Adaptive Mutation Scheduling with Deep Reinforcement Learning for Smart Contract Fuzzing

QIANQIAN PANG[†], Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIN YIN[†], Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
TINGTING BI, The University of Melbourne, Australia
LINGFENG BAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
CHAO NI[*], Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIAOHU YANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Smart contracts underpin a wide range of decentralized applications—from financial services to supply-chain management—but their immutability and direct control of assets magnify the impact of any security bugs. Although many fuzz approaches have been proposed and have demonstrated their effectiveness in uncovering vulnerabilities, existing methods often rely on unguided random mutation scheduling, generate redundant inputs, and fail to adapt to smart contract-specific characteristics. To overcome these challenges, we present FuzzMaster, a feedback-driven fuzzing framework that combines deep reinforcement learning (DRL) with lightweight probabilistic scheduling to steer mutation selection at runtime intelligently. By continuously analyzing execution feedback—code coverage, function-call sequences, and vulnerability signals—FuzzMaster's DRL agent and probabilistic tables prioritize high-impact mutations and avoid wasted effort on redundant seeds. On standard VeriSmart and SmartBugs benchmarks, FuzzMaster achieves a 66.2% detection rate with 100% precision (versus 46.9% for ItyFuzz and 43.1% for Confuzzius) and uncovers most bugs within the first second of execution. Meanwhile, in real-world Ethereum contracts, FuzzMaster identified 97 vulnerabilities in 6 categories. These results demonstrate that dynamic, vulnerability-aware mutation scheduling can dramatically improve both the efficiency and effectiveness of smart contract fuzz testing.

CCS Concepts: • **Software and its engineering** → Software maintenance tools.

Additional Key Words and Phrases: Smart Contract, Fuzzing, Reinforcement Learning

[†]Equal contribution.
[*]This is the corresponding author. Chao Ni is also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou, China.

Authors' Contact Information: Qianqian Pang, Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, qianqianpang@zju.edu.cn; Xin Yin, Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xyin@zju.edu.cn; Tingting Bi, The University of Melbourne, Melbourne, Australia, tingting.bi@unimelb.edu.au; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, lingfengbao@zju.edu.cn; Chao Ni, Innovation and Management Center (Ningbo) for School of Software Technology, Zhejiang University; The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, chaoni@zju.edu.cn; Xiaohu Yang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, yangxh@zju.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE114
https://doi.org/10.1145/3808121

ACM Reference Format:
Qianqian Pang, Xin Yin, Tingting Bi, Lingfeng Bao, Chao Ni, and Xiaohu Yang. 2026. Adaptive Mutation Scheduling with Deep Reinforcement Learning for Smart Contract Fuzzing. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE114 (July 2026), 23 pages. https://doi.org/10.1145/3808121

1 Introduction

Smart contracts are programmable protocols deployed on blockchain platforms (e.g., Ethereum [1] and EOS [2]) that automatically execute predefined terms without intermediaries. These self-executing programs have become fundamental to decentralized applications, governing financial transactions, supply chain management, and decentralized applications (DApps). However, their immutable nature makes post-deployment fixes impossible, amplifying the consequences of security vulnerabilities. In 2016, attackers exploited a reentrancy vulnerability to steal $50 million worth of Ether [3], exemplifying the critical need for robust pre-deployment vulnerability detection.

Vulnerability detection techniques for smart contracts fall into two main categories: **static analysis** and **dynamic analysis**. Static approaches employ pattern matching [20, 33, 51] or symbolic execution [29, 40, 52] to verify predefined security properties. However, pattern-matching approaches suffer from false positives due to limited reachability analysis, while symbolic execution faces scalability challenges including path explosion and constraint solving complexity. Dynamic analysis encompasses runtime monitoring [14, 42] and fuzzing [16, 26, 27, 38, 50, 55]. While monitoring systems can only detect known attack patterns with limited coverage, fuzzing systematically generates diverse inputs to uncover unknown vulnerabilities with higher effectiveness [23, 24, 35, 60]. The core mechanism of fuzzing involves generating extensive test cases through mutations of input data, executing the target program, and monitoring its behavior to identify anomalous patterns. This execution-driven perspective enables fuzzing to reveal intricate, path-dependent vulnerabilities that static analysis often overlooks [9, 23–25, 28, 30, 32, 35, 48, 53, 57, 58, 60]. However, existing fuzzing approaches for smart contracts face several critical limitations:

- **Random Mutation**: Most fuzzers randomly select mutation operators from fixed pools [16, 27, 38, 54], leading to inefficient allocation of testing resources and missed vulnerabilities.
- **Redundant Seeds**: Mutation scheduling strategies often generate repetitive seeds that fail to trigger new behaviors. This leads to the skipping of critical execution paths, thereby causing inefficient state space exploration and missed vulnerabilities.
- **Ignoring Contract Specifics**: Current techniques fail to adapt to smart contract-specific characteristics, including EVM semantics, gas constraints, and complex logical interactions in the contracts that are crucial for vulnerability discovery.

To address these limitations, we introduce FuzzMaster, a novel fuzzing framework that heuristically schedules the mutation operators, dynamically adjusting its strategies based on real-time execution feedback. Specifically, our method incorporates a reinforcement learning model that dynamically selects mutation operators during fuzzing. The model learns from feedback based on previous test outcomes and adjusts its strategy to maximize the rewards. In parallel, we employ probabilistic tables as a lightweight scheduling solution, offering a flexible approach to guide mutation choices.

Experimental evaluation demonstrates FuzzMaster's superior performance: FuzzMaster achieves a 66.15% vulnerability detection rate with 100% precision on benchmark contracts, outperforming state-of-the-art fuzzers including ItyFuzz (46.92%) and Confuzzius (43.08%). Furthermore, FuzzMaster identifies the majority of vulnerabilities within the first second of execution, illustrating its ability to rapidly target critical contract behaviors and reduce unnecessary exploration.

Specifically, we make the following contributions:

- We present FuzzMaster, a novel fuzzing framework that combines DRL and probabilistic scheduling to intelligently guide mutation operator selection based on real-time feedback, ensuring efficient and targeted vulnerability discovery.
- We collect a comprehensive dataset of smart contracts, including both labeled typical contracts and a large number of recently deployed contracts from the Ethereum blockchain.
- We conduct extensive evaluation demonstrating that FuzzMaster outperforms existing tools in both vulnerability detection effectiveness and speed.

2 Motivation

Fuzzing is critical for smart contract security, yet existing tools suffer from bottlenecks due to naive mutation strategies. This section motivates the need for a more intelligent approach. We first analyze the inefficiencies of representative fuzzers that utilize random mutation selection. We then demonstrate that overcoming these limitations requires runtime feedback and adaptive scheduling, effectively setting the stage for our proposed solution.

2.1 Random and Fixed Distribution Mutation Selection

Current fuzzing frameworks suffer from a major limitation: their dependence on random mutation selection, where operators are chosen uniformly at random from predefined lists or through fixed distribution strategies [54]. While such approaches introduce variability and explore broader input spaces, they often lead to suboptimal outcomes due to indiscriminate allocation of mutation attempts [34]. Valuable fuzzing iterations are frequently wasted on low-impact mutation operators that contribute little to discovering new execution paths or triggering vulnerabilities.

Lyu et al. [34] demonstrated significant disparities in the effectiveness of mutation operators. For instance, during AFL's deterministic stage, 49% of valid inputs in avconv originated solely from the bitflip operator, while the contributions of many others were negligible. However, random or fixed scheduling allocates mutation attempts evenly across the entire operator pool, ignoring their actual effectiveness. This uniform allocation creates three critical inefficiencies: (1) generation of redundant or near-duplicate test cases that provide no additional coverage; (2) underutilization of high-impact operators that could accelerate vulnerability discovery; and (3) delayed or missed identification of critical vulnerabilities because effective mutators are selected less frequently.

Consequently, random mutation scheduling results in poor resource utilization and extended time to vulnerability discovery, particularly under constrained time budgets.

2.2 Lack of Runtime Feedback Integration

Another critical limitation is the lack of runtime feedback integration in most smart contract fuzzers. Prior work [34] has demonstrated that feedback-driven mutation strategies, where coverage gains or vulnerability-triggering behavior influence operator selection, can improve fuzzing effectiveness.

