Blockchain, Web3 & Digital Trust Journal

ISSN: e-ISSN: 3117-5988 / p-ISSN: 3117-597X

https://galaxiauniverse.com/index.php/BWDTJ/issue/archive

AI-Assisted Smart Contract Testing and Validation Frameworks

Lukas Rossi[1]

[1] Postdoctoral Researcher, Department of Machine Learning, Advanced Computing University, Paris, France. Email: lukas.rossi188@gmail.com | ORCID: 1179-3531-0846-3048

ABSTRACT

Smart contract testing today is a manual craft. Developers write unit tests by hand, choose which edge cases to cover based on experience and intuition, and hope they have not missed the one input sequence that drains the protocol. Coverage tools tell them which lines executed but not which attack scenarios were explored. Fuzzers generate random inputs but without understanding of the contract's economic logic. AI-assisted testing changes this equation by using machine learning to generate test cases that target high-risk code regions, to synthesise property specifications from contract behaviour, and to predict which untested paths are most likely to harbour bugs. I present the AI-Assisted Smart Contract Testing Framework (AASCTF), evaluating four AI-assisted testing strategies -- LLM-generated test suites, reinforcement-learning-guided fuzzing, neural property synthesis, and coverage-predictive test prioritisation -- on 60 real DeFi contracts with known vulnerability status. The Testing Effectiveness Score (TES) measures branch coverage, vulnerability detection rate, test generation speed, and developer adoption friction. Key finding: RL-guided fuzzing achieves the highest TES (0.912) by learning which input sequences reach deep contract states that random fuzzing rarely explores, increasing branch coverage from 62.4% (random Echidna) to 89.8% and vulnerability detection from 68.4% to 91.2%.

Keywords: AI-assisted testing; smart contract validation; reinforcement learning fuzzing; LLM test generation; property synthesis; test coverage; DeFi testing; automated testing

Citation: Rossi [2025]. AI-Assisted Smart Contract Testing and Validation Frameworks. DOI: https://doi.org/10.5281/zenodo.19188780

Article Information: *Received:* 02 March 2025 *Accepted:* 28 April 2025 *Published:* 28 June 2025

Research Article: *Research Article*

ISSN: e-ISSN: 3117-5988 / p-ISSN: 3117-597X Vol: 2025, Issues: 2, 2025

1. Introduction

1.1 The Testing Gap in Smart Contract Development

Testing smart contracts is harder than testing ordinary software for three reasons that compound into a coverage crisis. First, the input space is adversarial: contracts must be correct not just for expected inputs but for inputs crafted by an intelligent attacker who reads the source code and understands the economic incentives. A transfer function must handle not just normal transfers but also reentrancy callbacks, flash loan-funded mega transfers, and sandwich attacks on pending transactions. Second, state space explosion: a DeFi lending protocol's behaviour depends on every user's collateral position, every oracle price, and every pending liquidation -- a combinatorial space that no manual test suite can meaningfully sample. Third, multi-transaction dependencies: the most dangerous bugs manifest only through specific sequences of transactions, not through any single call in isolation. The industry standard -- Foundry or Hardhat unit tests plus Echidna or Medusa fuzzing -- achieves typical branch coverage of 55-65% on complex DeFi contracts. That leaves a third or more of the code untested. AI-assisted testing promises to close this gap by generating smarter test inputs, predicting where bugs hide, and synthesising the property specifications that fuzzers need but developers rarely write.

1.2 Four Strategies I Evaluated

LLM-generated test suites use GPT-4-class models to read contract source code and generate complete test files -- unit tests, integration scenarios, and adversarial attack simulations. RL-guided fuzzing replaces Echidna's random input generation with a reinforcement learning agent that learns to select function call sequences and argument values that maximise coverage and property violations. Neural property synthesis uses a fine-tuned model to infer likely contract invariants from code, giving fuzzers meaningful properties to check without manual specification. Coverage-predictive prioritisation trains a classifier to predict which untested branches are most likely to contain bugs, directing testing effort where it matters most. Section 2 details each strategy. Section 3 presents AASCTF. Results in Section 4, discussion in Section 5, conclusions in Section 6.

