SCBF: AUTOMATED DETECTION AND PATCHING OF SMART CONTRACT VULNERABILITIES VIA MYTHRIL AND OPEN ZEPPELIN INTEGRATION

Iman Darvishi[1], Alireza Esfahani[1] and Hadeel Alsolai[2]

[1]School of Computer Engineering, University of West London, London, UK
[2]Department of Information Systems College of Computer and Information Sciences, Princess Nourah Bint Abdul rahman University, Riyadh, Saudi Arabia

ABSTRACT

The Smart Contract Bug Fix (SCBF) framework is an open-source platform for automated detection and remediation of vulnerabilities in Ethereum and decentralised finance smart contracts. SCBF combines symbolic execution through Mythril with OpenZeppelin-based repair strategies to support an end-to-end workflow from vulnerability scanning to patch generation and reporting. The framework organises analysis results using SWC-based classification, applies deterministic patching rules, and exports logs and results through an analytics dashboard. SCBF was evaluated on two public datasets, SmartBugs Curated and Messi-Q. Under the counting scheme adopted in this paper, the framework achieved a fixed rate of 68.5% (170 of 248 Mythril findings) on SmartBugs Curated and a consolidated fix rate of 77.3% (958 of 1,239 findings) on Messi-Q. The results indicate effective handling of several common SWC classes, including tx. origin misuse, arithmetic issues, and re-entrancy related patterns, while also showing lower performance on environment-dependent vulnerabilities. These findings indicate that SCBF supports reproducible and traceable smart contract remediation workflows.

KEYWORDS

Automated Patch, Blockchain, Bug Fix, Security, Smart Contracts, Solidity, Symbolic Execution, Vulnerability Remediation

1. INTRODUCTION

Decentralised Finance (DeFi) applications, primarily built on the Ethereum blockchain, have revolutionised programmable, permission less financial services, yet remain highly vulnerable to smart contract exploits that can have significant financial consequences. Leading systematic literature reviews reveal a strong focus on vulnerability detection; however, automated and deterministic remediation remains largely unaddressed, creating a critical gap between risk identification and organisational-scale mitigation. Despite progress in symbolic analysis (e.g., Mythril), static analysers (e.g., Slither and Oyente), and benchmarking protocols, existing tools still exhibit several limitations: detection-only platforms [1]–[3] provide no automated patching capability; bytecode-centric approaches [4] sacrifice source-level auditability and standards alignment; template-based solutions[5], [6] remain bounded by predefined repair patterns; and emerging LLM-agentic tools sacrifice reproducibility and organisational control for broader scope. Common vulnerabilities such as re-entrancy (SWC-107), arithmetic overflows(SWC-101), and logic errors continue to pervade, as illustrated by curated datasets like *SmartBugs*. These architectural gaps are particularly critical in DeFi, where interoperability, composability, and governance-compliance expand both the attack surface and remediation demands. This work presents the Smart Contract Bug Fix (SCBF)framework, a platform that automates the smart

contract remediation workflow, from vulnerability detection to patch generation and reporting. SCBF integrates SWC-based vulnerability classification, deterministic fix routing, and template-based patch generation to support end-to-end remediation of selected vulnerabilities in Ethereum smart contracts. In its current implementation, the framework relies on Mythril for vulnerability detection, while SCBF performs classification, patch selection, remediation, and reporting based on Mythril's findings. Mythril's analysis includes symbolic execution and internal AST- and bytecode-level processing, which SCBF uses indirectly via the integrated scanner. SCBF is evaluated on two public datasets, *SmartBugs* Curated and *Messi-Q*. The reported results indicate that the framework can support remediation across several common vulnerability classes while providing a traceable, reproducible workflow for detection, patching, and reporting. A comparative discussion with existing smart contract analysis and repair tools is provided later in the paper. This comparison is intended to position SCBF with respect to workflow integration, automated patching support, reporting capability, and standards alignment relative to detection-only, bytecode-level, template-based, and learning-based approaches. The main contributions of this work are:

1. Development of an end-to-end automated framework centred on Mythril-based vulnerability detection and SCBF-based SWC classification, deterministic fix routing, and OpenZeppelin-aligned patch synthesis, with architectural extensibility for future integration of additional analysis and verification tools.
2. Formalisation of a deterministic SWC-to-fix routing and injection framework to support reproducibility, traceability, and audit-oriented remediation workflows while avoiding the non-deterministic behaviour associated with LLM-based approaches.
3. Operational support infrastructure, including analytics dashboards, detailed logging, automated CSV/JSON/SQL export, and deployment automation to support reproducible and traceable smart contract remediation workflows.
4. Empirical evaluation on two public datasets, *SmartBugs* Curated and *Messi-Q*, with transparent logging and reproducible reporting to assess SCBF's automated remediation performance across multiple vulnerability classes.
5. The remainder of this paper is organised as follows. Section 2 reviews related work on smart contract vulnerability detection and automated remediation. Section 3 outlines the background, research problem, and current scope of the framework. Section 4 presents the experimental methodology and evaluation protocol. Section 5 describes the SCBF architecture and implementation pipeline. Section 6 reports the experimental results, comparative evaluation, and discussion. Finally, Sections 7 and 8 present the conclusion and future research directions, respectively.