Existing tools typically fail to leverage dynamic execution information (e.g., code coverage, opcode frequency, function call sequences, triggered oracles, and so on) to guide mutation scheduling. Instead, mutations are applied with static probabilities, regardless of how contracts respond to different inputs. This omission severely limits the fuzzer's ability to adapt to each contract's unique logic and control flow. Without feedback mechanisms, fuzzers cannot differentiate between high-impact and low-impact test cases, leading to inefficient input generation, inadequate state space exploration, and missed opportunities to discover edge-case vulnerabilities.

2.3 Limitations of Existing Adaptive Scheduling

While adaptive fuzzers like MOPT [34] and Classfuzz [15] optimize mutation schedules, their reliance on global strategies is inherently ill-suited for the state-dependent nature of smart contracts.

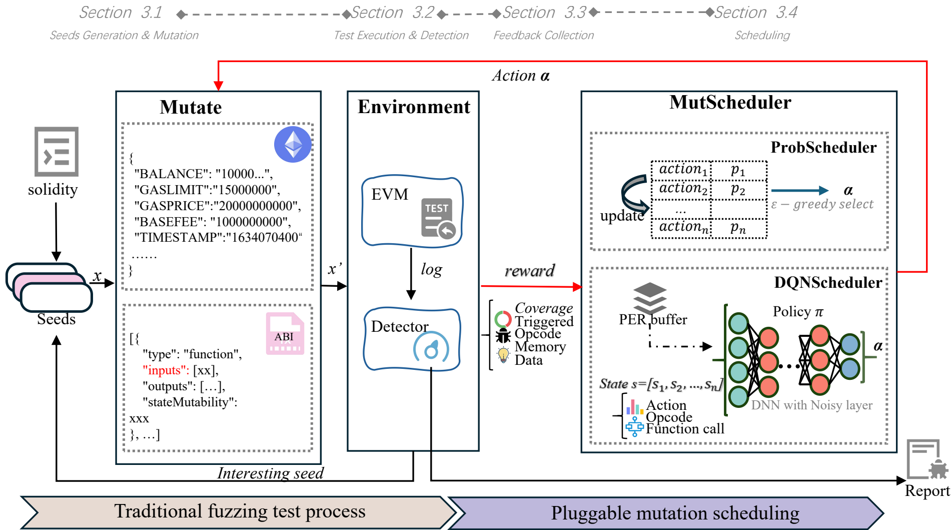

Fig. 1. Overview of FuzzMaster. MutScheduler selects actions based on execution feedback, with the environment returning state and reward signals for continuous learning.

Specifically, these tools optimize a single global probability distribution, applying an "average" strategy that fails to accommodate the context-aware mutation requirements of different functions. For instance, `transfer` logic may necessitate *integer overflow insertion*, whereas `ownership` checks often require *address replacement*. Furthermore, current tools lack fine-grained state feedback. By primarily focusing on shallow metrics like code coverage and treating the EVM as a black box, they fail to correlate persistent storage states with mutation effectiveness, a critical shortcoming given that smart contract vulnerabilities frequently depend on complex state transitions [26]. To overcome these limitations, we propose a DRL-based approach that dynamically maps the *current execution state* to the optimal mutation operator.

3 Approach

Fig. 1 illustrates FuzzMaster, an adaptive framework that optimizes mutation scheduling via real-time feedback. It comprises two synergistic components:

- **Fuzzing Engine**: Executes inputs within the EVM to capture runtime signals, including code coverage and opcode statistics.
- **MutScheduler**: A pluggable decision-making module that leverages these signals to dynamically refine mutation strategies, establishing a continuous learning loop for targeted vulnerability discovery.

3.1 Seeds Generation and Mutation

3.1.1 Seed Pool Initialization. FuzzMaster begins by generating a diverse initial pool of seed test cases, either randomly or derived from existing contract inputs. We leverage the contract's ABI to extract function signatures and parameter types, enabling the creation of syntactically correct inputs that conform to the contract's expected behavior.

3.1.2 Static Analysis. Before input generation, we conduct bytecode analysis to extract ABI information and perform taint analysis. Specifically, following the methodology of ItyFuzz [46], we parse the EVM bytecode to construct a CFG. We then apply data flow taint analysis to identify critical state-impacting paths. This deep analysis tracks how inputs influence state variables, allowing

Table 1. Test oracles for vulnerabilities.

Vulnerability	Description
Reentrancy	Allows to repeatedly call a function before state changes.
Self-Destruct	Attackers can remove the contract and transfer remaining funds.
Overflow	Improper arithmetic causes values to overflow or underflow.
Unchecked Call	Fails to verify external call success.
Denial of Service	Attackers can block or delay contract operations.
Access Control	Unauthorized users can access restricted functions.
Price Manipulation	Manipulable external data skews transaction outcomes.
Flashloan Exploitation	Attackers exploit short-term price or liquidity changes.

us to generate high-quality initial seeds that effectively target potential vulnerabilities such as uninitialized variables or overflow conditions.

3.1.3 Targeted Input Construction. To ensure generated inputs remain realistic and suitable for testing the contract's behavior, we generate type-specific candidates:

- *Fixed-size inputs* (e.g., integers or fixed-size byte arrays): Select from predefined value sets.
- *Variable-size inputs* (e.g., byte strings or dynamic arrays): Generate random lengths with valid domain elements.
- *Address-type parameters*: Choose from pools of valid contract addresses with matching ABI functions.

3.1.4 Interactive Fuzzing with Scheduled Mutations. . At each fuzzing iteration, we randomly select a seed from the pool and apply a mutation operator chosen by MutScheduler. This approach ensures controlled test case evolution guided by learned operator effectiveness.

3.2 Test Execution and Vulnerability Detection

Mutated test cases execute within the EVM while we monitor for vulnerabilities using eight test oracles adapted from ItyFuzz [46].

Table 1 lists each test oracle with a brief description. These oracles analyze execution logs, including opcodes and state changes, to identify potential vulnerabilities during fuzzing. The detection process focuses on identifying common and critical vulnerabilities in smart contracts, such as reentrancy attacks, self-destruct operations, and integer overflows.

3.3 Feedback Collection

Effective mutation scheduling within MutScheduler requires comprehensive runtime feedback collection and analysis. This feedback comprises two components: (1) structured state representations reflecting the current fuzzing context, and (2) reward signals evaluating test case outcomes.

3.3.1 ***State Representation****.* As input to MutScheduler, the state space characterizes the current execution context beyond the internal storage state of the smart contract.

We construct a continuous state space represented as a real-valued vector that embeds both static semantic features and dynamic execution feedback. This design enables MutScheduler to learn expressive representations suitable for reinforcement learning. The state vector comprises three core features:

(1) Action Frequency. We track the historical selection frequency of each mutation operator. This feedback provides visibility into past decisions, allowing the scheduler to deprioritize overused, low-yield operators in favor of those that consistently produce valuable outcomes (e.g., increased coverage).

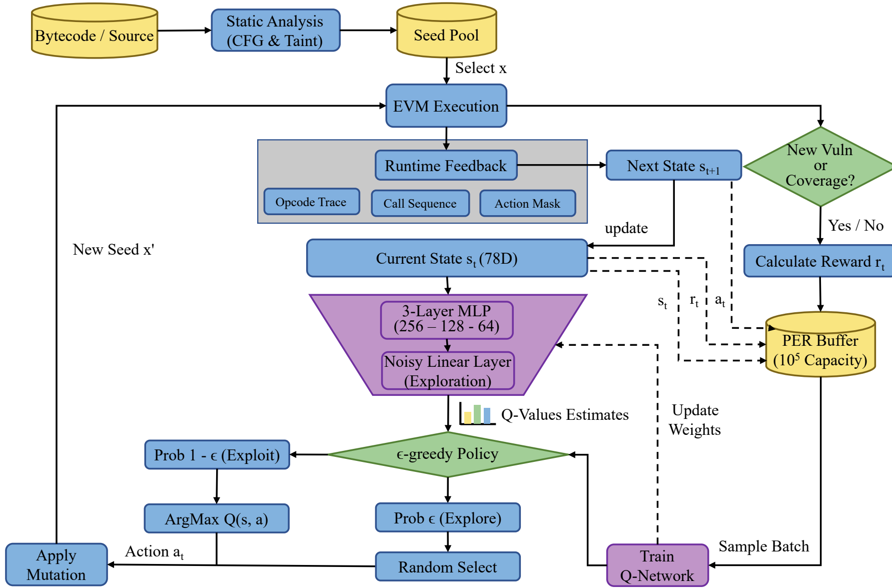

Fig. 2. Workflow of State Construction. Runtime execution feedback is encoded into a vector s_t used by the DQNScheduler for adaptive operator selection.