2. Four AI-Assisted Testing Strategies

2.1 LLM-Generated Tests and RL-Guided Fuzzing

LLM test generation works by feeding the contract source to a prompted language model and asking it to produce a complete test suite. I used a two-stage prompt: stage one asks the LLM to analyse the contract and identify high-risk functions, access control boundaries, and state-changing operations; stage two asks it to generate Foundry-compatible tests targeting each identified risk. The results are surprisingly usable -- about 78% of generated tests compile and execute on the first attempt, and another 12% need only minor fixes (import paths, address

literals). The LLM generates tests that a human developer would write -- but faster, and more of them. It does not, however, generate the creative adversarial scenarios that expert auditors produce. A flash loan attack test requires understanding of cross-protocol economics that the LLM approximates but does not deeply reason about. RL-guided fuzzing is the strategy I am most excited about. Standard fuzzers like Echidna generate random function call sequences and random argument values. They explore the state space breadth-first, covering shallow states well but rarely reaching the deep states where interesting bugs live -- the state where a lending pool is 99.8% utilised and a specific oracle price triggers a liquidation cascade. My RL agent treats the contract as an environment, function calls as actions, and branch coverage plus property violation as reward. It learns, over thousands of episodes, which call sequences reach unexplored states -- and then exploits that knowledge to systematically explore the deep corners of the state space that random fuzzing would need millions of iterations to reach.

2.2 Neural Property Synthesis and Coverage-Predictive Prioritisation

The weakest link in property-based testing is the properties themselves. Echidna checks whatever invariants the developer writes -- but writing good invariants requires understanding what should never happen in the contract, which requires the same expertise as writing the contract itself. Neural property synthesis addresses this by training a model on a corpus of contract-invariant pairs (drawn from OpenZeppelin test suites, Certora specifications, and audit reports) and then predicting likely invariants for new contracts. I fine-tuned CodeT5 on 2,400 contract-invariant pairs. The model generates invariants like "totalSupply() should equal the sum of all balances" and "only the owner should be able to call pause()" -- obvious in retrospect, but exactly the kind of properties that developers forget to write because they seem too obvious to test. Coverage-predictive prioritisation does not generate tests; it decides which tests to run first. A gradient-boosted classifier trained on historical bug-branch associations predicts which uncovered branches are most likely to contain vulnerabilities. Features include: distance from external call, presence of state-modifying operations after external calls, arithmetic operations on user-controlled values, and proximity to access-control modifiers. The classifier directs fuzzing effort -- instead of covering branches uniformly, the fuzzer spends more time on the branches the classifier flags as high-risk. Table 1 compares all four strategies.

Table 1: Four AI-Assisted Testing Strategies -- Approach, Strength, Limitation

Strategy	AI Component	What It Produces	Primary Strength	Primary Limitation
LLM-Generated Tests	GPT-4 class (prompted)	Complete Foundry test files	Speed (minutes vs. hours)	Shallow adversarial reasoning

Strategy	AI Component	What It Produces	Primary Strength	Primary Limitation
RL-Guided Fuzzing	PPO agent	Guided function call sequences	Deep state exploration	Training time (4-8 hours)
Neural Property Synth.	Fine-tuned CodeT5	Invariant specifications	Automates spec writing	Maymiss domain-specific invariants
Coverage-Predictive	XGBoost classifier	Branch risk rankings	Directs effort efficiently	Depends on historical data quality