2. RELATED WORK

2.1. Detection Only

Recent research confirms the widespread availability of platforms for smart contract vulnerability detection; however, architectural limitations in both the extent of detection and the automation of remediation persist, leaving a critical operational gap [7]–[17]. Symbolic execution engines such as Mythril support vulnerability detection through constraint solving and adjustable-depth analysis, but they do not provide integrated source-level remediation. Static analysis tools, including Oyente, Slither, Securify, and SmartCheck, provide valuable support for detection and classification, including SWC-oriented analysis in some cases, but generally require manual interpretation and separate post-detection repair. Gas-focused analysers [2] demonstrate strong scalability for resource-related issues, although their scope of vulnerability remains narrower than that of general security analysis. Machine learning and deep learning approaches have improved

feature extraction and detection precision through methods such as opcode-based analysis [1], graph neural networks, and source-code vulnerability detection models [54], but they remain primarily detection-oriented and offer limited deterministic support for patch generation. Overall, the persistent detection-remediation gap reflects a common architectural pattern in which tools prioritise vulnerability discovery over reproducible repair, leaving organisations to manually translate findings into fixed code, a labour-intensive and error-prone process that is difficult to scale for governance-oriented smart contract security.

2.2. Early Remediation and Bytecode level Patching

SmartShield [4] pioneered automated bytecode-level patching, demonstrating the feasibility of automated repair across 28,621 contracts and reporting 91.5% success across three vulnerability types. However, its bytecode-centric approach limits source-level auditability and primarily addresses arithmetic, re-entrancy, and external-call vulnerabilities. Although technically effective for selected low-level patterns, bytecode-level patching creates governance and maintainability challenges because patched bytecode is difficult to review, align with secure coding standards, audit for compliance, or integrate into conventional source-level development workflows. These limitations highlight the need for remediation frameworks that preserve source-level transparency while supporting automated, reproducible repair.

2.3. Template-Based Remediation

Recent advances in template-based repair [5], [6] integrate Slither or Mythril detection with predefined repair patterns, report moderate patching success on constrained vulnerability sets, and are beginning to address standards alignment through OpenZeppelin-based repair templates. These approaches improve on bytecode-level repair by preserving greater source-level transparency and by aligning selected patches with recognised secure-development libraries. However, they remain constrained by template cardinality: predefined repair patterns can only address vulnerabilities that structurally match the available templates. Contemporary LLM-agentic approaches [18] extend repair possibilities beyond fixed templates, but they also introduce risks, including non-deterministic outputs, hallucination-prone repair suggestions, API or runtime bottlenecks, reduced reproducibility, and the broader interpretability concerns associated with AI-driven cybersecurity systems [55]. Therefore, although LLM-based methods may broaden detection and repair coverage, their use in governance-sensitive or financial smart contract environments requires stronger controls to ensure auditability, repeatability, and alignment with standards.

2.4. Benchmarking, Provenance, and Industry Gap

Benchmarking platforms such as *SmartBugs* and *Messi-Q* facilitate objective tool evaluation and have driven research advances; however, they primarily support comparative assessment rather than the deployment of remediation. OpenZeppelin Contracts remains the de facto community standard for secure Solidity components, including *ReentrancyGuard*, *AccessControl*, and *SafeMath*, yet it is still underused as a direct integration target in automated remediation workflows. Systematic literature reviews and meta-analyses consistently identify the detection-remediation gap as unresolved: existing tools often achieve strong vulnerability detection performance but generally lack operational, verifiable, and governance-aligned automated patching workflows.

2.5. SCBF's Positioning

Fig. 9 positions SCBF relative to detection-only, bytecode-level, template-based, and learning-based approaches. Unlike detection-oriented tools such as Slither, Securify, Mythril, ContractWard, and MadMax, SCBF focuses on connecting vulnerability detection to deterministic patch selection, remediation, post-fix verification, and structured reporting. In its current implementation, SCBF uses Mythril for vulnerability detection and OpenZeppelin-aligned templates for source-level repair, with support for logging, SQL and CSV exports, and dashboard-based auditability. The framework addresses the detection-remediation gap by providing an end-to-end workflow from batch scanning to post-fix verification [19]–[23], while preserving reproducibility, standards alignment, operational transparency, and extensibility toward additional analysis and verification tools. This positions SCBF as a platform for governance-auditable smart contract remediation rather than a detection-only analyser.

