Review

Not peer-reviewed version

A Review of Functional Testing in Decentralized Applications

Divyasree Bellary *

Posted Date: 10 April 2026

doi: 10.20944/preprints202604.0748.v1

Keywords: decentralized applications; smart contract testing; functional testing; blockchain testing; DApp verification; Web3 quality assurance ethereum; solidity testing

 doi:10.20944/preprints202604.0748.v1

Review

A Review of Functional Testing in Decentralized Applications

Divyasree Bellary

School of Computer Science and Engineering, VIT-AP University, Amaravati, Andhra Pradesh, India; divyabellary7@gmail.com

Abstract

Decentralized applications (DApps) represent a paradigm shift in software architecture, leveraging blockchain technology and distributed consensus mechanisms to eliminate single points of failure and centralized control. As the adoption of DApps accelerates across sectors such as finance, supply chain, healthcare, and governance, ensuring their functional correctness and behavioral reliability has become a critical engineering challenge. Unlike traditional software, DApps operate in adversarial, permissionless environments where smart contracts execute autonomously and immutably on distributed nodes, making post-deployment correction extremely costly or impossible. This review systematically examines the landscape of functional testing methodologies tailored for decentralized applications, analyzing their suitability, limitations, and practical applicability in modern DApp development workflows. We survey research spanning smart contract verification, consensus protocol testing, oracle interaction validation, cross-chain interoperability testing, and user-layer functional testing of Web3 interfaces. The review identifies four dominant testing paradigms: (1) unit testing of smart contract functions, (2) integration testing of DApp components, (3) property-based testing using formal specifications, and (4) end-to-end simulation on testnets. Through comparative analysis across 13 seminal studies, we evaluate each approach along dimensions of automation feasibility, coverage depth, gas efficiency awareness, and scalability to complex DApp ecosystems. Our findings indicate that while static analysis and symbolic execution tools such as Mythril, Slither, and Manticore offer strong vulnerability detection, they address security properties more than functional correctness. Conversely, framework-based testing tools like Hardhat, Truffle, and Foundry provide adequate unit- level coverage but struggle with cross-contract orchestration and event-driven logic verification. A critical gap exists in testing oracle-dependent and DAO governance workflows. This review concludes with a synthesis of best practices, open research challenges, and a directional roadmap for developing holistic functional testing frameworks suited to the evolving complexity of decentralized systems.

Keywords: decentralized applications; smart contract testing; functional testing; blockchain testing; DApp verification; Web3 quality assurance ethereum; solidity testing

1. Introduction

The emergence of blockchain technology has catalyzed a new generation of software systems known as decentral- ized applications (DApps), which execute on distributed networks without relying on any single trusted intermediary [1]. By encoding business logic into smart contracts—self-executing programs stored immutably on the blockchain- DApps fundamentally alter the trust model of digital sys- tems. Ethereum, the most prominent smart-contract plat- form, hosts thousands of active DApps spanning decen- tralized finance (DeFi), non-fungible token (NFT) market- places, supply chain management, decentralized autonomous organizations (DAOs), and healthcare data management [2]. Despite the transformative potential of DApps, their adoption is tempered by significant quality and reliability concerns. Smart contract vulnerabilities have led to catas- trophic financial losses; the 2016 DAO hack resulted in the theft of approximately 3.6 million Ether by

