}$.

Table 1: Mapping from MDP concepts to fuzzing concepts

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

3.3 Neural Network Models

We now overview the neural network models used to represent our fuzzing policy described in Section 4.2.

Fully Connected Network. A Fully Connected Network (FCN) contains multiple fully connected layers which connect each input neuron to each output neuron. In order to provide non-linearity, neurons are passed through an *activation* function (*e.g.*, ReLU and Sigmoid). A *softmax* function at the end ensures the outputs of the network form a probability distribution (*i.e.*, sum up to one).

Gated Recurrent Unit. Gated Recurrent Unit (GRU) [16], a variant of Recurrent Neural Networks (RNN), is a model that deals with *variable length sequential inputs*. It is capable of tracking sequence history by maintaining internal state and memory. At every step i, GRU receives an input $x_i \in \mathbb{R}^m$, and based on the last state embedding $h_{i-1} \in \mathbb{R}^n$ computes a new state embedding $h_i \in \mathbb{R}^n$. This operation can be characterized as: $h_i = GRU(x_i, h_{i-1})$. Internally, GRU also computes an update gate $z_i \in \mathbb{R}^n$ and a reset gate $r_i \in \mathbb{R}^n$, which decide whether to keep or erase past information. For more details on GRUs, we refer the reader to [16].

4 FUZZING POLICY

In this section, we instantiate the concept of imitation learning to the problem of coverage-based fuzzing of smart contracts. We then show how we represent the learned policy for generating transactions using a tailored neural network architecture.

4.1 Formulating Fuzzing as MDP

Table 1 shows how the key components of MDPs connect to concepts in coverage-based fuzzing of smart contracts. The tested contract c and its initial block state b_{init} are fixed when fuzzing the contract. We therefore omit them from the formal definitions to avoid clutter. We represent the state as a sequence of transactions to allow the fuzzing policy to base its decision on the entire history of transactions. At step i, based on the sequence of transactions $\overline{t}_i = (t_1, \dots, t_{i-1})$, and the block state trace $b_{init} \xrightarrow{t_1} \cdots \xrightarrow{t_{i-1}} b_{i-1}$ induced when running $\overline{t}_i$ on the contract c under test, the agent selects a new transaction t_i to be executed based on a policy π. The selected transaction is executed, extending the sequence of transactions and the block state trace, and the agent receives a reward (*i.e.*, coverage increase). The goal is to obtain as much coverage as possible at the end of fuzzing (transaction limit or time limit reached). We now formally define the fuzzing policy and then instantiate different policies later in this section.

Fuzzing Policy. The fuzzing policy is captured by the function $\pi: \mathcal{T}^* \times \mathcal{T} \rightarrow [0,1]$. That is, given a current state and a candidate transaction, the policy outputs the probability of selecting that transaction. Since smart contract transactions have complex structure, we decompose the fuzzing policy into four components:

(1) $\pi_{func}: \mathcal{T}^* \times F \rightarrow [0,1]$, used to select a *function* f from $F = \{f^1, \ldots, f^n\}$, the set of (public or external) functions of the tested contract.
(2) $\pi_{args}: \mathcal{T}^* \times F \times \mathcal{X}^* \rightarrow [0,1]$, used to select function *arguments* $\overline{x}$ for the selected function f, where $\mathcal{X}$ is the set of possible function argument values.
(3) $\pi_{sender}: \mathcal{T}^* \times SND \rightarrow [0,1]$, used to select a *sender* from a pre-defined set of possible senders *SND*.
(4) $\pi_{amount}: \mathcal{T}^* \times F \times AMT \rightarrow [0,1]$, used to select payment *amount* for selected function f, where *AMT* is the set of possible ether amounts that can be sent with f.

We note the selections made by π_{args} and π_{amount} depend on the selected function f, because the number of arguments selected using π_{args} depends on f's arity, and the amount of ether selected using π_{amount} is 0 if f is a non-payable function.

Constructing a Transaction. The fuzzing policy π used to generate new transactions is defined as a sampling process based on the history of transactions $\overline{t}$:

$$(f(\overline{x}), sender, amount) \sim \pi(\overline{t})$$

where $f \sim \pi_{func}(\overline{t})$ is the sampled function, $\overline{x} \sim \pi_{args}(\overline{t}, f)$ is the sampled list of arguments for f, $sender \sim \pi_{sender}(\overline{t})$ is the sampled sender, and $amount \sim \pi_{amount}(\overline{t}, f)$ is the sampled ether amount.

Example: a Uniformly Random Policy. We provide the example policy π^{unif}, which selects transactions uniformly at random:

$$\pi^{unif}_{func}(\overline{t}) = Unif(F),$$
$$\pi^{unif}_{args}(\overline{t}, f) = Unif(Sig(f)), \quad \pi^{unif}_{sender}(\overline{t}) = Unif(SND),$$
$$\pi^{unif}_{amount}(\overline{t}, f) = \begin{cases} Unif([0,\ \mathrm{MA}]) & f \text{ is payable} \\ \{0 \rightarrow 1\} & \text{otherwise} \end{cases}$$

where $Unif(X)$ returns a uniform distribution over domain X, $Sig(f)$ returns the set of arguments that comply with f's signature, and MA is the maximum amount of ether for a transaction. We use this policy as one of the baselines in Section 6.

4.2 Neural Network-based Policy

We represent the neural network based fuzzing policy π^{nn} with a tailored architecture of neural network models, shown in Figure 4.

Fuzzing State. To memorize the information from the history of transactions during fuzzing, we use a GRU to encode the fuzzing state. As shown in box [0] of Figure 4, the hidden fuzzing state $h_i \in \mathbb{R}^n$ is computed as:

$$h_i = GRU_{fuzz}(\alpha^{f_{i-1}}(F, \overline{t}_i), h_{i-1})$$

where $h_{i-1} \in \mathbb{R}^n$ is encoding of the last fuzzing state, $\alpha^{f_{i-1}}(F, \overline{t}_i) \in \mathbb{R}^m$ is a feature representation of both the last selected function f_{i-1} and the history of transactions $\overline{t}_i$. We instantiate α in Section 5.1.

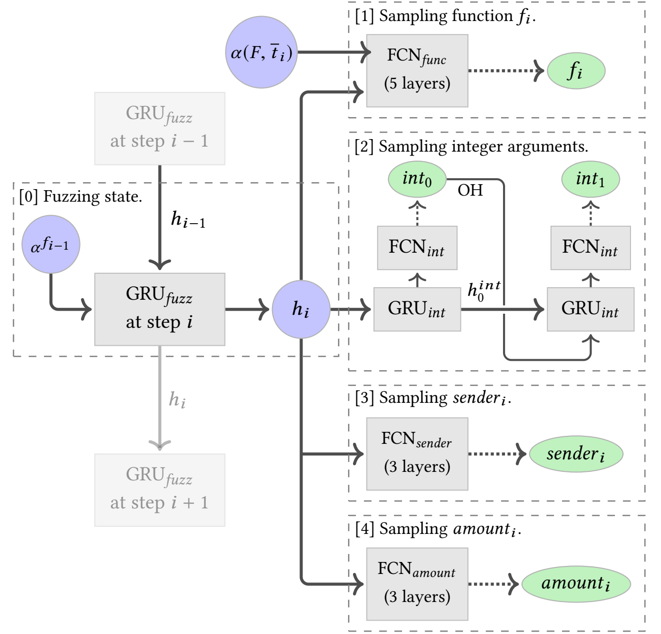

Figure 4: ILF's architecture of neural networks (NNs) for generating transactions using five steps (dashed boxes). Boxes represent NNs. Circles denote vectors and matrices provided as input to NNs. Ellipses are sampled components of the generated transaction. Solid lines (→) represent NN flows and dotted lines (⇢) denote sampling.

Generating a Transaction. To sample a new transaction at step i, we perform the following substeps:

(1) *Sampling function* f_i. We transform all functions in F (also taking into account the current trace $\overline{t}_i$), into the feature representation matrix $\alpha(F, \overline{t}_i) \in \mathbb{R}^{|F| \times m}$. Then, h_i and $\alpha(F, \overline{t}_i)$ are fed into a 5-layer FCN with softmax $FCN_{func}: \mathbb{R}^n \times \mathbb{R}^{|F| \times m} \rightarrow [0,1]^{|F|}$. This is shown in box [1] of Figure 4. Intuitively, FCN_{func} computes for each function a score representing its importance given the current state h_i. The scores are normalized using softmax and f_i is sampled from the resulting probability distribution. We summarize this process as:

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

(2) *Sampling function arguments* $\overline{x}_i$. π^{nn} can generate the most important types of arguments: integer (both signed and unsigned, any width), address, and array of them (both static and dynamic, any dimension). Other types (*e.g.*, string and bytes) appear less often in our dataset and are generated uniformly at random. Now we describe how to generate integer arguments.