(2) Opcode Trace. We monitor the execution frequency of eight security-critical EVM opcodes: `sha3`, `call`, `create`, `selfdestruct`, `jump`, `jumpi`, `sload`, and `sstore`. These opcodes govern critical behaviors such as control-flow changes and state access. By analyzing their frequency, the scheduler identifies underexplored components and potential vulnerability hotspots (e.g., frequent `call` operations indicating reentrancy risks).
(3) Function Call Sequences. This feature encodes the temporal order of function calls triggered during execution. Since many vulnerabilities (e.g., Reentrancy, TOCTOU) depend on specific inter-function interactions, modeling these sequential dependencies enables the scheduler to identify critical behavioral patterns and target mutations toward state-dependent vulnerabilities.
Workflow of State Construction. As illustrated in Figure 2, the framework operates through a continuous feedback loop that transforms raw execution data into intelligent scheduling decisions:

- **Feedback Collection:** During the execution of a transaction sequence, the system captures real-time runtime signals. These include the *Opcode Trace* (e.g., the frequency of critical instructions like `SSTORE`) and the *Call Sequence* representing the order of invoked functions.
- **State Construction:** These raw signals are synthesized into a fixed-size, 78-dimensional state vector s_t. Specifically, the normalized opcode frequencies and the vectorized representation of the function sequence are concatenated to form the input tensor for the neural network.
- **Scheduling Step:** The composite vector s_t is fed into the `DQNScheduler`. The Q-network predicts Q-values for the action space, and based on the policy (e.g., ϵ-greedy), the scheduler selects the optimal mutation operator a_t (e.g., `ByteFlipMutator`) to generate the next seed.
- **Loop Update:** Upon execution of the mutated seed, if it triggers new state coverage or identifies a vulnerability, a positive reward r_t is calculated. The experience tuple (s_t, a_t, r_t, s_{t+1}) is then stored in the PER Buffer, which is subsequently sampled to update the Q-network weights, thereby refining the policy for future iterations.

*3.3.2 **Reward Mechanism**.* The optimization objective of MutScheduler is fundamentally driven by the **Vulnerability Reward (R_{vul})**, which provides a positive signal solely when a bug is detected.

Table 2. Description of different rewards.

Reward	Description
R_{cov}	Branch coverage and instruction coverage.
R_{vul}	1 if Detector identifies any vulnerabilities; otherwise 0.
$R_{\text{instr_int}}$	Instruction interesting. 1 if critical instructions appear in the execution trace (specifically targeting state changes via SSTORE and external interactions via CALLs); otherwise 0.
$R_{\text{instr_datawp}}$	Data path interesting. 1 if a store instruction writes to a memory location that was previously loaded, and the written value has a different abstraction from all previously stored values; otherwise 0.
$R_{\text{instr_comwp}}$	Comparison path interesting. 1 if the distance between operands of a comparison instruction is smaller than the minimum distance from previous executions; otherwise 0.

However, relying exclusively on this signal leads to the *sparse reward problem*, making it difficult for the agent to learn effectively in the absence of immediate bug discoveries.

To overcome this and guide the agent through the complex state space, we design a dense, multi-dimensional reward function detailed in Table 2. Beyond standard **Coverage Rewards (R_{cov})**, we introduce three state-aware mechanisms to incentivize deep exploration:

- **State & Interaction Feedback ($R_{\text{instr_int}}$):** Encourages the execution of critical opcodes (e.g., `SSTORE`, `CALL`), guiding the agent toward state-changing logic often missed by stateless fuzzers.
- **Data Diversity ($R_{\text{instr_datawp}}$):** Adopted from ItyFuzz [46], this rewards the generation of novel memory states, facilitating the discovery of data-flow-dependent vulnerabilities.
- **Constraint Solving ($R_{\text{instr_comwp}}$):** Rewards reductions in operand distance for comparison instructions, guiding the fuzzer to break through hard path constraints.

By synthesizing these signals, the reward mechanism ensures the agent explores not just broad code paths, but also specific, vulnerability-prone semantic states.

3.4 Scheduling

Efficient mutation scheduling is pivotal for discovering deep-seated vulnerabilities. In this section, we first define the abstract **Action Space** of mutation operators. Based on this foundation, we introduce two distinct scheduling strategies to iteratively refine operator selection: the lightweight **ProbScheduler** and the adaptive **DQNScheduler**.

3.4.1 ***Action Space Design.*** We formulate the mutation scheduling as a decision-making problem within a discrete action space. However, directly mapping the vast array of raw mutation operators to the action space results in high dimensionality, which hinders the convergence of the RL agent.

To mitigate this, we employ a **hierarchical abstraction** (as shown in Fig. 3). First, the MutScheduler selects the mutation target domain: either the EVM environment (ENV) or the ABI parameters.
(1) ENV Mutation. This targets context fields such as `BALANCE`, `GASPRICE`, `BASEFEE`, and `TIMESTAMP`. This allows the fuzzer to explore boundary cases under diverse environments. Simulating extreme values (e.g., fluctuating gas prices) helps reveal reentrancy risks and abnormal behaviors triggered by specific network conditions.
(2) ABI Parameter Mutation. Similarly, targeting ABI parameters applies to static, dynamic, or array-type inputs. Mutating static parameters targets arithmetic flaws (e.g., overflows), while perturbing dynamic types exposes memory safety errors (e.g., out-of-bounds access), ensuring robustness against logic exploits.

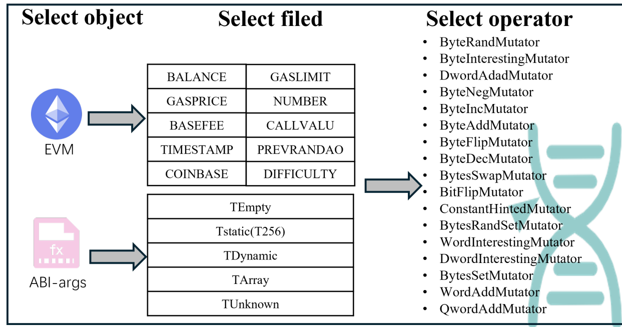

Fig. 3. Layered Mutation Process for Smart Contract Fuzzing.

(3) Operator Execution. Once the target domain and specific parameters are defined, a specific byte-level mutation operator—such as `ByteRandMutator`—is sampled for final execution.

3.4.2 ***ProbScheduler***. The probability table serves as a lightweight heuristic scheduler, using empirical probabilities to guide the selection of mutation operators. The table is dynamically updated based on the observed rewards associated with each mutation operator. For example, operators that frequently uncover vulnerabilities or improve code coverage are assigned higher probabilities. This mechanism ensures efficient scheduling in scenarios with limited computational resources or short testing durations.

At each step, the ProbScheduler employs an ϵ-greedy policy to balance exploration and exploitation. With a probability of $1-\epsilon$, the ProbScheduler selects the operator with the highest probability; otherwise, it explores other operators randomly.

3.4.3 ***DQNScheduler***. Based on the aforementioned continuous state space and discrete action space, we employ DQN [36] to construct the reinforcement learning framework for fuzzing.

Algorithm 1 details the complete training process of the tool, FuzzMaster. In each step, we select an action using an ϵ-greedy policy, where with a probability of ϵ, we randomly choose an action group and then randomly select a mutation operator within that group. After applying the selected mutation operator to the seed, the mutated seed is executed in the ENV, and we store the experience (s, a, r, s') for later experience replay and Q-network updates.

Additionally, we incorporate Noisy DQN to enhance exploration by injecting noise directly into the Q-network parameters. This enables the agent to aggressively test diverse mutations on unseen contract structures. Furthermore, we utilize Prioritized Experience Replay (PER) to prioritize transitions with high TD errors. By replaying unexpected outcomes more frequently, PER accelerates the learning process for critical, vulnerability-triggering conditions.