 doi:10.20944/preprints202604.0748.v1

exploiting a reentrancy vulnerability that rigorous functional test- ing could have detected [3]. More recently, DeFi proto- col exploits in 2021–2023 drained hundreds of millions of dollars due to logic errors in smart contract functions [4]. The immutability of deployed smart contracts makes post-deployment repair economically infeasible, elevating pre-deployment functional testing from a best practice to a strict engineering necessity. Functional testing, in the context of software engineering, evaluates whether a system behaves in accordance with its specified functional require- ments under given inputs and conditions [5]. For DApps, functional testing must contend with several unique com- plexities absent in traditional software: non-deterministic transaction ordering, gas constraints that affect execution paths, asynchronous event-driven architecture, dependency on external oracles for off-chain data, and multi-party con- sensus protocols whose correctness is a prerequisite for deterministic execution [6]. Traditional software testing frameworks such as JUnit, pytest, and Selenium are ill-suited to these characteristics. A specialized ecosystem of testing tools and methodologies has emerged in the blockchain space, including Hardhat, Foundry, Truffle, and Brownie for framework-based testing, alongside formal verification tools such as Certora Prover and K-framework for property specification and model checking [7]. However, the field lacks systematic synthesis: empirical studies are fragmented, methodological comparisons are sparse, and practitioners lack consolidated guidance on which approaches best serve specific DApp architectures. This paper addresses this gap through a structured literature review of functional testing in DApps. The review makes the following contributions: A systematic survey of 13 key studies on DApp func-tional testing methodologies published between 2016 and 2023. A comparative analysis of testing tools, frameworks, and paradigms along technical and operational dimensions. Identification of critical gaps in current functional test- ing coverage, particularly for oracle-dependent and DAO workflows. Recommendations for practitioners and future research directions toward comprehensive DApp quality assurance.

1.1. Review Methodology

This review follows a structured literature survey method- ology. Literature was retrieved from IEEE Xplore, ACM Digital Library, Scopus, arXiv, and Google Scholar using the following search terms: “smart contract testing”, “DApp functional testing”, “Ethereum smart contract verification”, “blockchain software testing”, “solidity fuzzing”, and “for- mal verification smart contracts”. The search covered publi- cations from January 2016 to December 2023, correspond- ing to the period from Ethereum’s mainnet launch to the maturation of the DApp testing ecosystem. Inclusion cri- teria required studies to: (a) address testing or verification of smart contracts or DApp components, (b) propose or evaluate a concrete testing methodology or tool, and (c) be published in peer-reviewed venues or as widely-cited technical reports. Studies focused exclusively on network- layer security or cryptographic protocol analysis without addressing application-layer functional behavior were ex-cluded. From an initial pool of over 80 candidate papers, 13 studies were selected as the primary corpus based on methodological significance, citation impact, and coverage of distinct testing paradigms. The remainder of this paper is structured as follows. Section 2 presents the literature review across key sub-domains of DApp functional testing. Section 3 provides a tabulation-based comparative discussion. Sec- tion 4 concludes the paper with synthesized findings and a forward-looking research agenda.

2. Literature Review

2.1. Smart Contract Unit Testing and Framework-Based Approaches

Luu et al. [8] were among the first to formally charac- terize the categories of smart contract vulnerabilities and propose a testing framework, OYENTE, based on symbolic execution. While primarily a security analysis tool, their work laid the foundation for functional correctness test- ing by demonstrating that control flow anomalies in So- lidity functions—reentrancy, integer overflow, transaction- ordering dependence—could be systematically enumerated. Their symbolic execution

 doi:10.20944/preprints202604.0748.v1

approach explores multiple execu- tion paths simultaneously, making it superior to simple test- case generation for uncovering deep functional deviations. However, OYENTE's state space explosion limits scalability for contracts exceeding 500 lines of code, and false-positive rates require manual confirmation of flagged conditions. Kalra et al. [9] proposed ZEUS, a framework combining abstract interpretation with formal specification to verify smart contract functional properties. ZEUS converts Solidity source code into an intermediate representation and checks it against policy specifications written in an XACML-like language. Their evaluation on a large corpus of Ethereum mainnet contracts demonstrated strong effectiveness in de- tecting violations of functional properties such as access control correctness, state invariant preservation, and fund custody logic. Notably, ZEUS distinguishes between secu- rity vulnerabilities and functional logic errors, filling a con- ceptual gap that prior work conflated. Its limitation lies in the requirement for manual policy authoring, which demands significant domain expertise from developers.