There are two challenges to generating integer arguments. First, it can be prohibitive to model all possible values (Solidity has 2^{256} integers). To address this, we learn a set *SI* of effective seed integers from the symbolic expert and select over *SI*. Second, functions can have a variable number of integer arguments. To this end, we use a GRU to model their generation

process, where the number of arguments generated is based on the function f_i's arity. As shown in box [2] of Figure 4 (where we have two integer arguments), to generate the jth integer int_j, we feed a one-hot encoding of the last generated integer $OH(int_{j-1})$ and the last integer generation state $h_{j-1}^{int} \in \mathbb{R}^n$ into GRU_{int} to obtain the current state $h_j^{int} \in \mathbb{R}^n$:

$$h_j^{int} = GRU_{int}(OH(int_{j-1}), h_{j-1}^{int}).$$

Then, $FCN_{int}\colon \mathbb{R}^n \to [0,1]^{|SI|}$, a 2-layer FCN with softmax, operates on h_j^{int} to produce a distribution where int_j is sampled:

$$int_j \sim FCN_{int}(h_j^{int}).$$

The initial state h_{init}^{int} is initialized as the fuzzing state h_i and the initial input is a zero vector.

Address arguments can also be variable-length and are generated using a GRU analogously, where the set of possible addresses ranges over the set of senders and deployed contracts.

(3) *Sampling $sender_i$*. As shown in box [3] of Figure 4, $sender_i$ is sampled from $FCN_{sender}\colon \mathbb{R}^n \to [0,1]^{|SND|}$, a 3-layer FCN with softmax:

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

(4) *Sampling $amount_i$*. Finally, if the selected function f_i is a non-payable function, $amount_i = 0$. Otherwise, we need to compute a probability distribution over the set of possible ether amounts. Similar to integers, we learn a set of seed amount values SA from the expert. As shown in box [4] of Figure 4, The 3-layer FCN with softmax $FCN_{amount}\colon \mathbb{R}^n \to [0,1]^{|SA|}$ outputs a probability distribution over SA from which $amount_i$ is sampled. That is:

$$amount_i \sim \pi_{amount}^{nn}(\bar{t}_i, f_i) = \begin{cases} FCN_{amount}(h_i) & f_i \text{ is payable} \\ \{0 \to 1\} & \text{otherwise.} \end{cases}$$

After the above steps, we generate a new transaction, run it against the contract and update the hidden state.

Learning from Expert. To train π^{nn}, we assume a dataset $\mathcal{D}$ containing sequences $[(\alpha(F, \bar{t}_i), t_i)]$ where $\alpha(F, \bar{t}_i)$ is the extracted feature representation and t_i is a transaction produced by the symbolic execution expert at step i. We describe our expert and the generation of such a dataset in Section 5.2. Our goal is to train π^{nn} to produce the same sequences of transactions as in $\mathcal{D}$. To train on a single sequence in $\mathcal{D}$, we re-execute its transactions. At every step i, we feed feature $\alpha(F, \bar{t}_i)$ and hidden state h_{i-1} to π^{nn}. We compute the cross-entropy loss based on the probabilities of choosing the same function, arguments, sender and amount as in t_i. Then, we execute t_i and update the hidden state of the GRU based on t_i. After re-executing all transactions in the sequence, we perform back-propagation on the loss and jointly update weights of all networks. As standard, we train on batches of sequences and stop when all networks reach high classification accuracy.

5 THE ILF SYSTEM

We now present the ILF system which instantiates the neural-network fuzzing policy described earlier. We first introduce our feature representation for functions. Then, we present the symbolic execution expert used to generate a training dataset. Finally, we discuss the detection of critical vulnerabilities in smart contracts during fuzzing.

Table 2: Features that capture semantic information about a function f for a given history of transactions $\bar{t}$. Transaction t_{last} denotes the last call to f in $\bar{t}$.

Feature	Description
Revert	(1) Boolean, true if t_{last} ends with `revert`.
	(2) Fraction of transactions in $\bar{t}$ that end with `revert`.
Assert	(1) Boolean, true if t_{last} ends with `assert`.
	(2) Fraction of transactions in $\bar{t}$ that end with `assert`.
Return	(1) Boolean, true if t_{last} ends with `return`.
	(2) Fraction of transactions in $\bar{t}$ that end with `return`.
Transaction	Fraction of transactions in $\bar{t}$ that calls f.
Coverage	(1) Instruction and basic block coverage of the contract.
	(2) Instruction and basic block coverage of f.
Arguments	(1) Number of arguments of f.
	(2) Number of arguments of type address.
Opcodes	50 most representative opcodes in f.
Name	Word embedding of f's name.

5.1 Feature Representation of Functions

In Table 2 we present semantic features for functions which we use as an input to neural networks. Each feature is derived for a given function f and history of transactions $\bar{t}$.

The top three features capture important information about the success of the last call to f and over all calls to f in a history of transactions $\bar{t}$. For example, if the last call to f has reverted or many calls to f in $\bar{t}$ have reverted, this indicates that f's arguments are more likely to violate the pre-conditions of function f. In both cases, f should be given more importance to improve coverage. Feature *Transaction* measures the fraction of transactions that call f. Feature *Coverage* captures the current contract coverage and the coverage gain by calling f.

The last three features characterize properties of f. In order to select effective function arguments $\bar{x}$, we define feature *Arguments*. Feature *Opcodes* returns a vector with counts of 50 distinctive opcodes in f, measuring the functionality and complexity of f. To select the 50 distinctive opcodes, we excluded commonly used opcodes, such as those used for arithmetic and stack operations. The 50 opcodes are listed in Appendix A. Feature *Name* encodes the function's name. To derive this feature, we tokenize f's name into separate sub-tokens, according to *camelCase* and *pascal_case*; cf. [5], and then map each sub-token into *word2vec* word embedding [41]. We average the embeddings of sub-tokens to obtain the final embedding, resulting in a 300-dimensional vector. Feature *Name* can capture the semantic meaning of functions (*e.g.*, getters and setters).

Based on these features, we define α_s, which extracts the semantic feature matrix $\alpha_s(F, \bar{t})$ from (public and external) functions of the tested contract F and history of transactions $\bar{t}$.

Embedding via a Graph Convolutional Network. To capture dependencies between functions, we employ a Graph Convolutional Network (GCN) [35]. We construct a graph g where each function in F is represented by a node, and an edge (f, f') captures that

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

1 **Procedure** RunExpert(c)
  Input : Contract c
2   $Q \leftarrow \{b_{init}\}$
3   **while** Q.size() > 0 **do**
4     $b \leftarrow Q$.pop()
5   DfsFuzz(b, Q, c)

6 **Procedure** DfsFuzz(b, Q, c)
  Input : Block state b
      Priority queue Q of block states
      Contract c
7   $\bar{t} \leftarrow$ Txs(b)
8   $t \leftarrow \pi^{expert}(\bar{t})$
9   **if** $t \neq \perp$ **then**
10     $b' \leftarrow$ Execute(t, b, c)
11   DfsFuzz(b', Q, c)
12   Q.push(b)

the sets of reads and writes in f and f' may overlap (indicating a dependency). We use this graph to further embed the functions' semantic information $\alpha_s(F, \bar{t})$ as follows:

$$\alpha(F, \bar{t}) = GCN(\alpha_s(F, \bar{t}), g)$$

where $GCN : \mathbb{R}^{|F| \times l} \times G \rightarrow \mathbb{R}^{|F| \times m}$ transforms the semantic feature matrix into an embedding matrix. The embedding matrix $\alpha(F, \bar{t})$ is then used as input to π^{nn} as described in Section 4.2. We note that the GCN is treated as an intermediate layer between the extracted input features $\alpha_s(F, \bar{t})$ and π^{nn}, and is trained jointly with π^{nn}. For more details on how the GCN is used, see Appendix B.

5.2 Symbolic Execution Expert

Now we discuss the challenge of scaling symbolic execution on smart contracts and design of our practical symbolic expert.