4 Experimental Design

4.1 Datasets

To conduct our experiments, we establish two datasets. Table 3 summarizes the two datasets. **D1** consists of 130 vulnerable contracts derived from VeriSmart-benchmarks [8] and SmartBugs [7], containing typical smart contract vulnerabilities widely adopted in numerous prior works [17, 21, 33, 38, 46, 47, 49, 50]. Notably, D1 includes contracts with small code size (minimum 6 LOC). We retain these "Gold Standards" in their original form as they are specifically designed to isolate core vulnerability semantics for precise ground-truth validation. D1 is primarily used for evaluating the efficiency and accuracy of the tools.

To evaluate the effectiveness of each tool on real-world contracts, we establish **D2**, from smart contracts deployed on the Ethereum mainnet. We collect the dataset from Etherscan [5], retrieving

Algorithm 1: Workflow of Training FuzzMaster

```
Input  : Iteration steps T, Sample size M, Search rate ϵ, Episode E, Smart Contract SC
Output : Network Q
1  Q ← InitializeNetwork();
2  s ← getInitState();
3  A, G ← groupFunctions(SC);
4  for k ← 1 to T do
5      a ← getRandomOrBestAction(A, Q, ϵ);
6      f(x) ← selectFunction(G[a]);
7      s', r ← executeFunction(f(x), SC);
8      saveExperience(s, a, r, s');
9      s ← s';
10     if k mod E = 0 then
11         {(s_i, a_i, r_i, s'_i)} ← loadExperience(M);
12         Q ← train({(s_i, a_i, r_i, s'_i)}, Q);
```

the most recent 10,000 smart contracts available as of November 2024. After removing duplicates, we compile and process these contracts using the specified compiler version associated with each contract. Contracts that fail to compile due to path configuration issues or missing keywords are excluded, resulting in a final dataset of 7,533 unique contracts.

Table 3. Statistics of Datasets.

ID	Source	# Num. of Contracts	# LoC Min	Max	Mean	Used For
D1	VeriSmart [7]	130	5	658	116	RQ1, RQ2
D2	Etherscan [5]	2,000	6	1,953	286	RQ3

4.2 Baselines

The available vulnerability detection tools can be classified into two categories: symbolic execution and fuzzing. For comparison, we select the vulnerability-guided symbolic executor SmartTest [47] following previous work, as it has been shown to outperform Mythril [37] and Oyent [33] in vulnerability detection. In the realm of fuzzing, We choose ConFuzzius, Smartian, and ityfuzz based on their superior performance as demonstrated in existing work [16, 26, 27, 38, 46, 50].

4.3 Evaluation Metrics

We adopt the following widely-used performance metrics [16, 46, 47, 49, 50] , to evaluate the effectiveness of FuzzMaster.

- **Precision**: This metric measures the proportion of reported vulnerabilities that are true positives, reflecting the reliability of the detection process.
- **Detection**: This metric calculates the percentage of ground-truth vulnerabilities successfully identified by the tool out of the total known vulnerabilities, indicating its capability to uncover security flaws.

- **Coverage:** This metric includes both instruction coverage, the percentage of executed instructions within the code, and branch coverage, the proportion of executed control flow branches.
- **Detection Time:** This metric records the amount of time taken to identify vulnerabilities within the smart contracts.

4.4 Implementation Details

4.4.1 Core Framework. We implemented FuzzMaster in Rust, leveraging LibAFL [22] as the fuzzing backbone and revm [10] as the EVM executor. We utilize revm's interpreter hooks to perform dynamic instrumentation, enabling efficient feedback collection. The architecture is modular, facilitating the integration of domain-specific feedback mechanisms to adapt to diverse blockchain ecosystems.

4.4.2 Scheduler Implementation. We explore two distinct algorithms:

- The ProbScheduler: This heuristic approach initializes a randomized probability table for mutation operators. During fuzzing, the table is iteratively updated based on execution feedback, employing a greedy strategy to prioritize high-yield operators.
- The DQNScheduler: Built on rustorch [6], this deep reinforcement learning approach is configured as follows:
  (1) Architecture: The input state s is a **78-dimensional vector** combining action masks, opcode traces, and function call embeddings. The Q-network utilizes a **3-layer MLP** (256-128-64 units) with ReLU activations. The final layer is a Noisy Linear Layer to enhance exploration stability.
  (2) Training Setup: We employ a **Prioritized Experience Replay (PER)** buffer (capacity 10^5). Training was conducted on the 130 contracts from Dataset D1 via 2-fold cross-validation (2 to 3 hours). The exploration rate ϵ decays linearly from 0.9 to 0.1 during training and is fixed at 0.1 during testing. To prevent local optima, the environment resets every 100 steps.
  (3) Hyperparameters: Detailed settings are listed in Table 4.

Table 4. Hyperparameter Settings for DQNScheduler.

Parameter	Value	Description
Input Dimension	78	Size of the concatenated state vector.
Hidden Layers	$256-128-64$	Units in the 3-layer MLP.
Replay Buffer	10^5	Capacity of the PER buffer.
Learning Rate	1×10^{-4}	Step size for the optimizer.
Discount Factor (γ)	0.99	Weight factor for future rewards.
Exploration (ϵ)	$0.9 \rightarrow 0.1$	Linear decay; fixed at 0.1 for testing.
Batch Size	64	Samples per training step.

4.4.3 Experimental Setup. To address the single-target limitations of baseline tools such as Smartian [16], we implemented an automated iterator that identifies and selects the correct `.bin` and `.abi` artifacts from the multiple files typically generated due to contract inheritance. Furthermore, we retained the specific Z3 solver [4] versions bundled with each respective baseline to ensure experimental stability and prevent compatibility errors. All experiments were executed on an Ubuntu server equipped with an Intel Xeon Gold 6226 CPU (24 cores, 2.70 GHz), an NVIDIA 2080Ti GPU, and 125 GB of RAM.

5 Experimental Results

In this section, we conduct extensive experiments to evaluate our FuzzMaster by addressing the following questions.

- **RQ-1 Performance of FuzzMaster.** Does FuzzMaster detect vulnerabilities more effectively and efficiently than baselines.
- **RQ-2 Impacts of Design.** How do individual design components (e.g., noise layer and PER module) affect FuzzMaster 's detection performance?
- **RQ-3 Performance on Real-world Contracts.** How does FuzzMaster perform on real-world smart contracts that were unseen during training?

Table 5. Vulnerabilities reported by different tools in D1. Detection is calculated based on all vulnerabilities, and precision is calculated based on the number of reported vulnerabilities.

	Tools	TP / Total	Detection(%)	Precision (%)
Symbolic	SmarTest [47]	30 / 31	23.08%	96.77%
Fuzzers	Confuzzius [50]	56 / 57	43.08%	98.25%
	Ityfuzz [46]	61 / 66	46.92%	92.42%
	FuzzMaster_ptable	70 / 70	53.85%	100.00%
	FuzzMaster_DQN	86 / 86	66.15%	100.00%

5.1 RQ1: Performance of FuzzMaster

Objective. This experiment aims to evaluate the efficiency and accuracy of FuzzMaster in detecting typical contract vulnerabilities compared to state-of-the-art baselines (SmarTest [47], Smartian [16], ConFuzzius [50], and Ityfuzz [46]) on dataset D1. We evaluate this by comparing detection, precision, and detection time.

Experimental Design. We enforced a strict 30-minute timeout per contract for all tools, repeating each experiment five times to mitigate randomness. For SmarTest, the Z3 solver timeout was set to 90 seconds. For FuzzMaster, we conducted 2-fold cross-validation to evaluate the model on unseen data. Specifically, the dataset D1 is split into two disjoint partitions for training and testing. Crucially, the results reported in Table 5 represent the **cumulative union** of unique vulnerabilities detected across both testing rounds. Since the partitions are mutually exclusive, this aggregation reflects the tool's performance on the entire D1 dataset (130 contracts), rather than on a single fold subset. Following automated detection, we verified the reported vulnerabilities against the ground-truth labels in D1 (Table 5) and visualized the detection trends over time in Fig. 4.

Results. We present and analyze the results from our comprehensive evaluation of FuzzMaster against state-of-the-art baselines on Dataset D1.

Comparison of FuzzMaster with Existing Symbolic Executors. FuzzMaster significantly outperforms SmarTest in identifying true vulnerabilities. Our log analysis indicates that SmarTest's reliance on the Z3 SMT solver creates significant bottlenecks. It severely suffers from path explosion and solver timeouts, particularly when handling complex arithmetic operations common in Dataset D1. Conversely, FuzzMaster avoids these solver bottlenecks by replacing constraint solving with DRL-guided mutation. The DQN agent learns to prioritize operators that drive state transitions, enabling the exploration of deep execution paths that are computationally inaccessible to symbolic executors.