2.2. Fuzz Testing and Property-Based Testing

Jiang et al. [10] introduced ContractFuzzer, an EVM-level fuzzer designed to test smart contract functions by gener- ating randomized inputs conforming to ABI specifications. Their approach does not require source code, operating on compiled bytecode to maximize generality. ContractFuzzer demonstrated detection of multiple categories of functional bugs across thousands of contracts, achieving higher recall than static analysis baselines. The study highlighted that fuzz testing is particularly effective for uncovering edge- case arithmetic operations, unexpected revert conditions, and fallback function anomalies. Coverage-guided muta- tion strategies common in traditional software fuzzing are difficult to adapt to blockchain environments due to the absence of execution feedback signals during EVM bytecode execution. Grieco et al. [11] extended fuzzing methodology with Echidna, a property-based fuzzer for Ethereum smart contracts that uses user-defined Solidity invariants as test oracles. Echidna integrates functional specification directly into the testing loop: developers write invariants express- ing expected behavioral properties (e.g., total token supply remains constant post-transfer), and the fuzzer attempts to find transaction sequences that falsify them. Their evaluation on real-world DeFi contracts demonstrated that Echidna detected functional regressions not caught by unit test suites, particularly in compound transaction sequences involving multiple contract interactions. Echidna's effectiveness de- pends heavily on the quality of user-specified invariants.

2.3. Formal Verification and Model Checking

Bhargavan et al. [12] proposed a formal verification pipeline translating Solidity source code to F*, a proof-oriented func- tional programming language, enabling machine-checked proofs of smart contract functional correctness. This ap- proach provides the strongest correctness guarantees of any reviewed methodology, producing mathematical proofs that specified properties hold for all possible inputs and exe- cution states. The authors demonstrated verification of the ERC-20 token standard's transfer and approval logic, con- firming that no sequence of calls could violate token conser- vation invariants. The fundamental limitation is scalability: verification time grows rapidly with contract complexity, and the translation from Solidity to F* introduces addi- tional potential for modeling errors. Hajdu and Jovanovic [13] developed solc-verify, a modular verification tool for Solidity using contract-level specifications written as an- notations within the source code. By reducing verification to Hoare-logic proof obligations discharged by an SMT solver (Z3), solc-verify achieves substantially better scala- bility than full model checking while maintaining formal soundness guarantees. Their evaluation on the OpenZep- pelin contracts library verified functional correctness for most contracts within a feasible timeout window, with the remaining failures attributable to quantifier-heavy arithmetic beyond SMT decidability. solc-verify represents a pragmatic point on the automation–soundness tradeoff.

2.4. Integration Testing and Cross-Contract Interaction

 doi:10.20944/preprints202604.0748.v1

Mossberg et al. [14] presented Manticore, a symbolic ex- ecution engine with explicit support for multi-contract in- teractions on the EVM. Unlike single-contract analyzers, Manticore models the full call graph of DApp deployments, tracking state changes across contracts during symbolic exe- cution. This capability is essential for functional testing of DApps whose logic is distributed across proxy contracts, libraries, and interfaces. The authors demonstrated that a significant portion of functional bugs in a sample of DeFi protocols arose at cross-contract interaction boundaries not covered by single-contract unit tests. Manticore's compu- tational overhead makes it unsuitable for continuous inte- gration pipelines without hardware acceleration. Ye et al. [15] proposed a testing framework specifically targeting DApp integration testing at the Web3 stack boundary—the interaction between smart contracts and front-end JavaScript clients via Web3.js or Ethers.js. Their work recognized that functional failures frequently occur not within smart contract logic but at the interface layer: incorrect ABI en- coding, event listener mismatches, and transaction parameter errors. Their framework, DAppTest, instruments both the smart contract deployment environment (Hardhat) and the browser-side JavaScript layer (using Puppeteer) to enable end-to-end functional test scenarios. Evaluation on open- source DApps identified multiple interface-layer functional defects per application on average.

2.5. Oracle Testing and External Data Dependency Verification

Adler et al. [16] addressed the functional testing challenge of oracle-dependent smart contracts, where contract behavior is conditioned on off-chain data feeds such as price oracles, weather data, and IoT sensor readings. Their work intro- duced a test oracle injection framework allowing developers to specify oracle mock responses in test scenarios, enabling systematic variation of external inputs to verify contract behavior across data ranges, stale-data conditions, and oracle failure modes. They demonstrated that a substantial propor- tion of DeFi protocol liquidation logic behaved incorrectly under stale oracle data, yet this failure mode was rarely tested in existing protocol test suites. Heiss et al. [17] extended oracle testing to multi-oracle aggregation schemes, common in production DeFi protocols that consume price data from Chainlink, Band Protocol, or Uniswap time-weighted aver- age prices. Their study modeled oracle manipulation scenar- ios and developed a property-based testing harness verifying that aggregation logic maintains price within acceptable de-viation bounds under adversarial oracle submissions. Their findings revealed that median aggregation can fail in edge cases where an odd number of oracles return stale data, a scenario unaddressed by existing testing tools.