Challenges of Scaling Symbolic Execution. An ideal expert would analyze multiple transactions fully symbolically and choose sequences of concrete transactions that achieve high coverage. Concretely, to fully analyze all behaviors reached by one transaction from the initial block state b_{init}, one can execute the contract symbolically (assuming symbolic transaction) to compute the symbolic states at the end of the transaction. This results in symbolic block states $\varphi_1^1(T_1), \ldots, \varphi_n^1(T_1)$, where T_1 is the symbolic transaction and n is determined by the number of paths in the contract. Then, to analyze all behaviors that can be reached within two transactions, we symbolically execute the contract from each symbolic state $\varphi_i^1(T_1)$, resulting in symbolic block states $\varphi_1^2(T_1, T_2), \ldots, \varphi_m^2(T_1, T_2)$, where T_1 and T_2 are the first and second symbolic transactions. Such scheme can continue to depth k of analyzed symbolic transactions.

Unfortunately, the above approach does not scale due to its exponential time complexity (as stated in Section 2.2). To measure its scalability in practice, we randomly selected 100 small contracts (with <3K instructions) and 100 large ones (with ≥3K instructions) from our dataset. We then measured how many of the selected contracts can be analyzed using symbolic execution (as described

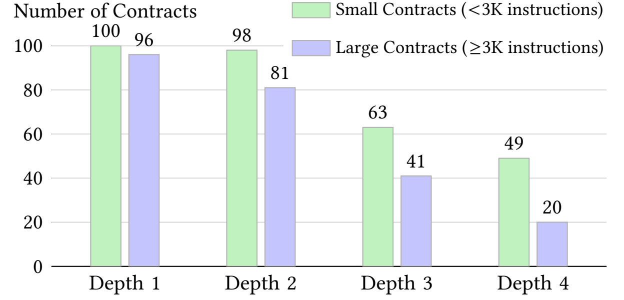

Figure 5: Number of contracts that can be fully analyzed symbolically at different transaction depths within 3 days.

above) for different depths k within 3 days. We plot the results in Figure 5. The data shows that symbolic reasoning about all states reachable within 1 transaction is feasible: all small contracts and 96 of the large ones can be analyzed at depth 1. However, only 20 of the big contracts can be fully analyzed symbolically at depth 4.

Aware of the above scalability issue, we design a practical symbolic execution expert π^{expert} that iteratively constructs a sequence of transactions by running the contract symbolically for a single transaction and selecting a concrete one to optimize coverage. We next describe our symbolic expert.

Expert Policy. Given a sequence of concrete transactions $\bar{t}$, our expert policy $\pi^{expert}(\bar{t})$ symbolically executes the contract from the current concrete block state (*i.e.*, the block state reached when running $\bar{t}$ at b_{init}), selects a feasible path that improves coverage the most (if any), runs a constraint solver to generate a concrete transaction t following that path, and returns t. If no transaction can improve coverage at the current block state, then $\pi^{expert}(\bar{t}) = \perp$. We use the symbolic execution engine of VerX [46] to symbolically execute the contract and construct the transaction t.

Running the Expert Policy. In Algorithm 1, we define the procedure for running our expert policy π^{expert} on a target contract c. The entry procedure, RunExpert, maintains a priority queue Q of observed block states, ranked by the coverage improvement that the expert policy π^{expert} can achieve by generating a new transaction for a given block state. Initially, Q contains the initial block state b_{init}. At every loop iteration (Line 3), RunExpert pops a block state b from Q (Line 4) and runs DfsFuzz (Line 5), until no observed block states can further improve coverage (*i.e.* until Q is empty).

The procedure DfsFuzz constructs a sequence of transactions in a depth-first manner, where each transaction strictly improves coverage. First, the procedure obtains the sequence of concrete transactions $\bar{t} = $ Txs(b) that results in reaching the current block state b and runs the expert policy $\pi^{expert}(\bar{t})$, which either returns a transaction t that improves coverage or $\perp$ (indicating that no transaction improves coverage). If coverage can be improved, the new transaction t is executed at the current block state b (Line 10), resulting in a new block state b'. DfsFuzz is recursively invoked with b', to generate another transaction that may further improve coverage. Every block state b encountered by DfsFuzz is pushed to Q so that it can be considered again by RunExpert for generating different sequences of transactions.

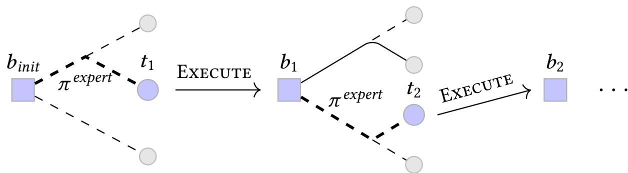

Figure 6: Example run of RunExpert in Algorithm 1. Squares represent block states. Circles and circles represent generated and unselected transactions, respectively. Solid lines represent covered paths, dashed lines uncovered paths, and arrows represent block state transitions.

In Figure 6, we show an example run of RunExpert. First, DfsFuzz is executed with the initial block state b_{init} as argument. Policy π^{expert} considers 3 paths and returns the transaction t_1 along the feasible path improving coverage the most. t_1 is executed and transitions the contract to block state b_1. At block state b_1, π^{expert} considers 4 paths that improve coverage and generates t_2, which transitions the contract to b_2. Note that $t_1, t_2, \ldots$ correspond to the sequence of transactions generated by π^{expert}. Further, DfsFuzz appends all observed block states $b_{init}, b_1, b_2, \ldots$, which are all considered by RunExpert for generating different sequences.

Generating Transaction Sequences for Learning. For each contract c_d in the training set, we use RunExpert (Algorithm 1) to generate transactions while keeping track of features α_s. We select the longest sequence $[(\alpha_s(F, \bar{t}_i), t_i)]_d$ where $\alpha_s(F, \bar{t}_i)$ is a feature matrix and t_i is produced by running π^{expert} on b_{i-1} (the block state induced by $\bar{t}_i$). This is usually the first sequence produced by DfsFuzz. Then a generated dataset $\mathcal{D} = \left\{[(\alpha_s(F, \bar{t}_i), t_i)]_d\right\}_{d=1}^{|\mathcal{D}|}$ can be used for learning π^{nn} as described in Section 4.2.

Learning Seed Integers and Amounts. As mentioned in Section 4.2, we sample integer arguments from a set of seeds *SI*. We construct this set by selecting *top-K* most frequently used integers in $\mathcal{D}$ produced by π^{expert} solving SMT constraints. The set of seed amounts *SA* is constructed similarly. We provide examples of *SI* and *SA* and their appearance counts in Appendix D.

5.3 Vulnerability Detection

We have adapted from [32, 42] six detectors that identify critical vulnerabilities of smart contracts. They are presented in Table 3 with a brief description and elaborated further in Appendix C. We note that more detectors can be easily added. We have implemented a fast dataflow analysis based on backward slicing over concrete execution traces, which was required by some detectors.

6 EXPERIMENTAL EVALUATION

In this section we present an extensive experimental evaluation of ILF, addressing the following questions:

- **Coverage:** Does ILF achieve higher coverage than other fuzzers?
- **Vulnerability Detection:** Does ILF discover more vulnerabilities than other security tools for Ethereum?
- **Components:** Are ILF's components relevant for its performance in terms of coverage and vulnerability detection?

Apart from answering these questions, we report three case studies.

Table 3: ILF's vulnerability detectors. An attacker for Leaking and Suicidal is an untrusted user who exploits the vulnerability but never sends ether to the contract. See Appendix C for detailed definitions of the detectors.

Vulnerability	Description
Locking [42]	The contract cannot send out but can receive ether.
Leaking [42]	An attacker can steal ether from the contract.
Suicidal [42]	An attacker can deconstruct the contract.
Block [32] Dependency	The contract's ether transfer depends on block state variables (*e.g.*, `timestamp`).
Unhandled Exception [32]	Root call of the contract does not catch exceptions from child calls.
Controlled Delegatecall [32]	Transaction parameters explicitly flow into arguments of a `delegatecall` instruction.

6.1 Implementation

Our transaction execution backend is implemented on top of Go Ethereum [20]. Instead of performing RPC calls, ILF sends transactions natively to achieve fast execution. Neural Networks (NNs) in π^{nn} are implemented using Pytorch [2]. Sizes of all hidden layers in NNs of π^{nn} are set to 100 and all FCNs use ReLU as activation functions. After obtaining a training dataset from our expert, we train ILF as described in Section 4.2 for 20 epochs, which takes around 2h on a single GPU. ILF uses 5 senders for fuzzing (3 trusted users, and 2 attackers) which proved to be sufficient in our experiments (more senders do not improve fuzzing). We set the sizes of the seed integer set *SI* and seed amount set *SA* both to 50 (examples are shown in Appendix D). A heuristic reverting the blockchain to the initial block state b_{init} after every 50 transactions is added when running π^{nn} and π^{unif} so to avoid getting stuck in a locked state.