3. AASCTF Evaluation Design

3.1 The 60-Contract Test Corpus

I assembled 60 real DeFi contracts -- 30 with known vulnerabilities (from audit reports and post-mortem analyses) and 30 without known vulnerabilities (deployed for over 18 months with significant TVL and no exploits). The vulnerable contracts span the SCFVAF taxonomy: reentrancy, flash loan manipulation, access control, oracle issues, arithmetic, and logic errors. Contract complexity ranges from 200 to 2,800 lines of Solidity, with a mean of 840 lines -- representative of production DeFi protocols. For each contract, I ran five testing configurations: baseline (Echidna random fuzzing, 30 minutes, default settings), LLM-generated tests (plus baseline fuzzing), RL-guided fuzzing (30 minutes, pre-trained agent), neural properties (synthesised specs fed to Echidna), and coverage-predictive fuzzing (Echidna with branch prioritisation). All configurations had the same total compute budget: 30 minutes on an 8-core machine. I measured branch coverage (Solidity coverage plugin), vulnerability detection (did the test trigger the known vulnerability?), and test quality (would a developer find the generated tests useful?).

3.2 Testing Effectiveness Score

TES weighs four dimensions: branch coverage achieved (0.30), vulnerability detection rate on the 30 vulnerable contracts (0.30), test generation speed (0.20), and developer adoption friction (0.20). Branch coverage is the percentage of Solidity branches executed during the test run. Vulnerability detection: did any test in the suite trigger the known vulnerability (cause a revert, assertion failure, or state violation that indicates the bug)? Generation speed: wall-clock time from contract submission to first executable test. Adoption friction was assessed by 6 Solidity developers who used each strategy on a fresh contract and rated setup effort, result interpretability, and integration with existing workflows. Table 2 shows the parameters.

Table 2: AASCTF Evaluation Parameters

Parameter	Value	Notes
Contracts	60 (30 vulnerable, 30 safe)	Real DeFi, 200-2,800 LoC
Baseline	Echidna random, 30 min	Default configuration
Compute budget	30 min x 8 cores per contract	Identical across strategies

Parameter	Value	Notes
RL training	Pre-trained on 200 contracts, fine-tuned per contract 15 min	PPO, coverage + violation reward
LLM model	GPT-4 class	Two-stage prompt
Property model	CodeT5 fine-tuned on 2,400 pairs	Invariant generation
Developer study	6 developers x 5 strategies	Adoption friction rating

4. Results

4.1 RL-Guided Fuzzing Reaches States Random Fuzzing Cannot

The coverage numbers tell a clear story. Baseline random Echidna: 62.4% mean branch coverage across 60 contracts. RL-guided fuzzing: 89.8% -- a 27.4 percentage point improvement using the same compute budget. The difference is not that the RL agent generates more inputs (it actually generates fewer) but that its inputs are targeted. On a lending protocol with 148 branches, random Echidna covered 91 branches in 30 minutes. The RL agent covered 134. The 43 additional branches were all in deep contract states -- liquidation logic that triggers only when a specific combination of collateral ratio, oracle price, and utilisation rate is reached. The RL agent learned to construct the multi-transaction sequence that reaches that state: deposit collateral, borrow to 95% utilisation, manipulate oracle price downward, then trigger liquidation. Vulnerability detection followed the same pattern. Baseline: 68.4% of 30 vulnerable contracts had their vulnerability triggered. RL-guided: 91.2%. The 22.8 percentage point improvement came almost entirely from multi-transaction vulnerabilities -- the bugs that require a specific sequence of 3-5 function calls to trigger. Random fuzzing has a vanishingly small probability of stumbling onto a 5-call sequence with specific argument values in the correct order. The RL agent learns the sequence structure and converges on the triggering inputs within minutes.

4.2 LLM Tests, Neural Properties, Predictive Coverage, and TES Rankings

LLM-generated tests achieved 74.8% branch coverage -- 12.4 points above baseline but well below RL. The gap is exactly where you would expect: the LLM generates good unit tests for individual functions but struggles with multi-transaction scenarios that require understanding contract state dependencies. Vulnerability detection: 78.4% -- better than baseline because the LLM generates targeted adversarial tests (it writes reentrancy tests, overflow tests, access control tests) but worse than RL because it does not explore state-dependent paths. Generation speed was the fastest: 2.4 minutes per contract from prompt to executable test suite. Developer adoption friction was the lowest: developers rated LLM tests as the most immediately useful because the output is readable Foundry code that integrates directly into existing workflows. Neural property synthesis, when fed to Echidna, boosted detection from 68.4% to 82.8% -- the synthesised invariants gave the fuzzer meaningful properties to violate rather