Comparison of FuzzMaster with Existing Fuzzers. When compared to other fuzzers like Confuzzius, Ityfuzz, and FuzzMaster_ptable, FuzzMaster supported by DQN consistently achieves

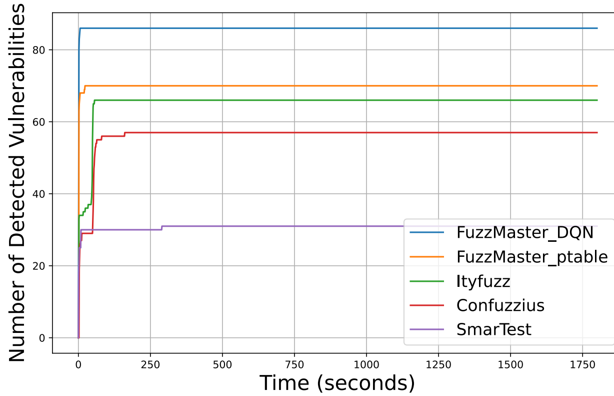

Fig. 4. True vulnerabilities being found versus time (in seconds) by different tools in D1.

superior results in both accuracy and efficiency. Notably, FuzzMaster demonstrates a steeper initial detection curve, identifying a majority of vulnerabilities within the first 1 seconds. This rapid detection is enabled by DQN's capability to dynamically prioritize high-risk mutation paths, ensuring targeted exploration of contract states. By leveraging historical data and contract-specific characteristics, the mutation operators guided by DQN make optimal decisions, avoiding redundant path explorations and focusing on key vulnerability-prone areas.

Given that other traditional tools utilize incompatible metrics (e.g., measuring Solidity code lines vs. basic blocks) or lack accessible coverage logs, we compared the coverage performance of FuzzMaster against the state-of-the-art baseline, ItyFuzz, on the D1 dataset. Specifically, our method achieves an average Instruction Coverage of 45.4% and Branch Coverage of 39.3%. In contrast, ItyFuzz reaches 41.2% and 34.5%, respectively. This corresponds to a relative improvement of approximately 10.2% in instruction coverage, confirming that the DRL-guided scheduling effectively penetrates deeper into complex contract logic.

Notably, even in cases where FuzzMaster yields lower code coverage than the baselines, it achieves accurate vulnerability detection with reduced time-to-detection. This efficiency is attributed to the DQN-assisted mutation mechanism, which prioritizes critical execution paths over redundant exploration. By learning to navigate directly toward vulnerability-prone states, the agent focuses computational resources on triggering key logic flaws rather than maximizing superficial metric coverage. This demonstrates that the DQN-based strategy effectively decouples detection speed from raw coverage, enhancing performance by targeting security-critical code regions.

Table 6. Detection Rates on D1 Stratified by Contract Size (Median LOC).

Tool	Small Contracts	Large Contracts	Overall
SmarTest [47]	28.50%	17.60%	23.08%
Confuzzius [50]	48.20%	37.90%	43.08%
ItyFuzz [46]	51.50%	42.30%	46.92%
FuzzMaster	**69.20%**	**63.10%**	**66.15%**

Performance on Different Contract Scales. To address concerns regarding whether the tool's performance is biased by simple contracts, we stratified the D1 dataset into "Small" and "Large" subsets based on the median LOC (50). Table 6 details the detection rates across these categories. The results confirm FuzzMaster's robustness across varying complexities. On **Small Contracts**,

FuzzMaster achieves a 69.20% detection rate, outperforming the nearest competitor ItyFuzz (51.50%) by a significant margin. Crucially, on **Large Contracts**, FuzzMaster maintains a high detection rate of 63.10%, whereas the performance of baselines drops more sharply (e.g., SmarTest drops to 17.60% and Confuzzius to 37.90%). This demonstrates that FuzzMaster's DRL-based scheduling effectively handles the increased state space of larger contracts, rather than just solving trivial cases.

Trade-off Analysis: DQN vs. Probabilistic Table. While FuzzMaster_DQN consistently outperforms FuzzMaster_ptable in detection count (Table 5), this gain entails a computational trade-off. As detailed in Section 6.2, the DQN component requires GPU resources and introduces a **15%–40%** runtime latency per iteration. In contrast, FuzzMaster_ptable is lightweight, requiring **no pre-training** and incurring negligible overhead (∼**5%**). Consequently, the choice depends on the scenario: FuzzMaster_DQN is optimal for deep scanning of high-value contracts where maximum effectiveness is paramount, whereas FuzzMaster_ptable is superior for rapid, resource-constrained environments like CI pipelines.

> ***Answer to RQ1***: *FuzzMaster is more efficient in detecting typical vulnerabilities than other state-of-the-art tools. Within 30 minutes, FuzzMaster detects 7%-43% more vulnerabilities than the compared tools in D1. In addition, FuzzMaster achieves 100% precision rate in D1.*

5.2 RQ2: Impacts of Design

Objective. Based on RQ1, we find that the DQNScheduler outperforms the ProbScheduler algorithm. Therefore, this study aims to explore how the noise layer and the PER module influence vulnerability detection performance within our reinforcement learning framework.

Experimental Design. We implement two tool variants: FuzzMaster $_{\text{noisy}}$, which includes the noise layer but excludes the PER module, and FuzzMaster $_{\text{PER}}$, which incorporates the PER module while using standard linear layers instead of the noise layer. The experimental details are the same as those in RQ1.

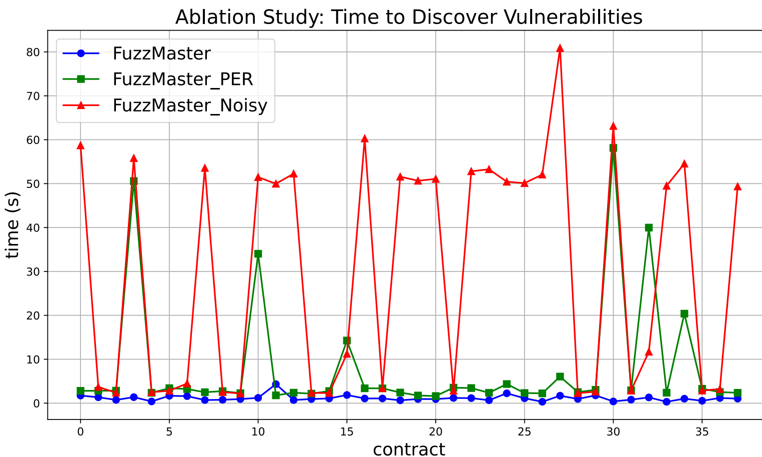

Fig. 5. The Time for Finding Vulnerabilities

Results. Fig. 5 illustrates the time overhead and coverage achieved by variants in detecting vulnerabilities. We then discuss the results from the time consumption and coverage aspects.

Time Consumption. As shown in Fig. 5, FuzzMaster noisy generally requires more time to detect vulnerabilities compared to FuzzMaster PER. For example, in the case of smart contract

number 27, the testing time for FuzzMaster noisy is 80 seconds, while FuzzMaster PER completes the test in just 5 seconds. This difference is attributed to the noise layer in FuzzMaster noisy, which introduces additional mutation operations, increasing the computational workload. In contrast, FuzzMaster PER leverages its optimized PER module to selectively perform mutations, reducing unnecessary exploration and significantly lowering the detection time.

We also observe greater variation in the time consumption of FuzzMaster noisy. For example, with smart contract number 33, the average testing time for FuzzMaster noisy across three trials is 50 seconds, but individual runs vary considerably: one takes 183 seconds, while another takes only 19 seconds. This fluctuation is primarily caused by the randomness introduced by the noise layer, which leads to varying levels of computational complexity depending on the mutations performed. In comparison, FuzzMaster $_{\text{PER}}$ exhibits more stable time consumption. This stability indicates that the PER module effectively manages the complexity of mutation operations, avoiding excessive or redundant computations.