6.2 Evaluation Setup

We first describe our experimental setup.

Baselines. In our experiments, we compare ILF to the baselines and existing tools listed in Table 4. For existing tools, we select those that report coverage or support more than one of ILF's detectors. Unif and Expert use fuzzing policies π^{unif} and π^{expert}, respectively. Echidna [17] reports coverage but does not implement any detectors. ContractFuzzer [32] and Maian [42] each support three of ILF's detectors but do not report coverage.

Dataset. We obtain our dataset by crawling Etherscan verified contracts [19], real-world contracts deployed on Ethereum mainnet. We removed 7, 799 duplicates by computing the hash of the contracts' source and bytecode. We filtered contracts which failed to deploy or contain hardcoded addresses to contracts not present in our dataset. Our final dataset contains 18, 496 contracts. To examine the coverage of contracts with different sizes, we split the dataset into 5, 013 large contracts (≥3K instructions) and 13, 483 small ones (<3K instructions). Table 5 presents statistics of our dataset. Note that we add the contracts (if any) that can interact with the target contract to the local test blockchain. Those contracts are identified by constructor arguments of the target contract.

Table 4: Baselines and detectors in common with ILF.

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

[1] Latest commit that reports coverage: d93c226b2ad4ff884f33faa850d7d36556c12211.
[2] We ignore the LO detector of CONTRACTFUZZER as we found a major bug there.

Table 5: Statistics on our dataset, showing the average numbers of: lines of source code, bytecode instructions, control flow blocks, functions, and payable functions.

Dataset	Source	Instr.	Block	Func.	Payable
Overall	227	2719	181	17	0.6
Small	165	2026	131	13	0.4
Large	392	4585	317	27	1.0

Experiments. In our experiments, we perform a standard 5-fold cross validation: we randomly split the entire dataset into five equal folds and each time we pick one fold as the testing set and use the remaining four as the training set. We repeat this training and testing procedure five times on the five different splits and report testing results on the entire dataset.

We ran EXPERT (with 3h limit for each contract) and CONTRACTFUZZER on a server with 512 GB memory and two 2.2 GHz AMD EPYC 7601 CPUs (64 cores and 128 threads in total). EXPERT spent 76h and CONTRACTFUZZER spent 62h to run on the entire dataset (with massive parallelization). All other experiments were done on a desktop with 64 GB memory, one 4.2 GHz Intel Core i7-7700K CPU (8 cores and 8 threads) and two Geforce GTX 1080 GPUs.

6.3 Code Coverage

We now present evaluation results on code coverage. We measure instruction and basic block coverage. In this section, we report instruction coverage. The results on basic block coverage are similar and can be found in Appendix E. In the following, we first compare ILF with fuzzers (UNIF and ECHIDNA), and then EXPERT.

Comparing ILF to Fuzzers. We present coverage of ILF, UNIF, and ECHIDNA on small contracts in Figure 7 and on big contracts in Figure 8. We plot it in the number of transactions (up to 1K). ILF consistently outperforms UNIF and ECHIDNA. On small contracts, ILF achieves 94% coverage, 16% higher than UNIF and 24% higher than ECHIDNA. On large contracts, ILF achieves 87%, 16% and 33% higher than UNIF and ECHIDNA, respectively. ECHIDNA did not perform well because it models EVM incompletely (*e.g.*, we found it uses one sender and does not model `msg.value`) and exited with error on 811 contracts (reporting 0% coverage). Note that the coverage on the 4.3% contracts cannot significantly impact the results. Overall, ILF, UNIF and ECHIDNA are fast - to fuzz 1K transactions, they spent 7s, 4s, and 6s, respectively, on average per contract.

Comparing ILF to the Symbolic Expert. In Figure 9, we compare ILF with EXPERT (which ILF learns to imitate). On average, EXPERT spent 30 transactions to achieve 90% coverage on small

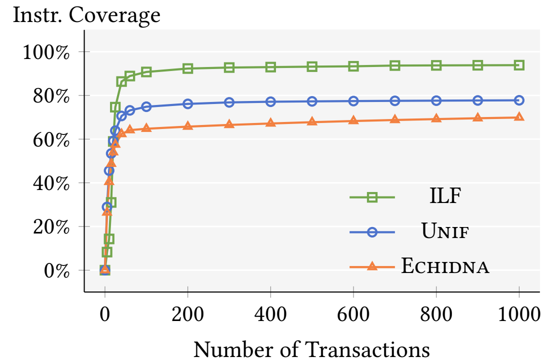

Figure 7: Instruction coverage on small contracts.

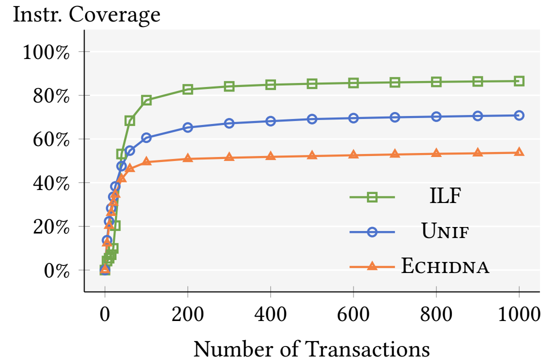

Figure 8: Instruction coverage on large contracts.

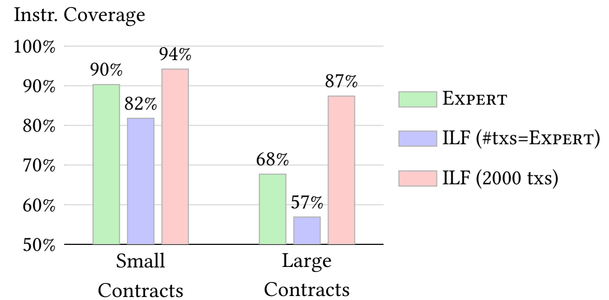

Figure 9: Instruction coverage of EXPERT and ILF.

contracts. On large contracts, it spent 49 transactions to get 68%. Compared to EXPERT, when ILF ran the same number of transactions, it achieves 8% lower coverage on small contracts and 11% lower coverage on large contracts. This is expected, as EXPERT is powerful (*i.e.*, it improves coverage at every transaction).

However, EXPERT is slow — it took on average 547s (on small contracts) and 2,580s (on big contracts) for EXPERT to achieve the coverage levels in Figure 9. We also ran ILF for 2K transactions (only spending 13s on small contracts and 17s on big contracts). In this case, ILF outperforms EXPERT by 4% on small contracts and 19% on large contracts. We found EXPERT times out more often on large contracts and thus achieves lower coverage. This indicates that ILF has the advantage of generating transactions orders of magnitude faster than EXPERT and thus covers more code than EXPERT within a reasonable amount of time.

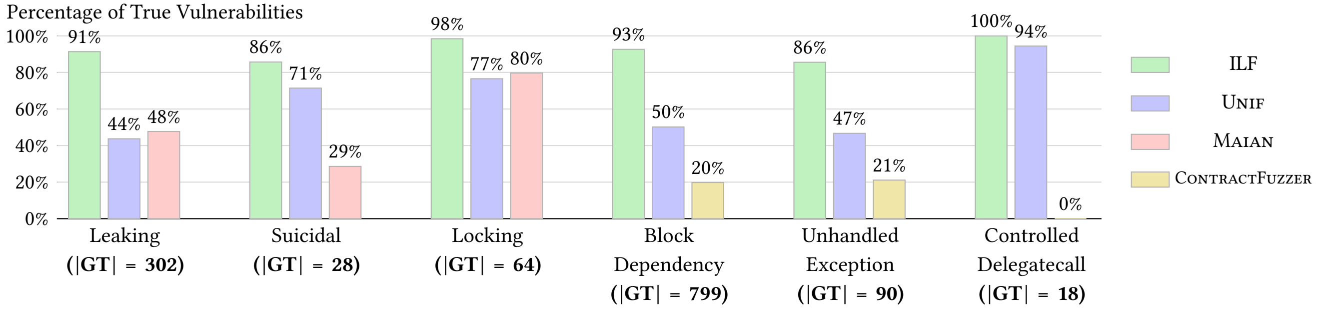

Figure 10: Fraction of true vulnerabilities found by each individual system. The number of ground truth vulnerabilities, which are computed as the union of all true vulnerabilities detected by each system, are shown under the x-tick labels.