3. BACKGROUND AND PROBLEM STATEMENT

Blockchain-based smart contracts have transformed decentralised finance (DeFi) and automated asset management, but face a rapidly expanding attack surface, exposing persistent and emerging vulnerabilities at scale. Foundational research identifies critical threats, including re-entrancy, unchecked call returns, and exception-state errors, as pervasive across Ethereum and EVM-compatible blockchains [23]–[25]. Recent incidents of logic-centric and DeFi-specific exploits, such as oracle manipulation, flash loan attacks, and composability flaws, have led to multimillion-dollar financial losses and legal actions across multiple jurisdictions [26], [27]. Despite extensive academic and industrial focus, systematic analyses reveal that DeFi vulnerabilities are both technical and socioeconomic, causing cascading effects throughout decentralised ecosystems [28], [29]. Benchmarking studies confirm that leading detection tools such as Oyente, Slither, Mythril, SmartCheck, Securify, Manticore, and SoliAudit primarily employ static analysis, symbolic execution, and syntactic pattern matching. However, these methods exhibit limited scalability and precision for complex dynamic and compositional vulnerabilities [30]–[32]. Large-scale reviews report high false positive rates, insufficient coverage of advanced vulnerability classes, and minimal support for remediation or post-fix auditing [11], [33]. Furthermore, many DeFi-centric platforms lack comprehensive handling of inter-contract dependencies, economic-layer interactions, and business-logic validation [8], [34].Recent progress in machine learning (ML) and large language models (LLMs), together with graph neural networks and interpretable deep learning, has improved vulnerability detection accuracy; however, these methods are still primarily used for detection rather than for deterministic remediation [9], [35], [56].These approaches underperform in critical areas such as automated remediation, traceability, and lifecycle governance [36], [37], which are essential for production-grade reliability and regulatory compliance. Few existing detection pipelines integrate standard patching libraries like OpenZeppelin, and none provide fully automated workflows for repair and verification [38]. Even advanced smart-fuzzing and hybrid symbolic-semantic tools lack robust automated remediation [39], transparent justification models, and cross-contract traceability, rendering them inadequate for composable DeFi applications [40]. Comprehensive literature reviews from 2018 to 2025 highlight three persistent challenges: (i) scalable pre-deployment repair for both classical and novel DeFi vulnerabilities; (ii) standards-driven countermeasure synthesis using maintained security libraries; and (iii) integrated workflows supporting compositional, cross-contract, and business logic vulnerabilities in real-world large-scale deployments [41], [42]. To bridge these gaps, the Smart Contract Bug Fixing (SCBF) framework presents an integrated solution for vulnerability profiling, automated patch synthesis, and pre-deployment analytics. In its current implementation, SCBF relies on Mythril-based symbolic analysis for vulnerability detection and uses deterministic, OpenZeppelin-aligned

patching as part of a reproducible remediation workflow. The architecture also remains modular, so future versions can incorporate additional detection engines or verification components without changing the core remediation workflow.

3.1. Current Scope and Limitations

Although SCBF is designed as an automated smart contract remediation framework, the current implementation uses Mythril as the primary vulnerability detection engine for evaluating the remediation pipeline. This choice provides a consistent, reproducible detection baseline, particularly because Mythril reports vulnerabilities using established smart contract security classifications, such as SWC identifiers. In the present version, SCBF runs Mythril, maps the findings to the appropriate remediation route, applies deterministic patching logic, and records the resulting repair workflow for analysis and reporting. These results therefore evaluate SCBF as a smart contract pre-deployment, remediation-focused pipeline that uses Mythril as its detection backend. SCBF is not intended to function as a comprehensive vulnerability discovery platform; instead, the framework focuses on transforming detected smart contract security issues into structured, traceable, and reproducible remediation actions. This design also preserves architectural extensibility, as additional detection engines could be integrated in future versions to broaden vulnerability coverage and improve detection diversity. Second, the current repair workflow is more effective for deterministic and template-compatible vulnerability classes than for vulnerabilities requiring deeper semantic reasoning. Contracts containing multi-step business logic, interdependent vulnerabilities, or state-dependent behaviours remain more difficult to remediate automatically. In such cases, repairing one issue may interfere with another, or the correct fix may depend on contract-specificintent that cannot be fully recovered from pattern-based analysis alone. Third, SCBF's patch validation mechanism is currently based primarily on post-patch verification through recompilation and Mythril re-scanning under the same configuration. This provides a practical check that the originally targeted vulnerability is no longer reported. Still, it does not guarantee that the patched contract is semantically equivalent to the original in all execution paths. Accordingly, a successful re-scan should be interpreted as evidence of vulnerability removal under the applied analysis settings rather than as full proof of behavioural correctness. Finally, the present evaluation does not include comprehensive validation of functional correctness, such as exhaustive unit testing, invariant checking, or full formal verification for every patched contract. For this reason, the reported remediation results should be understood as validated security-oriented patch outcomes within the current detection-and-repair workflow, rather than as a guarantee that every generated patch preserves all intended contract functionality.

4. METHODOLOGY

This section presents the empirical evaluation workflow for assessing the SCBF remediation pipeline, including dataset preparation, vulnerability analysis, automated patch generation, and post-fix verification. The study adopts a quantitative empirical methodology and assesses SCBF across two public benchmark datasets by analysing detected vulnerabilities, generated patches, and post-patch verification outcomes. In the current implementation, the evaluated pipeline relies on Mythril for vulnerability detection, while SCBF performs SWC-based classification, deterministic fix routing, patch generation, re-scan verification, and result reporting. Detailed architectural descriptions are deferred to the Implementation section.