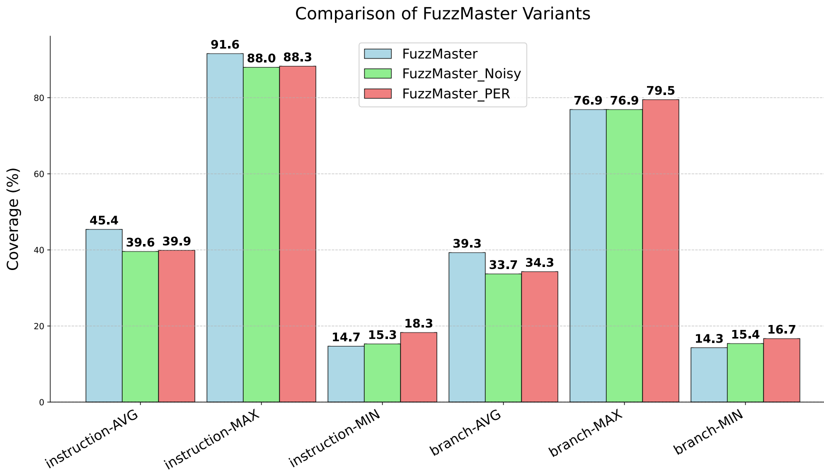

Fig. 6. Branch&Instruction Coverage for the Ablation Tool

Coverage. As shown in Fig. 6, FuzzMaster consistently outperforms both variants. Specifically, it improves instruction coverage by **5.6% and 9.8%**, and branch coverage by **4.7% and 8.9%** compared to FuzzMaster $_{\text{PER}}$ and FuzzMaster $_{\text{noisy}}$, respectively. These results confirm that the integrated adaptability and exploration mechanisms yield superior coverage.

In terms of maximum coverage (both instruction and branch), FuzzMaster achieves the highest values, reflecting its strong ability to explore program paths under optimal conditions. Interestingly, FuzzMaster has the lowest minimum coverage among the three configurations. This suggests that, while FuzzMaster excels in achieving high coverage, it may also be prone to neglecting some paths in less favorable scenarios. This variability highlights its potential for detecting rare or edge-case vulnerabilities, which might not be covered by the more stable but less exploratory approaches of FuzzMaster $_{\text{noisy}}$ and FuzzMaster $_{\text{PER}}$.

By contrast, FuzzMaster $_{\text{noisy}}$ and FuzzMaster $_{\text{PER}}$ appear to introduce certain biases or randomness in their coverage strategies—such as stochastic exploration in FuzzMaster $_{\text{noisy}}$ and an over-prioritization of specific experiences in FuzzMaster $_{\text{PER}}$—which may limit their overall effectiveness in achieving broad and consistent coverage.

Trade-off Between Coverage and Efficiency. While FuzzMaster generally achieves both higher coverage and faster detection, we observe specific cases where this balance shifts. For example, in

certain contracts, the coverage achieved by FuzzMaster is slightly lower than that of FuzzMaster $_{\text{noisy}}$, yet the time taken to detect vulnerabilities is significantly shorter. This finding suggests that the PER module effectively focuses on high-risk paths, reducing unnecessary exploration and enabling quicker identification of critical vulnerabilities.

> ***Answer to RQ2***: *The FuzzMaster tool, which integrates the noise layer and PER module, demonstrates superior performance in terms of both efficiency and more consistent coverage. The noise layer enhances exploration capabilities, while PER directs attention towards critical paths.*

5.3 RQ3: Performance on Real-world Contracts

Objective. To test whether FuzzMaster can effectively detect vulnerabilities in real-world smart contracts after being trained on benchmark data. This helps evaluate its generalization ability and practical usefulness.
Experimental Design. Because such contracts contain a low proportion of vulnerabilities, training a robust, vulnerability-guided neural model directly on them is difficult (i.e., a neural network). To address this issue, we first train the model on dataset D1 and then assess its performance on dataset D2. We run FuzzMaster, SmarTest [47], Ityfuzz [46], Confuzzius [50] on dataset D2, with each contract running for 60 minutes.

Table 7. Comparative performance on D2 (7,533 contracts). Precision is estimated via expert review.

Tool	Total Reports	Detection Rate	Precision (Verified)
FuzzMaster_DQN	**97**	**1.29%**	**65.00%**
ItyFuzz [46]	53	0.70%	55.00%
Confuzzius [50]	23	0.31%	25.00%
SmarTest [47]	0	0.00%	0.00%

Results and Validation. We present the results of FuzzMaster on the real-world dataset D2, validating its effectiveness through a rigorous multi-stage expert evaluation protocol, analyzing the economic impact of the confirmed vulnerabilities, and examining how contract complexity influences detection dynamics.

Human Evaluation. Since official confirmation from developers is often unavailable for in-the-wild contracts, we established a rigorous **multi-stage expert validation protocol** to strictly distinguish TP from FP, following industrial audit standards.

- **Expert Review Board:** We recruited three independent security experts from top-tier blockchain security firms, each with over 5 years of experience in EVM bytecode analysis and DeFi vulnerability research.
- **Cross-Validation Protocol:** To ensure objectivity, the 97 vulnerability reports were distributed via a blind cross-validation strategy. Each report was independently reviewed by two experts. They were tasked with labeling reports as TP or FP using a two-step technical verification: (1) *Static Reachability Check* using tools like Slither and Mythril; and (2) *Manual Trace Reconstruction*, where experts analyzed execution logs to confirm state exploitability.
- **Conflict Resolution:** Discrepancies were resolved through a consensus meeting, with the third expert acting as an arbitrator. A vulnerability was confirmed only if a consensus was reached or the exploit was strictly reproducible.

Based on this rigorous protocol, experts confirmed 63 True Positives out of 97 reports, yielding a Precision of 65.00%. Regarding the Detection Rate (the ratio of contracts with confirmed vulnerabilities to the total 7,533 contracts), FuzzMaster achieves 1.29%, nearly doubling that of the

nearest baseline (ItyFuzz at 0.70%). It is worth noting that the overall low detection magnitude across all tools reflects the reality that the majority of real-world contracts are inherently safe (e.g., standard ERC-20 tokens) and lack complex state logic. Despite this sparsity, FuzzMaster demonstrates superior capability in uncovering rare vulnerabilities.

Impact and Exploitability Analysis. To assess the economic impact, we categorized the 97 verified reports by severity [17]as shown in Table 8.

- **Critical:** Includes 15 Reentrancy and 15 Price Manipulation bugs. These pose the highest risk to DeFi protocols, enabling attackers to drain contract assets via recursive withdrawals or oracle manipulation.
- **High:** Comprises 26 Access Control and 25 Integer Overflow/Underflow issues. These vulnerabilities compromise contract integrity, allowing unauthorized ownership hijacking or infinite token minting, which fundamentally destroys the token's economic model.
- **Medium:** Involves 10 Arbitrary Memory Access and 6 DoS/Locking Ether bugs. While not always leading to direct theft, they cause significant operational damage by permanently freezing funds or blocking legitimate transactions.

Overall, FuzzMaster demonstrates the capability to detect high-stakes threats, with over **83%** of the verified cases classified as High or Critical.

Table 8. Distribution and Economic Impact of Verified Vulnerabilities detected by FuzzMaster.

Vulnerability Category	Count	Impact Level
Access Control	26	High
Integer Overflow/Underflow	25	High
Reentrancy	15	Critical
Price Manipulation	15	Critical
Arbitrary Memory Access	10	Medium-High
Locking Ether / DoS	6	Medium
Total Verified	**97**	–

Complexity and Detection Dynamics. We further analyze the temporal distribution of detected vulnerabilities to understand the impact of contract complexity. First, regarding the standard benchmark D1 (Fig. 4), detection counts saturate rapidly. This "early ceiling" reflects the limited scale of these reference contracts, which are widely used in the community [16, 21, 33, 38, 46, 47, 49, 50] to validate baseline correctness. In stark contrast, the real-world dataset D2 features significantly higher complexity, with larger LOC and intricate inter-function logic. As observed in our analysis of D2, the vulnerability discovery curve does not saturate early but exhibits a continuous rise throughout the 60-minute duration, steadily increasing to reach a total of 97 confirmed vulnerabilities. This temporal difference indicates that real-world contracts present a deeper state space, where FuzzMaster's DRL-based scheduling is essential for sustaining long-term exploration and uncovering latent vulnerabilities that reside deep within the execution paths.