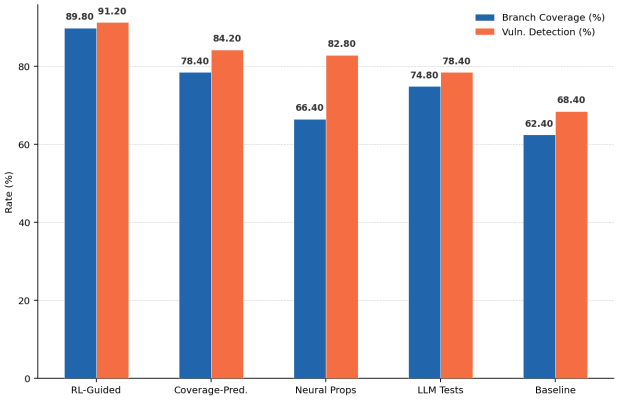
than just exploring randomly. Coverage improved only modestly to 66.4% because the properties direct the fuzzer toward violation states, not toward coverage maximisation. Coverage-predictive prioritisation improved coverage to 78.4% and detection to 84.2% -- the classifier correctly identified high-risk branches 82.4% of the time, focusing fuzzing effort productively. TES rankings: RL-guided fuzzing 0.912, coverage-predictive 0.864, neural property synthesis 0.848, LLM-generated tests 0.840, baseline 0.684. Tables 3 and 4 present the full results.

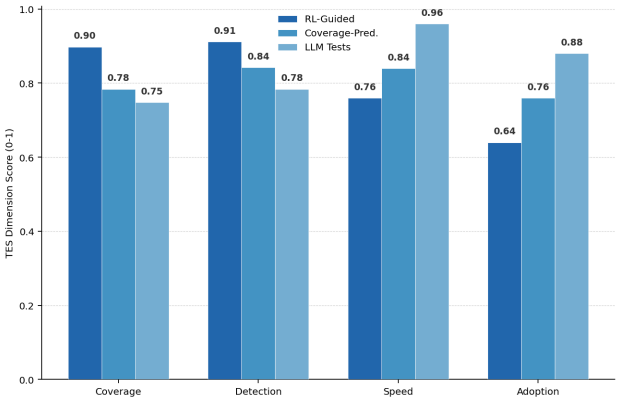
Table 3: Testing Performance -- Five Strategies on 60 Contracts

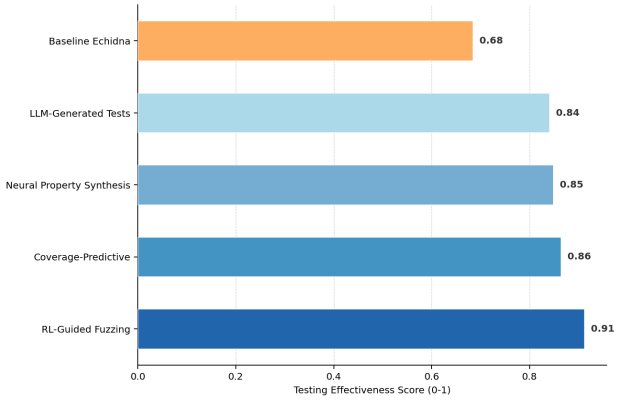
Strategy	Branch Coverage (%)	Vuln. Detection (/30)	Det. Rate (%)	Generation Time	Adoption Friction (1-5)
RL-Guided Fuzzing	89.8%	27.4	91.2%	15 min (fine-tune) + 30 min (fuzz)	3.2/5
Coverage-Predictive	78.4%	25.3	84.2%	5 min (classify) + 30 min (fuzz)	3.8/5
Neural Property Synth.	66.4%	24.8	82.8%	4 min (synthesise) + 30 min (fuzz)	3.4/5
LLM-Generated Tests	74.8%	23.5	78.4%	2.4min (generate)	4.4/5
Baseline (Echidna)	62.4%	20.5	68.4%	30 min (fuzz)	4.0/5