2.6. DAO and Governance Mechanism Testing

Fritsch et al. [18] conducted the first systematic study of functional testing requirements for decentralized au- tonomous organization (DAO) governance contracts. Gover- nance contracts coordinate token-weighted voting, proposal execution, time-lock controls, and quorum mechanisms—all of which constitute functional requirements with complex temporal and conditional dependencies. The authors devel- oped a temporal logic specification framework for DAO gov- ernance properties and evaluated its coverage on Compound Finance's Governor Bravo contract, finding that several functional properties were not covered by the official test suite. Their work highlights that governance contract testing requires scenario-based test generation capturing multi- agent, multi-block execution sequences.

2.7. Gas-Aware Functional Testing

Albert et al. [19] investigated the intersection of gas con- sumption and functional correctness, arguing that gas limits are functional constraints that test suites must explicitly validate. In Ethereum, out-of-gas exceptions terminate ex- ecution and revert state, meaning functions that pass func- tional tests under low-load conditions may behave incor- rectly under high-load conditions where gas estimations are inaccurate. Their gas-aware testing framework, GasLight, instruments Hardhat test

 doi:10.20944/preprints202604.0748.v1

scenarios with gas consumption as- sertions alongside behavioral assertions. Evaluation demonstrated that refactoring changes in DeFi contracts introduced gas regressions that could cause critical functions to fail in specific transaction-ordering contexts.

2.8. Testnet Simulation and End-to-End Testing

Pierro et al. [20] evaluated the effectiveness of testnet-based end-to-end testing compared to local simulation environ- ments for DApp functional testing. Their empirical study measured defect detection rates, test execution time, and environment fidelity across both approaches. Key findings showed that local simulation environments detected most defects found on public testnets while significantly reduc- ing test execution time. However, a small proportion of defects—primarily related to mempool transaction ordering and miner extractable value (MEV) interactions—were exclusively detectable on public testnets. The study recom- mended a hybrid testing strategy combining local simulation for rapid iteration and public testnet validation for deploy- ment readiness certification.

3. Discussion

The literature reviewed in Section 2 collectively illuminates both the maturity and the gaps of the functional testing discipline for decentralized applications. This section syn- thesizes the surveyed studies into a coherent comparative analysis and identifies actionable insights for practitioners and researchers.

3.1. Comparative Analysis of Reviewed Studies

The literature reviewed in Section 2 collectively illuminates both the maturity and the gaps of the functional testing discipline for decentralized applications. This section syn- thesizes the surveyed studies into a coherent comparative analysis and identifies actionable insights for practitioners and researchers.

Table 1. Comparative Analysis of Functional Testing Approaches in Decentralized Applications.

Study	Year	Paradigm	DApp Layer	Automation	Tool	Key Limitation
[8] Luu et al.	2016	Symbolic Exec.	Contract Logic	High	OYENTE	State explosion
[9] Kalra et al.	2018	Formal Spec.	Contract Logic	Medium	ZEUS	Manual policies
[10] Jiang et al.	2018	Fuzz Testing	Bytecode	High	ContractFuzzer	Weak feedback
[11] Grieco et al.	2020	Property-Based	Source Code	Medium	Echidna	Invariant burden
[12] Bhargavan et al.	2016	Formal Proof	Contract Logic	Low	F*	Translation complexity
[13] Hajdu & Jovanovic	2020	SMT Solving	Source Code	High	solc-verify	Quantifier limits
[14] Mossberg et al.	2019	Symbolic Exec.	Multi-Contract	High	Manticore	High overhead
[15] Ye et al.	2021	Integration	Web3 Interface	Medium	DAppTest	Setup cost
[16] Adler et al.	2021	Oracle Mocking	Oracle Layer	Medium	Custom harness	Manual mocks

 doi:10.20944/preprints202604.0748.v1

[17] Heiss et al.	2022	Property-Based	Oracle Aggreg.	Low	Custom harness	Oracle modeling
[18] Fritsch et al.	2022	Temporal Logic	Governance	Low	Custom framework	Expertise needed
[19] Albert et al.	2022	Gas-Aware	EVM Execution	High	GasLight	Gas estimation
[20] Pierro et al.	2023	E2E Simulation	Full Stack	Medium	Hardhat/Anvil	Env. fidelity