> ***Answer to RQ3:*** *Trained on benchmark D1, FuzzMaster effectively generalizes to real-world contracts (D2). It detects significantly more verified vulnerabilities than state-of-the-art baselines, covering high-impact categories like Reentrancy and Price Manipulation with a precision of 65%.*

6 Discussion

6.1 Portability and Integration

To further validate the modularity and portable of our proposed approach,. We conducted an additional experiment by integrating the **MutScheduler** component into ItyFuzz [46]. Specifically, we replaced ItyFuzz's default random mutation strategy with our DQNScheduler, while preserving its underlying snapshot-based execution engine and oracle detectors. We evaluated this enhanced version (ItyFuzz+MutScheduler) on the benchmark dataset D1 under the same experimental conditions as RQ1. The results demonstrate a significant improvement: the integration increased ItyFuzz's detection rate from **46.92% to 58.5%**. This 11.6% absolute improvement indicates that the intelligent scheduling provided by MutScheduler is not tightly coupled to our specific implementation but acts as a transferable optimization module that can be "plugged in" to enhance existing frameworks.

6.2 Runtime Overhead Analysis

To quantify the computational cost introduced by DRL-based scheduling, we measured the execution time latency and memory consumption of FuzzMaster compared to the baseline ItyFuzz. We recorded the average time consumed per fuzzing iteration across the D1 dataset.

- **Time Latency.** As summarized in Table 9, the lightweight baseline ItyFuzz achieves an average speed of 18.5 ms per iteration. In contrast, FuzzMaster_DQN records an average of 4.1 ms per iteration. This represents an absolute latency increase of 5.6 ms, resulting in a relative overhead of approximately 30.2%. This delay is primarily attributed to the feature extraction process and the forward pass of the DQN model during mutation operator selection.
- **Memory Consumption.** Regarding resource usage, FuzzMaster_DQN utilizes an additional 100–300 MB of GPU memory for storing the neural network and replay buffer, which is negligible for modern hardware.
- **Cost-Benefit Analysis.** The overhead of DRL is justified by the substantial gain in detection quality. Since minimizing false negatives is paramount in security auditing, FuzzMaster's superior detection rate compared to ItyFuzz validates this trade-off as acceptable and cost-effective.

Table 9. Runtime Overhead: Average Time per Iteration and Memory Usage.

Method	Time (s)	Overhead (%)	GPU Mem (MB)
ItyFuzz (Baseline)	18.5 ms	-	-
FuzzMaster_ptable	19.2 ms	+3.8%	-
FuzzMaster_DQN	24.1 ms	+30.2%	100–300 MB

6.3 Impact of Training Data Size

To understand how the quantity of training data influences the effectiveness of FuzzMaster, we conducted a sensitivity analysis. We trained the DQN agent using randomly sampled subsets of the D1 training partition at ratios of 20%, 50%, 80%, and 100%, while keeping the testing set constant. Table 10 presents the detection rates across these training volumes. The results reveal two significant observations:

- **Robustness to Data Scarcity:** Remarkably, even when trained on only **20%** of the data, FuzzMaster achieves a detection rate of **55.38%**. This performance is still superior to the state-of-the-art baseline ItyFuzz (46.92%), indicating that the DRL agent can efficiently extract core vulnerability patterns and learn effective mutation strategies from a limited number of samples.
- **Scalability:** As the training data volume increases from 20% to 100%, the detection rate improves consistently from 55.38% to 66.15%. This positive correlation suggests that providing the model

with a more diverse set of contract logic and vulnerability scenarios further refines its ability to navigate complex state spaces.

Table 10. Impact of Training Data Volume on Detection Rate.

Training Data Ratio	Detection Rate
20%	55.38%
50%	62.31%
80%	64.62%
100% (Full D1)	66.15%

6.4 Threats to Validity

6.4.1 Internal Validity: Scheduling Instability and Algorithmic Bias. A core threat to internal validity stems from the stochastic nature of our mutation scheduling mechanisms. First, regarding the DRL-based guidance, although it provides adaptive operator selection, the learning process is susceptible to instability caused by environmental fluctuations (e.g., gas limits, EVM semantics). Since the policy updates are based on prior execution outcomes, there is a risk of the agent converging to local optima—overfitting to specific mutation operators while neglecting others—thereby reducing state space coverage. Second, the Probabilistic Table (ProbScheduler) relies on predefined priors. If the initial probability distribution does not align with the target contract's specific logic, the scheduler may suffer from reduced adaptability, leading to redundant fuzzing cycles.

To mitigate these instability and bias issues, we have employed mechanisms such as periodic policy resets and entropy regularization to penalize overconfident decisions. Future work could further enhance robustness by incorporating multi-strategy learning frameworks, ensuring consistent exploration efficiency across diverse execution environments.

6.4.2 External Validity: Generalization and Data Dependence. A significant threat to external validity lies in the risk of **structural overfitting** to the training dataset. Since FuzzMaster employs a learning-based approach, its policy is inevitably influenced by the vulnerability distribution and coding patterns present in the training set (D1). Consequently, the model's effectiveness might fluctuate when encountering novel vulnerability types or distinct architectural styles that are underrepresented in the training phase.

To address this, we designed a **hybrid reward mechanism** (Section 3.3.2) that mitigates the risk of "memorizing" specific exploits. By integrating generic exploration incentives—such as code coverage (R_{cov}) and persistent state modifications (R_{instr_int})—the agent is encouraged to learn generalizable state-exploration strategies (e.g., navigating complex control flows) rather than solely fitting to specific bug traces.

Looking ahead, we aim to further bolster the tool's generalization capabilities by expanding the training dataset to include contracts from diverse domains and blockchain platforms. Additionally, we plan to apply domain randomization and advanced regularization techniques (e.g., dropout) to ensure the model remains resilient to unseen contract variations.

7 Related Work

7.1 Smart Contract Fuzzing

Fuzz testing plays a critical role in detecting vulnerabilities in smart contracts. ContractFuzzer [27] was the first fuzzing framework designed specifically for this purpose. It combines static and dynamic analysis techniques to collect contract execution data via the EVM, extracts function signatures from the ABI, and generates test cases based on predefined vulnerability patterns. Building on

ContractFuzzer, SmartGift [59] enhances vulnerability detection by replacing the original test input generation method with NLP, improving its overall performance. sFuzz [38] further refines ContractFuzzer's approach by enhancing seed mutation strategies, incorporating AFL fuzzing methods, and introducing a lightweight, multi-objective adaptive strategy. This helps focus on hard-to-cover branches and identifies more security vulnerabilities. xFuzz [56], an extension of sFuzz, addresses challenges in cross-smart contract scenarios. By using machine learning and adaptive metrics, it intelligently selects functions to execute, reducing unnecessary transaction sequence combinations. Smartian [16] considers the effect of transaction sequences on code coverage, generating sequences based on predefined usage relationships and retaining test cases that cover broader data flows. This enables faster and more comprehensive contract code coverage. ItyFuzz [46] introduces an on-the-fly fuzzing approach to efficiently detect real vulnerabilities in blockchain contracts. Unlike other tools, ItyFuzz stores states and transactions using a snapshot-based method, rather than altering the transaction sequence. It leverages data flow and comparison waypoints to classify and save interesting states, allowing for more effective program exploration.

7.2 Static Analysis and Deep Learning-based Detection

Beyond fuzzing, smart contract security has been extensively studied using static analysis and deep learning techniques.

- **Static Analysis.** Tools like Slither [21] and Oyente [33] analyze code structure or control flow graphs (CFGs) without actual execution. While they offer rapid detection speeds, they typically suffer from high false-positive rates due to the over-approximation of execution states and the inability to accurately model complex inter-contract calls [39].
- **Deep Learning (DL) Approaches.** Recently, DL-based methods [41, 61] have proposed treating vulnerability detection as a classification task. These approaches employ models like GNNs or LSTMs to learn vulnerability patterns from code representations. While promising, they face two primary limitations: (1) heavy reliance on high-quality labeled datasets, which are often scarce in the smart contract domain; and (2) the inability to generate concrete exploit inputs to verify the flagged vulnerabilities.

In contrast to these approaches, FuzzMaster focuses on *dynamic verification*. Unlike static analysis or DL classifiers which indicate *potential* issues, FuzzMaster generates executable transaction sequences that serve as Proof-of-Exploit, ensuring zero false positives.