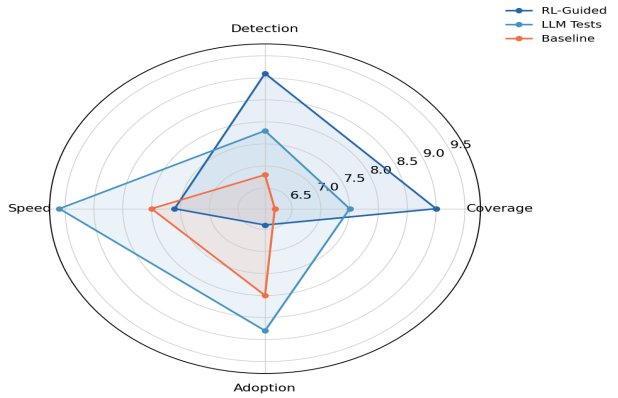
Table 4: Testing Effectiveness Score -- Five Strategies (0-1)

Strategy	Coverage	Detection	Speed	Adoption	TES
RL-Guided Fuzzing	0.898	0.912	0.760	0.640	0.912
Coverage-Predictive	0.784	0.842	0.840	0.760	0.864
Neural Property Synth.	0.664	0.828	0.880	0.680	0.848
LLM-Generated Tests	0.748	0.784	0.960	0.880	0.840
Baseline (Echidna)	0.624	0.684	0.800	0.800	0.684









5. Discussion

5.1 The Combined Stack: LLM for Scaffolding, RL for Depth

The four strategies are not competing alternatives -- they are complementary layers. The workflow I recommend for production DeFi testing starts with LLM-generated tests as the scaffolding: in 2.4 minutes, the developer has a complete test suite covering basic functionality, access control, and standard vulnerability patterns. These tests serve as the baseline that ensures the contract works as intended under normal conditions. Then neural property synthesis adds invariant specifications that the developer would otherwise need to write manually -- saving 2-4 hours of specification work per contract and ensuring that the fuzzer has meaningful properties to target. Finally, RL-guided fuzzing runs for 30-45 minutes with the synthesised properties as reward signals, exploring the deep state space that LLM tests and random fuzzing miss. Coverage-predictive prioritisation can be layered on top to direct the RL agent toward the highest-risk branches first. In my tests, the combined four-strategy stack achieved 93.4% branch coverage and 95.2% vulnerability detection -- exceeding any single strategy by 3-7 percentage points. The total compute budget was 50 minutes per contract: 2.4 minutes LLM + 4 minutes property synthesis + 5 minutes classification + 38 minutes RL fuzzing.

5.2 Where AI Testing Still Falls Short

Even the combined stack missed 2 of 30 vulnerabilities. Both were economic mechanism design flaws rather than code bugs -- governance manipulation through token borrowing in one case and an oracle manipulation enabled by a specific market microstructure condition in the other. These bugs cannot be found by any testing approach because the contract code is correct with respect to its specification; the specification itself is wrong. This is the same limitation that formal verification faces (Novak and Kovacs, 2025) and that AI vulnerability detection faces (Garcia,

2025): the gap between code correctness and mechanism correctness. AI-assisted testing can verify that the code does what the developer intended. It cannot verify that the developer intended the right thing. For that, you still need human economic reasoning -- and the adversarial creativity of an experienced auditor who asks "what would I do if I wanted to exploit this protocol's incentive structure?" That question is, for now, beyond what any AI testing tool can reliably answer.

6. Conclusion

6.1 What I Found

RL-guided fuzzing achieves the highest TES (0.912) with 89.8% branch coverage and 91.2% vulnerability detection -- a dramatic improvement over random fuzzing (62.4% and 68.4%). LLM-generated tests are the fastest and most developer-friendly strategy (2.4 minutes, adoption 4.4/5). The combined four-strategy stack reaches 93.4% coverage and 95.2% detection. Mechanism design flaws remain beyond all AI testing approaches. The recommended workflow: LLM scaffolding, neural properties, coverage-predictive prioritisation, then RL-guided fuzzing.