4.1. Experimental Workflow

The experimental process was conducted in a three-phase workflow: (1) environment setup and deployment of the SCBF tool in an isolated virtual machine, (2) batch analysis of smart contracts sourced from public datasets, and (3) log aggregation and analytics export through the web dashboard. For each contract file, the evaluated SCBF pipeline executed vulnerability detection using Mythril, mapped the resulting findings to SWC classifications, and, when applicable, routed supported issues to deterministic patch-generation modules aligned with OpenZeppelin-inspired remediation patterns. Post-patch validation and reporting, including CSV exports and database storage, were then used to support traceability and reproducibility.

4.2. Dataset Selection And Preparation

Evaluation was performed using two established smart contract benchmark datasets, covering both curated contracts and larger-scale contract collections. Each contract was pre-screened for Solidity compatibility before analysis. Benchmark-provided ground-truth vulnerability labels were obtained from the benchmark datasets and aligned with SCBF outputs using contract identifiers and vulnerability-category mappings where applicable. This enabled per-contract and per-category evaluation across common SWC-related vulnerability classes.

4.3. Evaluation Protocol

Contracts were processed in automated batches, with SCBF recording per-contract and per-vulnerability outcome metrics. In the current evaluation, Mythril served as the vulnerability detection engine, and SCBF was assessed for its ability to process Mythril findings via SWC-based classification, deterministic fix routing, patch generation, and post-patch verification. Fix rates were calculated globally and by SWC class. A vulnerability instance was considered fixed only when a patch was generated, and a subsequent Mythril rescan did not report the originally targeted issue. Instances without a supported repair routine or in which the issue persisted after verification were classified as unfixed. Pareto-style analysis was used to identify high-impact vulnerability classes and to guide refinement of patch templates and routing logic.

4.4. Reporting And Reproducibility

All outputs, including patched contract variants, detailed reports, and analytics summaries, were exported in structured formats such as CSV, SQL, and web dashboard visualisations. The SCBF toolchain and datasets were maintained under version control to support reproducibility. The workflow was designed to evaluate the current Mythril-driven SCBF pipeline independently from broader architectural extensions that may be incorporated in future versions.

5. IMPLEMENTATION

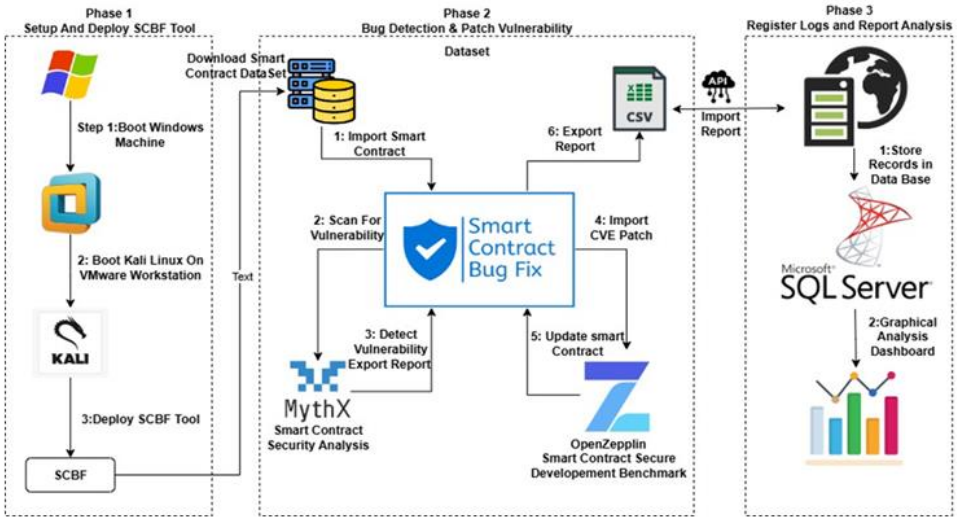

Figure 1.Overview of the SCBF operational workflow, showing environment preparation, Mythril-based vulnerability detection, deterministic patch generation using Open Zeppelin-aligned routines, and export of results to the database-backed analytics dashboard.

5.1. SCBF System End to End Pipeline

The SCBF framework is implemented as a modular end-to-end pipeline that connects contract ingestion, vulnerability detection, deterministic patch generation, verification, and reporting. The purpose of this design is to transform vulnerability analysis from an isolated detection step into a reproducible remediation workflow that can be executed in both single-contract and batch-processing modes. As illustrated in Fig. 1, the SCBF workflow is structured as a seven-stage pipeline comprising input validation, vulnerability scanning, SWC-based analysis, deterministic patch generation, post-patch verification, result storage, and dashboard-based analytics and reporting. In Phase 1, the analysis environment is prepared, and the SCBF tool chain is deployed in an isolated execution setting to support reproducibility. In Phase 2, SCBF processes input contracts, invokes Mythril for vulnerability detection, maps findings to SWC categories, and applies supported deterministic repair routines using Open Zeppelin-aligned templates. In Phase 3, scan outcomes, patch results, and verification artefacts are exported to persistent storage and made available through the analytics dashboard. The design rationale for this pipeline is to maintain a clear separation among detection, remediation, validation, and reporting. This separation allows SCBF to maintain traceability for each remediation step, support repeatable evaluation across datasets, and provide a structured basis for future extension of the framework.