6.4 Vulnerability Detection

We ran ILF, Unif, Maian and ContractFuzzer for detecting vulnerabilities in our dataset. We first describe the setup for running these tools and report their speed. Then, we describe how we obtain the ground truth of vulnerabilities. Finally, we compare ILF with each individual tool on vulnerability detection.

Setup and Speed. ILF and Unif ran for 2K transactions on each contract, spending on average 14s and, respectively, 8s. ContractFuzzer fuzzes a set of contracts together. We ran it on our whole dataset with 40 threads until it exited automatically, which spent 62h. Maian detects each vulnerability in separate runs. It spent (with default depth 3) on average 61s, 15s and 13s to detect Leaking, Suicidal and Locking, respectively, on each contract.

Obtaining Ground Truth of Vulnerabilities. The detectors in ILF and Unif report only true positives. For Leaking and Suicidal, Maian concretely validates exploits produced by symbolic execution to filter false positives and then reports only true positives. For Locking, it may report false positives. ContractFuzzer produces only true positives on Unhandled Exception and Controlled Delegatecall but can have false positives on Block Dependency.

By manual inspection, we found that Maian reports 13 false positives for Locking and ContractFuzzer produces 6 false positives for Block Dependency. We take the union of all true vulnerabilities found by each system as ground truth and report the fraction of each system (over the ground truth). Figure 10 shows our results, indicating that ILF outperforms all other systems on all types of vulnerabilities. We next compare ILF to each individual system.

Comparing ILF to Uniformly Random Baseline. ILF finds on average 43% more Leaking, Block Dependency, and Unhandled Exception vulnerabilities than Unif. ILF also improves over Unif on the other vulnerabilities: 15% on Suicidal, 22% on Locking, and 6% on Controlled Delegatecall. By inspection, we found that ILF triggers more ether transfers, which results in detecting 47% more Leaking and 43% more Block Dependency vulnerabilities than Unif.

Comparing ILF to a Symbolic Execution Detector. We now compare ILF to Maian. ILF detects 43% more Leaking vulnerabilities than Maian. Inspection indicates that Maian's false negatives are mostly because (i) it fails to send ether to those contracts or (ii) it stops before the required depth to trigger the vulnerabilities. For Suicidal, ILF detects 86%, while Maian detects 29%. Maian does not report some vulnerabilities that can be revealed within 3 transactions (*i.e.*, it fails to explore some paths at depth 3). ILF detects 18% more Locking vulnerabilities than Maian. Maian reports 29 false positives out of the total 80 Locking reports.

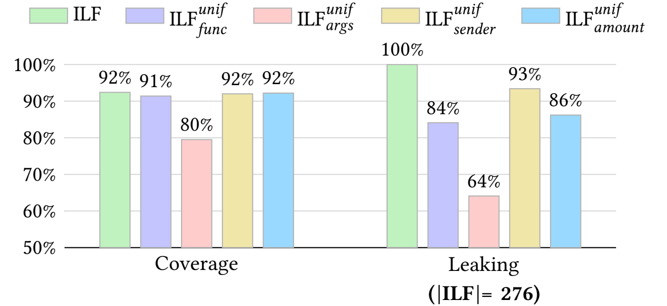

Figure 11: Comparison between ILF and the four baselines with components replaced by their counterparts in Unif.

Comparing ILF to an Existing Fuzzer. We now compare ILF to ContractFuzzer. Overall, ILF finds substantially more vulnerabilities than ContractFuzzer. This is likely because the randomly generated transactions by ContractFuzzer fail to cover vulnerable code, unlike ILF which uses its learned strategies to explore the code. ContractFuzzer generates 6 false positives for Block Dependency because its detection rule syntactically checks for the existence of ether transfer and block state variables in the trace, while ILF uses a semantic dataflow check.

6.5 Importance of Policy Components

We now evaluate the relevance of components π^{nn}_{func}, π^{nn}_{args}, π^{nn}_{sender} and π^{nn}_{amount} used in ILF's fuzzing policy. We show four additional baselines in Figure 11, where each baseline replaces one component in ILF with its counterpart in Unif. For example, ILF^{unif}_{func} replaces π^{nn}_{func} with π^{unif}_{func} while keeping the other three intact. We compare ILF with the additional baselines in terms of coverage and the number of Leaking vulnerabilities detected after 2K transactions (ILF detects 276). We decide on this vulnerability as it usually involves multiple transactions. Our results show that π^{nn}_{args} is required for achieving high coverage and critical for detecting the Leaking vulnerability. The remaining three baselines are all sub-optimal in detecting the Leaking vulnerability.

```
contract Grid {
  struct Pixel { address owner; uint256 price; }
  address admin;
  mapping(address => uint256) pending;
  Pixel[1000][1000] pixels
  uint256 public defaultPrice;

  function Grid() public {
    admin = msg.sender;
    defaultPrice = 2 * 10 ** 15;
  }
  function setDefaultPrice(uint256 price) public {
    require(admin == msg.sender);
    defaultPrice = price;
  }
  function buyPixel(uint16 row, uint16 col)
      public payable {
    var (owner, price) = getPixelInfo(id);
    require(msg.value >= price);
    pending[owner] += msg.value;
    pixels[row][col].owner = msg.sender;
  }
  function withdraw() public {
    uint256 amount = pending[msg.sender];
    pending[msg.sender] = 0;
    msg.sender.transfer(amount);
  }
  function getPixelInfo(uint16 row, uint16 col)
      public returns (address, uint256) {
    Pixel pixel = pixels[row][col];
    if (pixel.owner == 0) return (admin, defaultPrice);
    return (pixel.owner, pixel.price);
  }
}
```

Figure 12: A contract (simplified[2]) where ILF achieves high coverage and detects a new Leaking vulnerability.

6.6 Case Studies

We now present three case studies. The first one gives insights on how ILF achieves high coverage and detects a Leaking vulnerability. The second one illustrates a false negative of ILF in detecting a Leaking vulnerability. Finally, in the third case study, we show a contract where ILF achieves lower coverage than Expert.

Illustrating Fuzzing Strategy of ILF. Figure 12 shows a contract where ILF achieves 98% coverage (Unif and Echidna achieve 57% and 59%, respectively). Function `getPixelInfo` has an if condition `pixel.owner == 0` at Line 31. If the condition holds, the pixel does not exist and default values are returned. Otherwise, the pixel's `owner` and `price` are returned (Line 32). To cover Line 32, one must call `buyPixel` beforehand to create a pixel and reuse values of arguments `row` and `col` to call `getPixelInfo`. Unif and Echidna fail to cover Line 32 because they generate arguments of type `uint16` uniformly at random which makes it almost unlikely to reuse values for `row` and `col`. ILF, however, samples integer arguments from a probability distributions over a set of seed integers, and this distribution assigns more weights to certain values (*e.g.*, 0 and 1). We also inspected ILF's probability distributions over functions. We found that it initially assigns almost equal probabilities to all functions. Later during the fuzzing process, ILF gradually assigns lower probability (*e.g.*, less than 0.001) to simpler functions, such as `defaultPrice`, and higher probability (*e.g.*, more than 0.1) to more complex functions such as `getPixelInfo` and `buyPixel`.

[2] https://etherscan.io/address/0x9d6cde970c30e45311cff3d1f3f9a8dd24ff70f1#code

```
contract TransferableMultsigImpl {
  function () payable public {}
  function execute(uint8[] sigV, bytes32[] sigR,
      bytes32[] sigS, address destination,
      uint val, bytes data) external {
    bytes32 txHash = keccak256(...);
    verifySignatures(sigV, sigR, sigS, txHash);
    require(destination.call.value(val)(data));
  }
  function verifySignatures(
      uint8[] sigV, bytes32[] sigR,
      bytes32[] sigS, bytes32 txHash) internal {
    ... // complex operations and requires
  }
}
```

Figure 13: A false negative (simplified version[3]) in detecting a Leaking vulnerability.

```
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
  }
}
```

Figure 14: An example contract (simplified version[4]) where ILF achieves lower coverage than Expert.

In addition to achieving high coverage on this example, ILF also exposes a Leaking vulnerability not found by Maian and Unif. A sequence of transactions exposing the vulnerability is:

t_1: `admin` calls `setDefaultPrice(0)` making the contract vulnerable.
t_2: An attacker calls `buyPixel(1, 2)` with `msg.value` = 0.
t_3: A user (not the attacker) calls `buyPixel(1, 2)` with `msg.value` > 0.
t_4: The attacker calls `withdraw` to steal earned ether from the user.