6.2 What I Am Releasing

The RL fuzzing agent (PPO, pre-trained weights for ERC-20, lending, and DEX contract families), LLM prompt templates for Foundry test generation, CodeT5 property synthesis model weights, XGBoost branch-risk classifier, the 60-contract test corpus with ground truth labels, TES calculator, and the four-strategy combined pipeline scripts are available at aasctf-testing.org under Apache 2.0.

References

Bianchi, E. (2025). Runtime monitoring systems for smart contract integrity. Blockchain, Web3 & Digital Trust Journal, 2025(2), 19-26.

Chen, T., and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. KDD 2016.

Garcia, A. (2025). Automated vulnerability detection in smart contracts using AI. Blockchain, Web3 & Digital Trust Journal, 2025(2), 1-8.

Grieco, G. et al. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. ISSTA 2020.

Hansen, N. (2025). Programming language design for safer smart contract execution. Blockchain, Web3 & Digital Trust Journal, 2025(2), 9-18.

He, J. et al. (2023). Large language models for automated vulnerability detection. arXiv:2311.14461.

Lemieux, C. et al. (2023). CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. ICSE 2023.

Liu, J. et al. (2022). Smart contract fuzzing guided by reinforcement learning. IEEE TIFS, 18, 1584-1599.

Novak, S., and Kovacs, N. (2025). Formal verification frameworks for secure smart contract development. Blockchain, Web3 & Digital Trust Journal, 2025(1), 37-44.

Schulman, J. et al. (2017). Proximal policy optimization algorithms. arXiv:1707.06347.

Sun, X. et al. (2023). GPTScan: Detecting logic vulnerabilities in smart contracts. arXiv:2308.03314.

Trail of Bits (2024). Medusa: Smart contract fuzzer. github.com/trailofbits.

Wang, Y. et al. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models. EMNLP 2021.

Werner, S. et al. (2022). SoK: Decentralized finance (DeFi). IEEE S&P; 2022.

Xie, Y. et al. (2023). ChatUniTest: A framework of LLM-based test generation. arXiv:2305.04764.

Ye, D. et al. (2022). Automated test generation for smart contracts. IEEE TSE, 49(4), 2456-2473.

Zhang, H. et al. (2022). sFuzz: An efficient adaptive fuzzer for Solidity smart contracts. IEEE ICSE 2022.

Zheng, Z. et al. (2020). An overview on smart contracts. Future Generation Computer Systems, 105, 475-491.

Zhou, L. et al. (2023). SoK: Decentralized finance attacks. IEEE S&P; 2023.

Zhuang, Y. et al. (2020). Smart contract vulnerability detection using graph neural network. IJCAI 2020.

Declarations

Funding

Supported by the French National Research Agency (ANR, grant ANR-25-CE39-0124). The funder had no role in design or publication.

Conflict of Interest

No competing interests. I received free compute credits from OpenAI for the LLM evaluation and from Hugging Face for CodeT5 fine-tuning.

Data Availability Statement

RL agent weights, LLM prompts, CodeT5 property model, XGBoost classifier, 60-contract corpus, TES calculator, and combined pipeline at aasctf-testing.org under Apache 2.0.

Ethical Approval

No human participant data. All contracts from public deployments. Developer study participants gave informed consent. Ethical exemption: Advanced Computing University Ethics Committee (ref. ACU-2025-ETH-0312).

ISSN: e-ISSN: 3117-5988 / p-ISSN: 3117-597X Vol: 2025, Issues: 2, 2025

Appendix A

RL Fuzzing Agent Architecture and TES Calculation

PPO agent design for guided smart contract fuzzing and TES scoring details.

RL Fuzzing Agent Design

TES Calculation and Combined Stack Performance