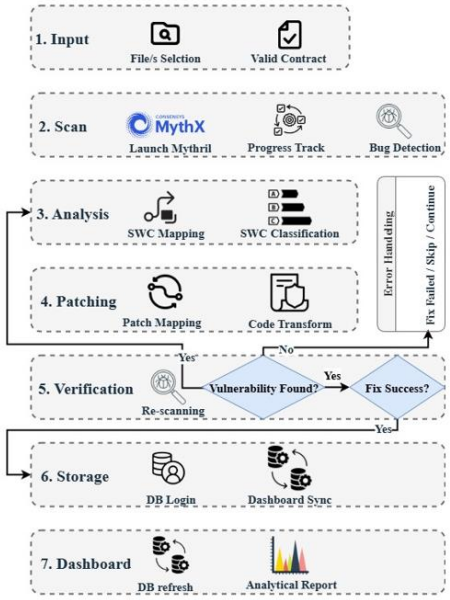

FIGURE 2: Layered SCBF workflow showing contract input, Mythril-based scan execution, SWC classification, patch mapping, code transformation, re-scan verification, result persistence, and dashboard reporting.

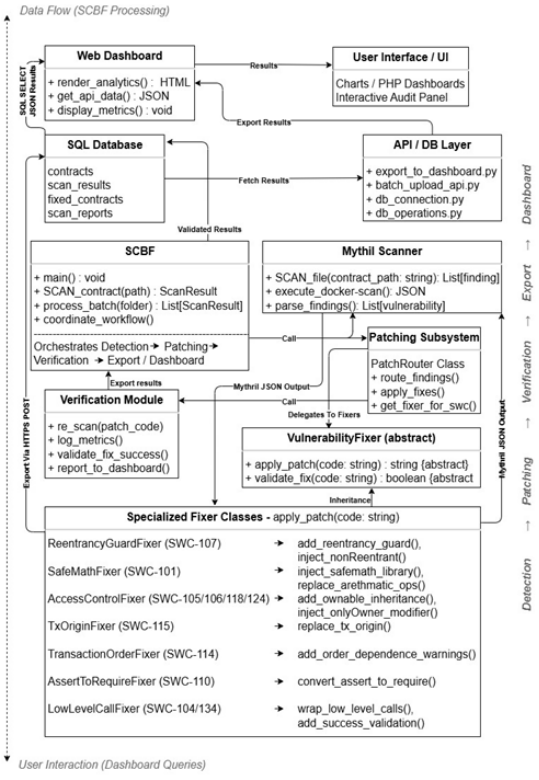

FIGURE 3: SCBF class-level architecture, highlighting the coordination between the Mythril scanner, patch routing logic, SWC-specific fix modules, verification stage, database layer, and dashboard interface used for reporting with analytics layer

5.2. System Architecture

The SCBF framework adopts a layered architecture to separate contract ingestion, vulnerability analysis, remediation, verification, persistence, and presentation, as illustrated in Fig. 2. This separation ensures that each stage of the remediation workflow remains independently traceable and extensible. Rather than treating vulnerability detection and patching as a single opaque process, SCBF decomposes the workflow into distinct stages so that findings can be classified, matched to deterministic repair routines, re-validated, and then exported for later inspection. At the core of this architecture, Mythril functions as the active detection engine in the current implementation. Its findings are normalised into SWC-based categories and passed to the patching layer, where supported vulnerabilities are mapped to predefined remediation routines. A dedicated verification stage then re-scans patched contracts to determine whether the originally targeted issue is still reported. Finally, validated outputs, metadata, and performance records are stored in the backend and exposed through the dashboard layer. This layered design supports reproducibility, structured reporting, and future extensibility without changing the current evaluated pipeline. The detection layer invokes Mythril in a controlled execution environment to analyse each contract and generate vulnerability findings. The analysis layer then maps these findings to SWC categories and prepares them for remediation. In the patching layer, SCBF applies supported deterministic repair routines, including OpenZeppelin-aligned countermeasures where appropriate, to generate modified contract variants. The verification layer subsequently re-scans patched contracts to check whether the originally targeted issue is still reported under the same analysis configuration. The storage and dashboard layers preserve the resulting artefacts and provide access to scan summaries, patch outcomes, and associated metrics for later review.

5.3. Core Class Structure and Major Module Functions

The system's core architecture is depicted in the class diagram shown in Fig. 3. This diagram illustrates the main structural elements and object interactions that constitute the SCBF platform for automated smart contract vulnerability detection and patching. The figure illustrates the complete SCBF data flow and class hierarchy. The left annotation represents the bottom-to-top automated data flow (SCBF Processing) and the top-to-bottom user interaction path (Dashboard Queries). The right-hand labels identify the system's layered architecture from Detection to Dashboard, showing how the SCBF engine integrates Mythril scanning, patching via OpenZeppelin countermeasures, verification, and final dashboard visualisation. Major components include the central SCBF coordinator, the Mythril scanner scanning engine, the *PatchRouter* vulnerability patching system and its fixer function, the Database Manager, and the web dashboard for analytics. Each component interacts via programmatic interfaces, with the *PatchRouter* dynamically routing each vulnerability finding to the appropriate fixer module.