A False Negative on Detecting Leaking Vulnerability. In Figure 13, we present a contract with Leaking vulnerability which is detected by Maian but not by ILF. An attacker can call function `execute` with argument `destination` set to the attacker's address to steal ether. To ensure success of the call, its arguments must satisfy the complex constraints imposed by the call to `verifySignatures`. ILF failed to generate such arguments.

A Contract Where ILF Achieves Lower Coverage Than Expert. In Figure 14 we show a contract where Expert achieves 95% instruction coverage while ILF achieves only 74%. Function `giveKudos` requires the block's timestamp (returned by `now`) to be set to a value in [`voteStart`, `voteEnd`). However, `voteStart` and `voteEnd` are contract-specific values defined by the constructor of the contract, which ILF can hardly learn.

[3] https://etherscan.io/address/0x38c1908487de0b82e9ed6d416bc50d5ab08eac75#code
[4] https://etherscan.io/address/0x5e569e1ecd56fe30dd97ee233ec1675b60fb6680#code

7 DISCUSSION

Now we discuss ILF's limitations and potential future improvement.

Handling Contracts with Unique Functionality. ILF is effective under the assumption that most contracts have similar functionality [34]. However, some contracts implement unique functionality not appearing in other contracts (*e.g.*, the contracts in Figure 13 and Figure 14 have contract-specific arguments and variables).

While the learned fuzzer has limited effectiveness in dealing with unseen functionality, symbolic execution can effectively handle custom constraints. To this end, we propose a new policy π^{mix}, that combines π^{nn} and π^{expert} as a preliminary solution for handling unseen functionality. The goal of π^{mix} is to combine the strengths of the learned and the expert policies: π^{nn} is fast at handling functionality similar to that in the dataset, while π^{expert} is slow but effective at handling unseen functionality. To generate a transaction, π^{mix} runs π^{expert} with probability p and π^{nn} with probability $1 - p$. The algorithm for running π^{mix} is defined in Appendix F.

We evaluate π^{mix} on 59 contracts in our dataset where Expert achieves >20% higher coverage than ILF. We inspected those contracts and confirmed that they all have unseen functionality. We ran π^{mix} on the 59 contracts with a timeout of 20m per contract and p set to 0.1. On average, π^{mix} achieves 73% coverage using 230s (π^{nn} used by ILF achieves 57% using 14s). Moreover, π^{mix} discovers 9 more Block Dependency vulnerabilities than π^{nn}.

Handling Contract Interaction. Ethereum smart contracts often have interaction by calling each other. ILF supports fuzzing multiple interacting contracts that are deployed on the local test blockchain. We note, however, that dependencies between the interacting contracts are not considered when generating transactions. At each step, ILF samples a contract uniformly at random and generates a transaction for the sampled contract with π^{nn}, which considers only the state of the sampled contract and ignores the other contracts. For fuzzing a set of interacting contracts, the probability distribution for generating transactions needs to be defined over multiple contracts. Using the framework of imitation learning, one can extend ILF with a neural policy for selecting a target contract before generating a transaction. To support such a policy, a revised neural network architecture handling contract selection and new features abstracting contract interaction are needed.

8 RELATED WORK

We now discuss the works that are most closely related to ours.

Fuzz Testing. Numerous techniques for fuzz testing have been proposed in prior work. White-box fuzzing techniques leverage symbolic/concolic execution [13, 14, 52, 67] to solve path constraints and generate effective inputs. NEUEX [54] leverages neural constraints to capture dependencies between program variables and approximate exact symbolic constraints. Generation-based fuzzers create inputs based on input specifications which can be defined with a grammar [21, 30, 38, 66], or learned from a given corpus [10, 23, 31, 64]. Mutation-based fuzzers [15, 39, 45, 48, 59, 63, 68] mutate initial seeds to craft new inputs that may expose vulnerabilities. Static analysis [39, 48], taint analysis [15, 48], and symbolic/-concolic execution [45, 59] are used to guide mutation processes.

Recent years have witnessed an interest in improving fuzzing with machine learning. Several works [18, 23, 64] learn generative models from existing input corpus. AFLFast [12] and NEUZZ [53] model program branching behavior using Markov Chains and NNs, respectively, learned from inputs generated during fuzzing. Unfortunately, their models may not be ideal as the datasets used for training are sub-optimal. In contrast, our training dataset generated using symbolic execution is high-quality.

Security Analysis of Smart Contracts. A number of systems have been proposed for detecting vulnerabilities in smart contracts. Static analysis systems based on Datalog [24, 62] or the LLVM analysis engine [33] provide soundness guarantee but can have false positives and do not generate exploits. Erays [69] is a reverse engineering tool producing high-level pseudocode from EVM bytecode to help security inspection. Grossman et al. [25] specifically detect reentrancy through dynamic linearization checker.

Symbolic execution has been widely applied to smart contract security. Oyente [40] is the first symbolic engine to detect vulnerabilities. Osiris [61] detects integer bugs. However, they cannot deal with multiple transactions. teEther [37] (only detects Leaking) and Maian [42] can generate exploits over multiple transactions. However, they are limited to a small depth (*e.g.*, 3).Our learned fuzzer explores deep states rapidly to locate vulnerabilities.

Several works have studied formal verification of smart contracts. KEVM [28] is an executable formal specification of EVM bytecode in the $\mathbb{K}$ Framework [50]. The authors of [11] and [6, 29] formulate smart contracts in F* and the Isabelle/HOL proof assistant, respectively. VerX [46] is the first automated verifier for proving functional properties of contracts.

Machine Learning for Security. Machine learning methods have been applied in various aspects of computer security. Bao et al. [9] and Shin et al. [55] learn to recognize functions in stripped binaries. Debin [26] leverages ensemble tree models and probabilistic graphical models to predict debug information for stripped binaries. Naive Bayes, GRU and convolutional networks have been used in malware classification [7, 36, 51].

Reinforcement learning has been applied on various programming language tasks, such as speeding up numerical program analysis [58] and SMT solvers [8], inferring loop invariants for program verification [56], and program reduction [27]. While those works treat their problems as MDP, they do not learn from an expert. Our work instead formulates fuzzing as an MDP and learns a policy by imitating a symbolic execution expert.

9 CONCLUSION

We presented a new approach for learning a fuzzer from symbolic execution and instantiated it to the domain of smart contracts. The key idea was to learn a fuzzing policy, represented using neural networks, which is trained over a high-quality dataset of inputs generated using a symbolic execution expert. The learned policy is then used to generate inputs for fuzzing unseen smart contracts. We implemented our method in a system called ILF and showed that it is fast and achieves high code coverage on real-world contracts. The high coverage facilitates and improves the detection of critical vulnerabilities, outperforming existing tools.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. We also thank Anton Permenev for his support with the implementation of the symbolic execution expert.

REFERENCES

- [1] 2016. Post-Mortem Investigation (Feb 2016). https://www.kingoftheether.com/postmortem.html
- [2] 2019. Pytorch. https://pytorch.org/
- [3] 2019. Solidity Documentation. https://solidity.readthedocs.io/en/v0.5.8/
- [4] Pieter Abbeel, Adam Coates, and Andrew Y. Ng. 2010. Autonomous Helicopter Aerobatics through Apprenticeship Learning. *I. J. Robotics Res.* 29, 13 (2010), 1608–1639. https://doi.org/10.1177/0278364910371999
- [5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* https://openreview.net/forum?id=BJOFETxR-
- [6] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018.* 66–77. https://doi.org/10.1145/3167084
- [7] Ben Athiwaratkun and Jack W. Stokes. 2017. Malware Classification with LSTM and GRU Language Models and a Character-level CNN. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017.* 2482–2486. https://doi.org/10.1109/ICASSP.2017.7952603
- [8] Mislav Balunovic, Pavol Bielik, and Martin Vechev. 2018. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.* 10338–10349. http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas
- [9] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* 845–860. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao
- [10] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 95–110. https://doi.org/10.1145/3062341.3062349
- [11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, et al. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016.* 91–96. https://doi.org/10.1145/2993600.2993611
- [12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 1032–1043. https://doi.org/10.1145/2976749.2978428
- [13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings.* 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
- [14] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006.* 322–335. https://doi.org/10.1145/1180405.1180445
- [15] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* 711–725. https://doi.org/10.1109/SP.2018.00046
- [16] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL.* 1724–1734. http://aclweb.org/anthology/D/D14/D14-1179.pdf
- [17] Crytic. 2019. Echdina. https://github.com/crytic/echidna/
- [18] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018.* 95–105. https://doi.org/10.1145/3213846.3213848
- [19] Etherscan. 2019. Ethereum (ETH) block explorer. https://etherscan.io/
- [20] The go-ethereum Authors. 2019. Go Ethereum. https://geth.ethereum.org/
- [21] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* 206–215. https://doi.org/10.1145/1375581.1375607
- [22] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008.* http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf
- [23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017.* 50–59. https://doi.org/10.1109/ASE.2017.8115618
- [24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. *PACMPL* 2, OOPSLA (2018), 116:1–116:27. https://doi.org/10.1145/3276486
- [25] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136
- [26] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1667–1680. https://doi.org/10.1145/3243734.3243866
- [27] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 380–394. https://doi.org/10.1145/3243734.3243838
- [28] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018.* 204–217. https://doi.org/10.1109/CSF.2018.00022
- [29] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers.* 520–535. https://doi.org/10.1007/978-3-319-70278-0_33
- [30] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012.* 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
- [31] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016.* 720–725. https://doi.org/10.1145/2970276.2970321
- [32] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* 259–269. https://doi.org/10.1145/3238147.3238177
- [33] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf
- [34] Lucianna Kiffer, Dave Levin, and Alan Mislove. 2018. Analyzing Ethereum's Contract Topology. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018.* 494–499. https://dl.acm.org/citation.cfm?id=3278575
- [35] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* https://openreview.net/forum?id=SJU4ayYgl
- [36] Bojan Kolosnjaji, Ghadir Eraisha, George D. Webster, Apostolis Zarras, and Claudia Eckert. 2017. Empowering Convolutional Networks for Malware Classification and Analysis. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017.* 3838–3845. https://doi.org/10.1109/IJCNN.2017.7966340
- [37] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[38] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 216–226. https://doi.org/10.1145/2594291.2594334