7.3 Deep Reinforcement Learning

Reinforcement Learning (RL) enables agents to learn optimal behaviors through environmental interaction, mathematically modeled as a Markov Decision Process (MDP). At each step, an agent in state s executes action a, transitions to s', and receives a reward r. To handle the exponential state spaces characteristic of complex systems like smart contracts, Deep Reinforcement Learning (DRL) integrates DNNs to approximate value functions effectively.

Recent studies have successfully combined RL with blockchain technologies. For instance, Dai et al. [18] utilized Deep RL for caching-resource optimization in 5G networks, while Chen et al. [13] applied it to supply chain decision-making. In the financial sector, Schnaubelt [43] designed specialized environments for optimizing cryptocurrency execution strategies.

In the fuzzing domain, frameworks like Fuzzergym [19] and Deep Reinforcement Fuzzing [11] have applied DQN to binary analysis. However, these approaches primarily target generic, stateless programs (e.g., file parsers), focusing on code coverage rather than the persistent state transitions inherent in smart contracts. While RLF [49] targets smart contracts, it often employs simplified state representations that neglect storage dynamics. **In contrast, FuzzMaster explicitly models**

the EVM execution state (including persistent storage and function selectors). By utilizing a state-aware reward mechanism (e.g., incentivizing `SSTORE`), FuzzMaster learns context-specific strategies to uncover deep vulnerabilities that generic RL fuzzers miss.

7.4 Active Learning

Active Learning (AL) represents a paradigm shift from standard supervised learning by allowing the algorithm to query an oracle (e.g., program execution) only on the most informative samples [44]. In the broader Software Engineering domain, AL has been applied to tasks like Defect Prediction and Test Suite Refinement. For instance, Bowring et al. [12] employed AL to optimize the classification of test case behaviors. In defect prediction, ACoForest [31] and TAL [45] utilize active sampling to reduce labeling costs while maintaining high prediction accuracy. However, the application of AL in *Smart Contract Fuzzing* remains relatively underexplored compared to these fields. Unlike RL, which focuses on exploitation (maximizing rewards), AL emphasizes exploration (reducing uncertainty). We identify this as a critical gap: while widely used for static prediction, AL's query-efficiency principles offer a promising, yet largely untapped, complement to dynamic fuzzing approaches, potentially addressing the computational bottlenecks of heavy-weight DRL models.

8 Conclusion

In this work, we present a Mutation Scheduling fuzzer based on reinforcement learning and probability table, namely FuzzMaster, for effectively selecting mutation operators in smart contracts. In particular, we firstly model the process of fuzzing smart contracts as MDP to construct our reinforcement learning framework. Then, we design a new appropriate reward with consideration of both vulnerability and code coverage so as to guide FuzzMaster to generate specific transaction sequences for revealing vulnerabilities, especially for the complicated vulnerabilities related to multiple functions. Finally, we design a neural network for FuzzMaster to automatically learn from the previous sequences, so that our FuzzMaster can effectively generate vulnerable transaction sequences. Extensive experimental results demonstrate that the proposed FuzzMaster detects more vulnerabilities than other state-of-the-art tools within a limited time. Our results also validate the effectiveness of the designed reward contributing to the superior performance of FuzzMaster.

Furthermore, to address the computational overhead of Deep Reinforcement Learning, we plan to integrate AL techniques in future iterations. We distinguish that while our current RL framework is reward-driven(optimizing policies based on feedback), AL is uncertainty-driven and sample-efficient. To leverage this, we will employ *Uncertainty Sampling* [44] to prioritize mutation candidates where the Q-network exhibits high *epistemic uncertainty* (i.e., low confidence). This hybrid strategy specifically aims to overcome inefficient early exploration by filtering ineffective mutations, thereby accelerating convergence in the sparse reward environments characteristic of complex smart contracts.

9 Data Availability

The replication of this paper is publicly available [8].

Acknowledgements

This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198 (SD2)), the National Natural Science Foundation of China (No.62572429), Ningbo Global in Innovation Center, Zhejiang University, the Fundamental Research Funds for the Central Universities (No.226-202200064), Zhejiang Provincial Natural Science Foundation of China (No.LY24F020008), the Key Research and Development Program of Zhejiang Province (No.2021C01105), the State Street Zhejiang University Technology Center.

References

[1] 2016. Ethereum white paper: a next generation smart contract & decentralized application platform. [Online]. https://github.com/ethereum/wiki. Accessed: 2024-11-29.
[2] 2018. Eos.io technical white paper. [Online]. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md. Accessed: 2024-11-29.
[3] 2018. "Ethereum smart contract best practices,". https://consensys.github.io/smart-contract-best-practices/known_attacks/. Accessed: 2024-11-29.
[4] 2023. Z3Prover. https://github.com/Z3Prover/z3. Accessed: 2024-11-29.
[5] 2024. Etherscan. http://etherscan.io. Accessed: 2024-11-29.
[6] 2024. rustortch. https://github.com/LaurentMazare/tch-rs. Accessed: 2024-11-29.
[7] 2024. VeriSmart-benchmarks. https://github.com/kupl/VeriSmart-benchmarks. Accessed: 2024-11-29.
[8] 2026. Replication. https://github.com/vinci-grape/FuzzMaster
[9] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 427–430.
[10] bluealloy. 2022. revm - Rust Ethereum Virtual Machinebluealloy. https://github.com/bluealloy/revm. Accessed: 2024-11-29.
[11] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 116–122.
[12] James F Bowring, James M Rehg, and Mary Jean Harrold. 2004. Active learning for automatic classification of software behavior. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 195–205.
[13] Huilin Chen, Zheyi Chen, Feiting Lin, and Peifen Zhuang. 2021. Effective management for blockchain-based agri-food supply chains using deep reinforcement learning. *IEeE Access* 9 (2021), 36008–36018.
[14] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts.. In *NDSS*.
[15] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49, 4 (1995), 327–335.
[16] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.
[17] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. 2023. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology* 159 (2023), 107221.
[18] Yueyue Dai, Du Xu, Sabita Maharjan, Zhuang Chen, Qian He, and Yan Zhang. 2019. Blockchain and deep reinforcement learning empowered intelligent 5G beyond. *IEEE network* 33, 3 (2019), 10–17.
[19] William Drozd and Michael D Wagner. 2018. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490* (2018).
[20] Bernhard Mueller et al. 2023. Consensys mythril. Website. https://github.com/Consensys/mythril
[21] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
[22] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065.
[23] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1110–1114.
[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.
[25] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns* 10, 8 (2010), 239.
[26] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 531–548.
[27] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.
[28] Keith Joiner. 2024. Review of Fuzz Testing to Find System Vulnerabilities. *The ITEA Journal of Test and Evaluation* 45, 4 (2024).

[29] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.
[30] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1 (2018), 1–13.
[31] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. 2012. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering* 19, 2 (2012), 201–230.
[32] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.
[33] 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*. 254–269.
[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.
[35] Zalewski M. 2015. American fuzzy lop. Website. http://lcamtuf.coredump.cx/afl/
[36] Volodymyr Mnih. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[37] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9, 54 (2018), 4–17.
[38] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.
[39] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. 1325–1341.
[40] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.
[41] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE access* 8 (2020), 19685–19695.
[42] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).
[43] Matthias Schnaubelt. 2022. Deep reinforcement learning for the optimal placement of cryptocurrency limit orders. *European Journal of Operational Research* 296, 3 (2022), 993–1006.
[44] Burr Settles. 2009. Active learning literature survey. (2009).
[45] Victor S. Sheng, Bin Gu, Wei Fang, and Jian Wu. 2014. Cost-sensitive learning for defect escalation. *Knowledge-Based Systems* 66 (2014), 146–155. doi:10.1016/j.knosys.2014.04.033
[46] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.
[47] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.
[48] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*.
[49] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[50] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.
[51] 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*. 664–676.
[52] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82.
[53] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2012. SECFUZZ: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 1–7.
[54] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[55] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 789–800.

[56] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).
[57] Chi Zhang, Yu Wang, and Linzhang Wang. 2020. Firmware fuzzing: The state of the art. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 110–115.
[58] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1099–1114.
[59] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. SmartGift: Learning to generate practical inputs for testing smart contracts. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 23–34.
[60] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.
[61] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence*. 3283–3290.

Received 2026-02-23; accepted 2026-03-24