5.3.1. Scanner and Vulnerability Finding

The Mythril scanner class is devoted to automated vulnerability detection using static and symbolic analysis. Its interface offers methods for launching contract file scans, optionally inside Docker sandboxes, and parsing vulnerability findings into structured outputs. The central controller invokes the scanner and outputs a list of findings, each keyed by SWC-ID and including metadata such as description, explanation, severity, transaction sequence, affected function, line number, estimated gas usage, and memory consumption information. These structured findings are then passed directly to the patching subsystem for SWC-based routing and remediation. In the user interface, the findings tab displays the identified vulnerabilities, severity levels, affected functions, SWC identifiers, and descriptive metadata, which are then used to support SWC-based routing and remediation.

5.3.2. Patch Router

Receives vulnerability findings and applies appropriate fixes by invoking external modules based on SWC tags. The *PatchRouter* function pairs each detected vulnerability with its corresponding SWC number, then iteratively calls the relevant fix function until all vulnerabilities are fixed. Additional remediation routines can be added to the *PatchRouter* by extending the mapping from supported SWC categories to deterministic, OpenZeppelin-aligned patch modules.

5.3.3. Vulnerability Fixer

Defines the abstract structure for patch application and validation, inherited by each concrete fixer class, such as *AccessControlFixer* and *ReentrancyGuardFixer*. The Vulnerability fixer mechanism calls the relevant fix function and applies the fix to an instance if an assert-violation vulnerability is detected. To patch the access control vulnerability, the Vulnerability fixer function applies the access control patch from the OpenZeppelin library. The repair routine adds the required OpenZeppelin Own able dependency when applying the access-control patch, while avoiding duplicate dependency insertion during repeated scan-and-fix cycles. The fixer then extends the contract with ownership semantics and attaches the *only Owner* modifier to state-changing functions that previously lacked explicit access protection. This patch targets access-control weaknesses associated with SWC-105 and SWC-136[43], [44].

5.3.4. Database Manager And Web Dashboard

The Database manager handles storage and retrieval of scan, patch, and reporting data. Further, the web dashboard generates analytics, API feeds, and metrics visualisations for contract batches from the stored data.

6. RESULTS AND PERFORMANCE ANALYSIS

This section presents the empirical evaluation of the SmartContract Bug Fix (SCBF) framework across two independent datasets: the *SmartBugs* Curated [45] and the *Messi-Q* SmartContract Dataset [46]. The analysis quantifies SCBF detection accuracy, automated patch success, and vulnerability coverage per category.

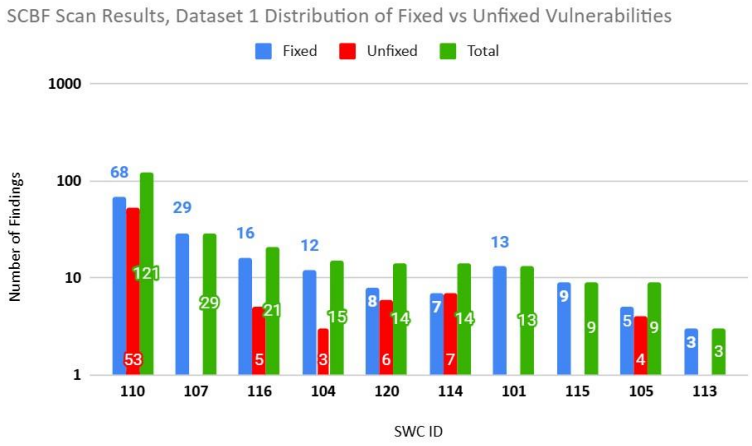

FIGURE 4: Dataset 1 Results. Distribution of fixed vs unfixed vulnerabilities across SWC *fixed.sol*classes.

6.1. Dataset 1, SMARTBUGS Curated benchmark

Fig. 4 illustrates the results of the SCBF scan in dataset 1 [45], which comprised 269 Solidity contracts with 340 identified vulnerabilities across ten SWC classes. Under the findings-based counting rule adopted in this paper, SCBF generated248 Mythril findings for Dataset 1, of which 170 (68.5%)were successfully patched, and 78 (31.5%) remained unfixed.

6.2. Dataset 2, MESSI-Q Large Scale Evaluation

Dataset 2 contained 2,217 original contracts and yielded 1,239 vulnerability findings across 12 SWC categories. In the previous experimental iteration, SCBF automatically generated 295 *fixed.sol* variants representing distinct instances in which the system successfully synthesised and validated patched contracts. Subsequently, these results were verified and included in the total fix count, increasing the cumulative number of successful repairs. After incorporating these 295 patches, the final results show that SCBF achieved 958 successful fixes out of 1,239 findings (77.3%). In contrast, the remaining 281 findings (22.7%) were classified as unfixed because no verifiable patch could be applied and validated under the configured settings. Fig. 5visualises this distribution across the SWC classes, distinguishing the total number of detected vulnerabilities (green), those fixed (blue), and those unfixed (red).