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 627–637. https://doi.org/10.1145/3106237.3106295

[40] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 254–269. https://doi.org/10.1145/2976749.2978309

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality

[42] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 653–663. https://doi.org/10.1145/3274694.3274743

[43] OpenZeppelin. 2019. OpenZeppelin is a Library for Secure Smart Contract Development. https://github.com/OpenZeppelin/openzeppelin-solidity

[44] Santiago Palladino. 2017. The Parity Wallet Hack Explained. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[45] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 697–710. https://doi.org/10.1109/SP.2018.00056

[46] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-chohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Jose, CA, USA, May 18-20, 2020*.

[47] Dean Pomerleau. 1988. ALVINN: An Autonomous Land Vehicle in a Neural Network. In *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]*. 305–313. http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network

[48] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[49] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*. 627–635. http://jmlr.org/proceedings/papers/v15/ross11a/ross11a.pdf

[50] Grigore Rosu and Traian-Florin Serbanuta. 2010. An Overview of the K Semantic Framework. *J. Log. Algebr. Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[51] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*. 38–49. https://doi.org/10.1109/SECPRI.2001.924286

[52] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 263–272. https://doi.org/10.1145/1081706.1081750

[53] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Learning. *CoRR* abs/1807.05620 (2018). arXiv:1807.05620 http://arxiv.org/abs/1807.05620

[54] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/neuro-symbolic-execution-augmenting-symbolic-execution-with-neural-constraints/

[55] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 611–626. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[56] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 7762–7773. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification

[57] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489. https://doi.org/10.1038/nature16961

[58] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 211–229. https://doi.org/10.1007/978-3-319-96145-3_12

[59] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[60] Parity Technologies. 2017. Security Alert. https://www.parity.io/security-alert-2/

[61] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 664–676. https://doi.org/10.1145/3274694.3274737

[62] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 67–82. https://doi.org/10.1145/3243734.3243780

[63] Petar Tsankov, Mohammad Torabi Dashti, and David A. Basin. 2012. SECFUZZ: Fuzz-testing Security Protocols. In *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*. 1–7. https://doi.org/10.1109/IWAST.2012.6228985

[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 579–594. https://doi.org/10.1109/SP.2017.23

[65] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper* (2014).

[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294. https://doi.org/10.1145/1993498.1993532

[67] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[68] Michal Zalewski. 2019. American Fuzzy Loop. http://lcamtuf.coredump.cx/afl/

[69] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 1371–1385. https://www.usenix.org/conference/usenixsecurity18/presentation/zhou

A OPCODE FEATURES

We list the 50 distinctive opcodes used as features to represent functions and categorize them into 15 kinds by their functionalities:

- hash: `sha3`.
- gas: `gas`, `gasprice`.
- execution: `pc`, `msize`.
- error: `revert`, `assert`.
- logic: `and`, `or`, `xor`, `not`.
- storage: `sload`, `sstore`.
- logging: `log0`, `log1`, `log2`, `log3`, `log4`.
- comparison: `lt`, `gt`, `slt`, `sgt`, `iszero`.
- account: `address`, `origin`, `caller`, `balance`.
- creation/destruction: `create`, `selfdestruct`.
- return: `return`, `returndatasize`, `returndatacopy`.
- call: `call`, `callcode`, `delegatecall`, `staticcall`.
- code: `codesize`, `codecopy`, `extcodesize`, `extcodecopy`.
- calldata: `callvalue`, `calldataload`, `calldatasize`, `calldatacopy`.
- block: `blockhash`, `coinbase`, `timestamp`, `number`, `difficulty`, `gaslimit`.

B GRAPH CONVOLUTIONAL NETWORK TO IMPROVE FEATURE REPRESENTATION

We first introduce construction of dependency graph over functions and then discuss how Graph Convolutional Network is used to embed a dependency graph into features.

Dependency Graph. We define an undirected graph $G = (F, E)$ over the contract's functions F where the edges E represent dependencies between functions. To derive dependencies, we collect the read set $read(f)$ and the write set $write(f)$ for each function f, which contain the set of read and, respectively, write storage locations (which store values of fields) that appear in f. An edge (f, f') is then added to E if the read and write sets of the functions f and f' are overlapping:

$$read(f) \cap write(f') \neq \varnothing \text{ or } write(f) \cap read(f') \neq \varnothing.$$

We also add self edges. The intuition for such dependency is that to successfully call function f^i, one may need to call function f^j that sets specific fields used in f^i. This holds for both of our examples in Figure 2 and Figure 12.

To extract the read and write sets for each contract function, we perform a lightweight static analysis over the contract's bytecode. First, it disassembles the bytecode and constructs control flow graph from the disassembled instructions. Then, we recover function entry blocks by their unique 4-byte identifiers (EVM uses the identifiers like a dispatch table to jump to entry of each function). For every function, we traverse its basic blocks starting from its entry, compute backward slicing of each `sload` and `sstore` instruction, and recover the read and write offsets.

Graph Convolutional Network. Given a (dependency) graph $G = (F, E)$, Graph Convolutional Network (GCN) [35] is used to embed dependencies (edges E) between vertices F to produce feature embedding for vertices. It is usually composed of multiple graph convolutional layers. A graph convolutional layer $C^i : \mathbb{R}^{|F| \times d_i} \rightarrow \mathbb{R}^{|F| \times d_{i+1}}$ performs the following operation:

$$C^i(H^{i-1}) = \sigma(A\, H^{i-1} W^i)$$

where $A : \mathbb{R}^{|F| \times |F|}$ is the adjacency matrix of F, $H^{i-1} : \mathbb{R}^{|F| \times d_i}$ is the output of the previous layer, $W^i : \mathbb{R}^{d_i \times d_{i+1}}$ is the learned weight matrix and σ is an activation function.

In our work, we leverage a 3-layer GCN with hidden dimensions set to 100 and ReLU as activation function. Its input is extracted semantic feature matrix $\alpha_s(F, \bar{t})$ defined in Section 5.1 and output matrix $\alpha(F, \bar{t})$ is used as input to π^{nn}. Training of *GCN* is performed jointly with π^{nn}. During the back-propagation process described in Section 4.2, gradients are also passed to *GCN*, thus improving the learned embeddings.

C DETAILS ON VULNERABILITY DETECTION

We now elaborate on the six vulnerability detectors.

Locking. A bug in November 2017 locked 160M in a contract [60]. The contract relied on another library contract to send out ether. Unfortunately, the library contract was accidentally killed by a user. We detect Locking vulnerabilities if a contract can receive ether from other accounts but cannot send ether out. That is, a contract does not contain `create`, `call`, `delegatecall`, and `selfdestruct` opcodes (only ways to send out ether), and it can receive ether.