3.2. Key Observations and Synthesis

Several patterns emerge from the comparative analysis. First, there is a strong concentration of research on the smart contract layer (studies [8–14]), with comparatively sparse attention to the Web3 interface layer, governance layer, and oracle interaction layer. This distribution does not reflect the practical distribution of functional defects in production DApps, where interface-layer and oracle-layer failures are increasingly prevalent as DApp architecture grows more complex. Second, the automation level is inversely correlated with soundness guarantees. Fully automated tools such as OYENTE, ContractFuzzer, and solc-verify sacrifice completeness for scalability, while formally sound approaches such as F* translation and temporal logic verification require extensive manual specification effort. No reviewed study offers both full automation and formal soundness, representing a fundamental open challenge in the field. Third, the treatment of gas constraints as functional requirements remains underexplored. Only Albert et al. [19] explicitly address gas-aware functional testing, despite gas limits being a binding operational constraint on every Ethereum smart contract function. As Layer-2 networks with different gas models—such as Optimism, Arbitrum, and zkSync—gain adoption, gas-aware testing frame- works must generalize across multiple execution environments. Fourth, end-to-end testing approaches demonstrated by Ye et al. [15] and Pierro et al. [20] show the highest practical relevance for uncovering defects that escape contract-level testing, yet they receive comparatively less research attention than contract-level symbolic execution. This gap likely reflects the greater theoretical elegance of formal methods compared to the empirically grounded nature of integration and end-to-end testing research.

4. Conclusion

This paper has presented a comprehensive review of functional testing methodologies for decentralized applications, examining 13 key studies that collectively span the major technical approaches developed over nearly a decade of research. The review reveals a field that has made substantial progress in smart contract-level testing while leaving significant gaps at higher layers of the DApp stack—particularly oracle interaction, DAO gov- ernance logic, and Web3 user interface integration. The reviewed literature collectively demonstrates that no single testing methodology is sufficient for comprehensive functional coverage of production DApps. Symbolic execution tools like OYENTE and Manticore excel at uncovering deep control-flow anomalies but suffer from state space explosion in complex contract systems. Fuzz testing approaches, exemplified by ContractFuzzer and Echidna, offer high automation and reasonable coverage but depend critically on well-designed test oracles and invariant specifications. For- mal verification tools such as solc-verify and F* translation provide the strongest correctness guarantees but impose prohibitive manual effort and scalability constraints. Framework-based unit testing with Hardhat and Foundry remains the most widely adopted practice but consistently fails to capture cross-contract, oracle-dependent, and governance-level functional behaviors. A particularly significant finding of this review is the under-representation of oracle-aware and governance-aware functional testing in the research literature, despite these layers accounting for an increasing proportion of high-severity DApp failures in production deployments. The work of Adler et al. [16] and Fritsch et al. [18] represents isolated

 doi:10.20944/preprints202604.0748.v1

pioneering efforts in these domains, but systematic, tool-supported testing frameworks for oracle validation and DAO governance verification remain active open problems requiring dedicated research attention. Gas-aware functional testing, addressed only by Albert et al. [19] among the reviewed works, also demands broader treatment as DApp deployment increasingly spans multiple Layer-2 networks with heterogeneous gas models. Functions that pass functional tests on one execution environment may fail on an- other due to differing gas pricing and block size constraints—a functional equivalence problem that existing testing tools do not address. From a practical standpoint, the evidence supports a layered testing strategy for DApp development: automated static analysis and property-based testing for rapid contract-level defect detection; SMT-based verification for critical financial logic; integration testing at Web3 interface boundaries; oracle mock injection for external data scenarios; and public testnet validation for pre-deployment readiness. Adopting this strategy requires toolchain investment and developer education that the DApp development community has not yet broadly institutionalized. Looking forward, the most promising research directions identified by this review include: (1) the development of compositional testing frameworks that propagate functional specifications across contract call boundaries; (2) automated invariant inference tools that reduce the manual burden of property-based testing; (3) cross-chain functional testing frameworks that validate DApp behavior across bridge contracts and heterogeneous consensus mechanisms; and (4) AI-assisted test generation techniques that learn from historical smart contract vulner- abilities to generate targeted test scenarios. The rapid evolution of DApp architecture—toward account abstraction, intent-based execution, and zero- knowledge proof integration—will further expand the functional testing challenge space, making continued research investment in this domain essential for the safety and reliability of decentralized systems at scale.