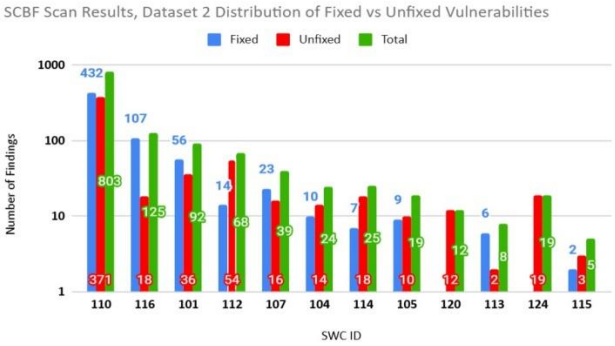

FIGURE 5: Dataset 2 Results. SCBF per-SWC distribution of vulnerability detection, fixed, and unfixed instances.

For reporting purposes, the 77.3% figure represents the consolidated proportion of Mythril-reported findings that were remediated and successfully cleared during post-patch verification. In this calculation, the unit of analysis is the Mythril-reported vulnerability finding. A finding was counted as fixed when SCBF generated a patched contract variant, and the targeted Mythril-reported issue was no longer reported during post-patch verification under the same configuration. In addition to this findings-level analysis, a stricter contract-level diagnostic validation was performed against the benchmark-provided ground-truth annotations. This secondary validation used benchmark-labelled vulnerable contracts as the unit of analysis and compared normalised original and patched contract pairs to determine whether verifiable structural or semantic code changes had occurred. Under this stricter validation, no confirmed semantic repairs were observed in the examined subset of ground-truth-positive contracts. This does not contradict the findings-level fix rate; rather, it shows that the 77.3% result reflects SCBF's remediation of Mythril-emitted findings, whereas the contract-level audit evaluates a narrower, stricter semantic validation criterion. These findings highlight the need for improved handling of multi-instance repairs, deeper configuration of symbolic execution, and broader semantic validation in future SCBF iterations.

6.3. Evaluation Metrics And Performance Analysis

Each SCBF result was systematically aligned with benchmark-provided ground-truth vulnerability annotations derived from *vulnerabilities. json* within the *SmartBugs* and *Messi-Q* benchmark suites. The benchmark annotations provided an external reference point for organising SCBF outputs by vulnerability category and for checking whether Mythril-reported findings corresponded to recognised benchmark-labelled weaknesses. A Mythril-reported finding was treated as successfully fixed when SCBF generated a patched contract variant, and the targeted issue was no longer reported during post-patch verification under the same configuration. Findings that lacked a supported repair routine, failed patch generation, or remained detectable after re-scanning were classified as unfixed. This mapping enabled reproducible calculation of precision, recall, F1-score, failure rate, and residual rate across the selected SWC categories. The stricter contract-level semantic validation described in Section 6.2 was treated as a separate diagnostic analysis and was not used to override the findings-level metrics reported in Table 1. To assess whether the performance differences between *SmartBugs* (D1) and *Messi-Q* (D2) were statistically significant, we compared matched SWC-level F1 scores across the SWC categories reported in Table 1.

TABLE 1: Benchmark-referenced detection metric values used for Fig. 6 and Fig. 7 (D1=*SmartBugs*, D2=*Messi-Q*).

Metric	**Dataset-level pooled D1D2**		**SWC-110**		**SWC-116**		**SWC-107**		**SWC-104**	
			D1	**D2**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Precision	0.78	0.69	0.72	0.67	0.85	0.90	0.95	0.75	0.82	0.62
Recall	0.685	0.538	0.562	0.538	0.762	0.856	1.00	0.59	0.80	0.417
F_1	0.729	0.604	0.632	0.596	0.804	0.878	0.974	0.66	0.81	0.497
Failure (1−*P*)	0.22	0.31	0.28	0.33	0.15	0.10	0.05	0.25	0.18	0.38
Residual (1−*R*)	0.315	0.462	0.438	0.462	0.238	0.144	0.00	0.41	0.20	0.583

Because the same SWC classes were evaluated in both datasets and the comparison therefore involved paired continuous summary scores, a paired t-test was used as an exploratory parametric test to compare mean F1 performance across datasets. The mean paired F1 difference was 0.147, with no statistically significant difference observed ($t(3) = 1.49$, $p = 0.232$). To complement the hypothesis test, we also report Cohen's $dz = 0.75$, indicating a moderate-to-large effect size in practical terms [47]. Although the dataset-level variation did not reach the 0.05 significance threshold, the observed effect size indicates a practically meaningful difference that should be considered, given the limited number of matched SWC categories. To support interpretation, radar and grouped bar visualisations were used to compare five key metrics: Precision, Recall, F1-score, Failure (1−P), and Residual (1−R). The core metrics are defined as follows:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN} F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Where TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives. These metrics quantify SCBF's detection accuracy, fix effectiveness, and residual defect rate across the benchmark datasets *SmartBugs* (D1) and *Messi-Q* (D2). This approach provides both geometric and numerical perspectives: the radar chart highlights harmony and equilibrium among related metrics, while the grouped bar chart reveals magnitudes and dataset-specific trends across vulnerability classes (SWCs). The radar chart (Fig. 6) provides insight into SCBF's metric balance and structural robustness.

