Feb 23, 2026

SARMF: Smart Contract Automated Remediation and Mitigation Framework

DOI
https://dx.doi.org/10.17504/protocols.io.bp2l6eyxdgqe/v1

Mohit Tiwari[1]

[1]Bharati Vidyapeeth's College of Engineering, New Delhi, India

Mohit Tiwari: Assistant Professor, Department of Computer Science & Engineering. Research interests include blockchain security, cybersecurity frameworks, and automated vulnerability analysis.

AI and Digital Trust Rese...

Mohit Tiwari
Bharati Vidyapeeth's College of Engineering,Delhi

DOI: https://dx.doi.org/10.17504/protocols.io.bp2l6eyxdgqe/v1

Protocol Citation: Mohit Tiwari 2026. SARMF: Smart Contract Automated Remediation and Mitigation Framework. **protocols.io** https://dx.doi.org/10.17504/protocols.io.bp2l6eyxdgqe/v1

Protocol status: Working
We use this protocol and it's working

Created: February 22, 2026

Last Modified: February 23, 2026

Protocol Integer ID: 243800

Keywords: Smart Contracts, Blockchain Security, Vulnerability Analysis, Automated Mitigation, Ethereum, Cybersecurity, Digital Trust, Smart Contract Security, Blockchain Security, Vulnerability Detection, Static Analysis, Automated Mitigation, Secure Coding, Smart Contract Audit, SWC Registry, Reentrancy Prevention, Decentralized Applications, structured workflow for smart contract vulnerability detection, systematic foundation for secure smart contract lifecycle management, smart contract vulnerability detection, reproducible security engineering pipeline, unified reproducible security engineering pipeline, secure smart contract lifecycle management, smart contract automated remediation, based automated patch generation, automated patch generation, adversarial validation of ethereum, mitigation framework sarmf, compatible smart contract, vulnerability detection, mitigation within blockchain, vulnerability normalization, automated mitigation generation, mitigation framework, automated mitigation, combining st

Disclaimer

This protocol is intended for research and controlled testing environments. Deployment of smart contracts in production systems should be performed with additional independent audits and compliance verification.

Abstract

SARMF (Smart Contract Automated Remediation and Mitigation Framework) is a structured and reproducible security engineering pipeline designed for vulnerability detection, taxonomy alignment, automated remediation, and adversarial validation of Ethereum-compatible smart contracts.

This operational protocol presents a structured workflow for smart contract vulnerability detection and automated mitigation within blockchain-based systems. The methodology integrates deterministic environment setup, multi-tool static analysis, vulnerability normalization using standardized taxonomies, rule-based automated patch generation, and dynamic adversarial validation. By combining static detection tools with controlled refactoring patterns and behavioral verification, the framework ensures reproducibility, traceability, and measurable performance impact assessment. The protocol concludes with comprehensive audit reporting and archival procedures to support transparency and independent verification. This workflow provides a systematic foundation for secure smart contract lifecycle management in decentralized applications.

Unlike traditional audit checklists, this framework operationalizes vulnerability detection, taxonomy alignment, automated remediation generation, and validation feedback loops into a unified reproducible security engineering pipeline.

Key Contributions of SARMF:

1. Deterministic environment and compilation reproducibility model.
2. Unified multi-tool vulnerability normalization aligned with SWC taxonomy.
3. Rule-based automated mitigation generation preserving semantic integrity.
4. Iterative validation loop combining static, adversarial, and fuzz testing.
5. Structured audit archival enabling independent verification and traceability.

SECTION 1 – Project Intake and Environment Setup

1 Clone the smart contract repository and freeze the exact commit hash (SHA-256 or Git commit ID) for audit reproducibility. Identify the Solidity compiler version specified in the pragma directive and install the exact solc version using a version manager (e.g., solc-select). Record dependency versions via lock files (package-lock.json / yarn.lock).

2 Install required dependencies and libraries. Confirm successful compilation without warnings. Document environment configuration including OS version, compiler version, toolchain versions, and containerization details (e.g., Docker image hash) to ensure deterministic rebuild capability.

SECTION 2 – Static Vulnerability Detection

3 Run automated static analysis tools (e.g., Slither for structural analysis, Mythril for symbolic execution) on compiled contract artifacts. Export machine-readable reports (JSON format) identifying vulnerabilities such as reentrancy, arithmetic overflows, unchecked external calls, and improper access control mechanisms.

4 Normalize and categorize detected vulnerabilities by severity (Critical, High, Medium, Low) using CVSS-inspired scoring where applicable. Remove duplicate findings across tools and consolidate results into a unified vulnerability matrix.

5 Map detected vulnerabilities to standardized taxonomies such as the Smart Contract Weakness Classification (SWC) Registry. Document corresponding SWC IDs, affected functions, and line references to enable traceable remediation and audit transparency.

6 Construct a structured vulnerability knowledge base mapping SWC categories to remediation templates, enabling systematic vulnerability-to-fix translation.

SECTION 3 – Automated Mitigation and Patch Generation

7 For each confirmed vulnerability, generate candidate mitigation strategies using predefined secure coding templates aligned with SWC remediation guidelines. Apply rule-based transformation logic to produce patch candidates while preserving semantic equivalence.

8 Apply automated refactoring where feasible (e.g., replacing unsafe external calls with checks-effects-interactions patterns, introducing access modifiers, or using SafeMath equivalents where required).

9 Recompile patched contracts and re-run static analysis to verify elimination of previously detected vulnerabilities. Perform differential vulnerability analysis and confirm absence of regression vulnerabilities.

10 Incorporate a feedback loop mechanism where mitigation outcomes are re-evaluated through static and dynamic analysis outputs, enabling iterative refinement of patch strategies until no critical or high-severity vulnerabilities persist.

11 Implement an iterative remediation feedback loop in which patched contracts are re-evaluated through static, symbolic, and dynamic testing until no critical vulnerabilities persist, forming a closed-loop security refinement cycle.

SECTION 4 – Dynamic Testing and Behavioral Validation

12 Deploy the patched smart contract to a controlled local blockchain environment (e.g., Hardhat, Ganache).
Execute predefined unit and integration test suites to validate functional correctness post-mitigation.

13 Conduct adversarial testing scenarios simulating exploit patterns (e.g., reentrancy, malicious fallback invocation, privilege escalation attempts) using scripted attack contracts to empirically validate resilience of the patched implementation.

14 Measure gas consumption differentials between original and patched contracts.
Document performance overhead introduced by mitigation logic to evaluate optimization trade-offs.

15 Perform fuzz testing using automated transaction generators to identify edge-case execution paths and ensure no latent vulnerabilities remain after mitigation.

SECTION 5 – Reporting, Documentation, and Audit Archival

16 Generate a structured security engineering report synthesizing vulnerability detection metrics, remediation effectiveness, adversarial validation outcomes, and performance trade-off analysis. including vulnerability matrix, mitigation mapping, validation results, fuzz testing outcomes, gas analysis, and reproducibility metadata. Archive all artifacts (source code, tool outputs, attack scripts, environment configuration, container images) for independent verification and long-term traceability.