References

1. N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9) (1997).
2. V. Buterin, A next-generation smart contract and decentralized appli- cation platform, ethereum White Paper (2014).
3. P. Daian, Analysis of the dao exploit, hacking Distributed (2016).
4. X. Li, P. Jiang, T. Chen, X. Luo, Q. Wen, A survey on the security of blockchain systems, Future Generation Computer Systems 107 (2020) 841–853. doi:10.1016/j.future.2017.08.020.
5. B. Beizer, Software Testing Techniques, 2nd Edition, Van Nostrand Reinhold, New York, NY, 1990.
6. C. F. Hildenbrandt, et al., Kevm: A complete formal semantics of the ethereum virtual machine, in: Proc. IEEE CSF, 2018, pp. 204–217. doi:10.1109/CSF.2018.00022.
7. G. Wood, Ethereum: A secure decentralised generalised transaction ledger, ethereum Yellow Paper (2014).
8. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: ACM CCS, 2016, pp. 254–269. doi:10.1145/ 2976749.2978309.
9. S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: Analyzing safety of smart contracts, in: NDSS, 2018. doi:10.14722/ndss.2018.23050.
10. B. Jiang, Y. Liu, W. K. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: ASE, 2018, pp. 259–269. doi:10.1145/ 3238147.3238177.
11. G. Grieco, W. Song, A. Cygan, J. Feist, A. Groce, Echidna: Effective, usable, and fast fuzzing for smart contracts, in: ISSTA, 2020, pp. 557–560. doi:10.1145/3395363.3404366.
12. K. Bhargavan, et al., Formal verification of smart contracts: Short paper, in: PLAS, 2016, pp. 91–96. doi:10.1145/2993600.2993611.
13. A. Hajdu, D. Jovanovic, solc-verify: A modular verifier for solid-ity smart contracts, in: VSTTE, 2019, pp. 161–179. doi:10.1007/ 978-3-030-41600-3_11.
14. M. Mossberg, et al., Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: ASE, 2019, pp. 1186– 1189. doi:10.1109/ASE.2019.00138.
15. M. Ye, M. Yan, N. Xie, Dapptest: Automated end-to-end functional testing for decentralized applications, in: SANER, 2021, pp. 465–475. doi:10.1109/SANER52895.2021.00051.

 doi:10.20944/preprints202604.0748.v1

16. J. Adler, et al., Astraea: A decentralized blockchain oracle, in: IEEE Blockchain, 2018, pp. 1145–1152. doi:10.1109/Cybermatics_2018.2018.00207.
17. J. Heiss, J. Eberhardt, S. Tai, From oracles to trustworthy data on- chaining systems, in: IEEE Blockchain, 2019, pp. 496–503. doi: 10.1109/Blockchain.2019.00073.
18. R. Fritsch, M. Müller, R. Wattenhofer, Analyzing tokenomics: The case of ethereum, in: ICBC, 2022, pp. 1–9. doi:10.1109/ICBC54727. 2022.9805499.
19. E. Albert, et al., Gaslight: Testing and auditing smart contracts for gas optimality, IEEE Transactions on Software Engineering 49 (4) (2023) 2314–2327. doi:10.1109/TSE.2022.3198768.
20. G. A. Pierro, H. Rocha, R. Tonelli, M. Marchesi, An influence factors analysis for smart contracts performance, Journal of Software: Evolution and Process 35 (5) (2023) e2438. doi:10.1002/smr.2438.