Leaking and Suicidal. A contract is considered to have *Leaking* vulnerability if it leaks ether to attackers. Similarly, A contract is considered *Suicidal* [42] if it can be killed by attackers. Leaking and Suicidal are the most challenging vulnerabilities to detect because multiple transactions with correct order are needed. To define the notion of *attacker* in these two vulnerabilities, we first define *trusted* users. The set of trusted users consists of the contract's creator and all users whose addresses have appeared as arguments in transactions sent by trusted users. All remaining users are considered as *untrusted*. This definition ensures that the contract's ownership or administrative permission is not transferred to untrusted users. An *attacker* is an untrusted user who never sends ether to the tested contract. A contract is considered to have Leaking (Suicidal) vulnerability if any attacker receives ether (kills the contract).

Block Dependency. A contract has Block Dependency vulnerability if there exists a transaction that (i) transfers ether, and (ii) the behavior of sending ether depends on block state variables (*i.e.*, output of `timestamp`, `number`, `difficulty`, `gaslimit`, and `coinbase` opcodes). Such dependency can be explicit (*i.e.*, the variables flow into amount of ether to transfer) or implicit (*i.e.*, the variables flow into conditions for triggering the transfer). An attacker can observe block state and generate transactions to achieve malicious behaviors (*e.g.*, earning profit), or a malicious miner can set certain variables (*e.g.*, `timestamp`) to exploit vulnerable contracts.

Unhandled Exception. The contract *King of the Ether* was vulnerable due to not handling an exception of a `call` opcode, forcing the developers to publicly ask users to stop sending ether to it [1]. We identify a contract with Unhandled Exception vulnerability if there exists a transaction during fuzzing where (i) the root call does not throw exception, and (ii) any of child calls throws exception.

Controlled Delegatecall. The `delegatecall` opcode is designed for invoking library contracts. It takes as argument the address of a library contract, loads the library contract's code, and executes

Table 6: A set of 50 seed integers and their appearance count produced by π^{expert} solving SMT constraints on one of the five training datasets.

Integer	Appearance
0	10585
1	3058
$2^{256} - 1$	2531
8	702
2	577
4	354
3	299
5	95
9	84
0x10	80
0x20	75
6	73
2^{255}	67
0x400000000000000000000	55
0x1000000000000000000000	54
0x2000	52
0x80	49
0x4000000000000000000000	46
0x800000	44
0x40	44
0x10000	43
0x400	42
0x4000000	41
0x200000000	40
0x800000000000000000000	40
0x1000000	39
0x20000000	39
0x40000	38
0x20000	38
0x20000000000000000000000	37
0x4000000000000	37
0x200	36
0x80000	36
0x8000000000000000000000	36
0x8000	36
0x800	36
0x1000	35
0x80000000000	35
0x2000000000000000000000	33
0x8000000	33
1000000000000000000000000001	33
0x4000000000000000000	33
0x100000000	32
0x200000000000000000000	32
0x800000000000	32
0x80000000	31
0x1000000000	31
0x100	31
0x80000000000000	30
0x200000000000	30

Table 7: A set of 50 seed amounts and their appearance count produced by π^{expert} solving SMT constraints on one of the five training datasets.

Amount	Appearance
99999999999999999999999999999	1661
0	787
1	234
0x1000000000000000000000000	168
0x30000000000000	28
1000000000000000000	21
0x180000000000000	18
100000000000000000	18
10000000000000000	16
1000000000000000	16
2	14
5000000000000000	13
0x20	11
0x700000000000000	10
8	9
0x3c00000000000	9
0xe00000000000000	8
0x400000000000000000000000	8
50000000000000000	7
500000000000000000	7
0x18000000000000	6
3	6
0x80	6
0x300000000000000	6
0x1000000000000000000000001	5
5000000000000000000	5
0x1c00000000000000	5
0x4	5
10000000000000000000	5
0xc000000000000	5
0x2000	5
20000000000000000	4
0x40	4
200000000000000000	4
2000000000000000	4
0x800000000000000000000	4
0x800000000000000000000000	4
0x1000000000000000000000002	3
0x400	3
0x80000000000000	3
0x100000000000000	3
0xc00000000000	3
0x1800000000000000000	3
0x800000000000000000	3
0x70000000000000	3
250000000000000	3
0x380000000000000	3
0x8000000000000000000	3
0x8000000000000000	3
0x1000	3

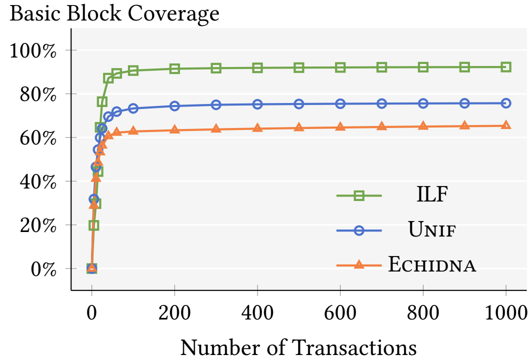

Figure 15: Basic block coverage of fuzzers on small contracts.

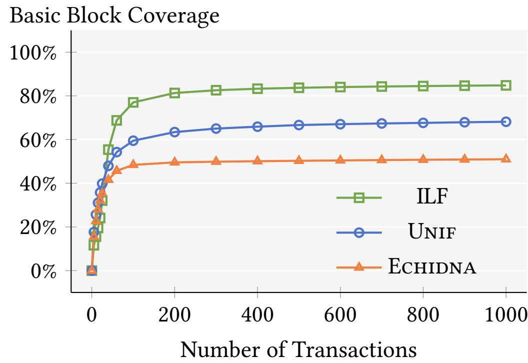

Figure 16: Basic block coverage of fuzzers on large contracts.

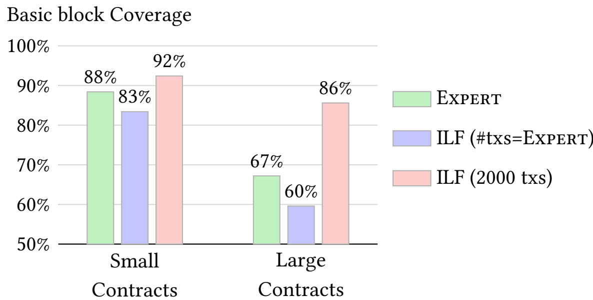

Figure 17: Basic block coverage of Expert and ILF.

Algorithm 2: Algorithm for running π^{mix} Policy.

1 **Procedure** RunMix(c, p, b):
  Input : Contract c
  Probability p to run π^{expert}
  Starting block state b
2 $\bar{t} \leftarrow \varnothing$
3 **while** *true* **do**
4 $t = \bot$
5 **if** Random() $< p$ **then** $t \leftarrow \pi^{expert}(\bar{t})$
6 **if** $t == \bot$ **then** $t \sim \pi^{nn}(\bar{t})$
7 $b \leftarrow$ Execute(t, b, c)
8 $\bar{t} \leftarrow \bar{t} \cdot t$

it in the context of the caller contract. If an attacker can manipulate the argument of the `delegatecall` opcode, then she can cause the contract to execute arbitrary code. An attack that exploits a Controlled Delegatecall vulnerability of the *Parity Multisig Wallet* contract led to $30M loss [44]. We detect a contract with Controlled Delegatecall vulnerability if transactions parameters explicitly flow into arguments of `delegatecall`.

D EXAMPLES OF SEED INTEGERS AND AMOUNTS

We show a set of 50 seed integers *SI* in Table 6 and a set of 50 seed amounts *SA* in Table 7. They are produced by π^{expert} running on one of our training sets (there are 5 training sets in total because we perform 5-fold cross validation in our experiments).

E EVALUATION RESULTS WITH BASIC BLOCK COVERAGE

Apart from results on instruction coverage discussed in Section 6.3, we present results on basic block coverage. We compare basic block coverage achieved by ILF, Unif and Echidna in Figure 15 (on small contracts) and in Figure 16 (on large contracts), and compare ILF with Expert in Figure 17. From the results on basic block coverage, we can draw similar conclusions as for instruction coverage.

F COMBINING LEARNED FUZZER WITH SYMBOLIC EXECUTION

We introduce a mixed policy π^{mix} that randomly chooses learned policy π^{nn} and symbolic expert policy π^{expert} to generate transactions. The goal of π^{mix} is to combine the strengths of the learned and the expert policies: π^{nn} is fast at handling functionality similar to that in the dataset, while π^{expert} is slow but effective at handling unseen functionality. In Algorithm 2, we outline the algorithm for running π^{mix}. At Line 5, the algorithm tries to call π^{expert} with probability p. At Line 6, if the algorithm decides not to call π^{expert} or π^{expert} cannot find a transaction improving coverage (in both cases, $t == \bot$ holds), it falls back to call π^{nn}. Then the generated transaction t is executed and block state is updated.