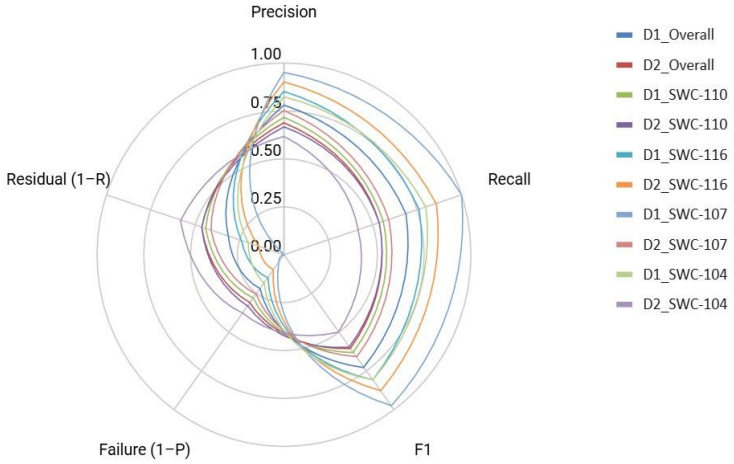

FIGURE 6. Radar chart of SCBF performance metrics across datasets.

Each axis represents a distinct performance measure, and the resulting polygonal shapes aggregate performance behaviours across datasets. Near-regular polygons indicate balanced metric harmony, demonstrating that SCBF sustains consistent fix selectivity and completeness over multiple vulnerability types. Asymmetric or irregular shapes, by contrast, reveal local tensions between precision and recall or localised instability. Notably, a larger, more regular polygon corresponds to higher aggregate robustness and greater metric uniformity, with *SmartBugs* (D1) exhibiting greater stability than *MessiQ* (D2), where increased variance stems from diverse contract structures. The grouped bar chart (Fig. 7) complements this analysis by mapping metric magnitudes side by side for each dataset and SWC category.

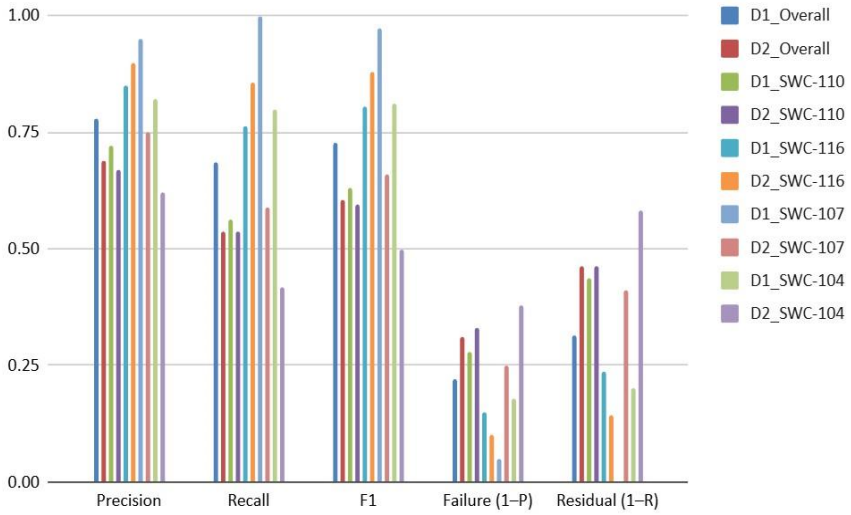

FIGURE 7: Grouped bar chart of SCBF detection metrics by dataset and SWC.

SmartBugs (D1) achieves a higher F1-score and lower residual rates (1 − R), thereby indicating reliable detection and repair validation. *MessiQ* (D2) yields higher Precision but lower Recall and increased dispersion across SWCs, suggesting selective optimisation under heterogeneous contract structures. The bar chart thus provides a granular quantitative validation of the harmonic relationships observed in the radar chart. Collectively, these visualisations demonstrate that SCBF achieves superior metric harmony, consistency, and defect minimisation in *SmartBugs*, while scenario-specific precision gains in *Messi-Q* highlight sensitivity to the dataset. This

integrated geometric-numeric analysis aligns with IEEE standards for multidimensional performance reporting, thereby ensuring both interpretability and analytical rigour. SCBF is designed as a modular automation pipeline that integrates Mythril for vulnerability detection and OpenZeppelin for automated patch synthesis. The set of detected, fixed, and unfixed vulnerabilities primarily reflects the current Mythril-driven detection configuration and the repair routines implemented in this version, rather than a fixed limitation of SCBF's architecture. Patches are applied only when Mythril emits a corresponding finding, a supported repair routine is available, and post-patch verification succeeds. Therefore, the absence of a patch may indicate that Mythril did not report the issue that no supported repair routine was available, or that verification failed after patching. These cases should be interpreted as current implementation boundaries rather than architectural limitations, since SCBF's modular design allows additional countermeasure modules, SWC-specific repair routines, and complementary analysis engines to be integrated in future versions. This design positions SCBF as an extensible remediation platform that can expand detection coverage and improve automated fix generation over time. SCBF's architecture adheres to established software extensibility and modularity standards [48], facilitating the future integration of alternative analysis modules, such as Slither, to expand detection coverage and improve corresponding fix generation. Accordingly, the framework's methodological soundness derives from its capacity to support upgrades, validation within current tool constraints, and compliance with contemporary software engineering standards.

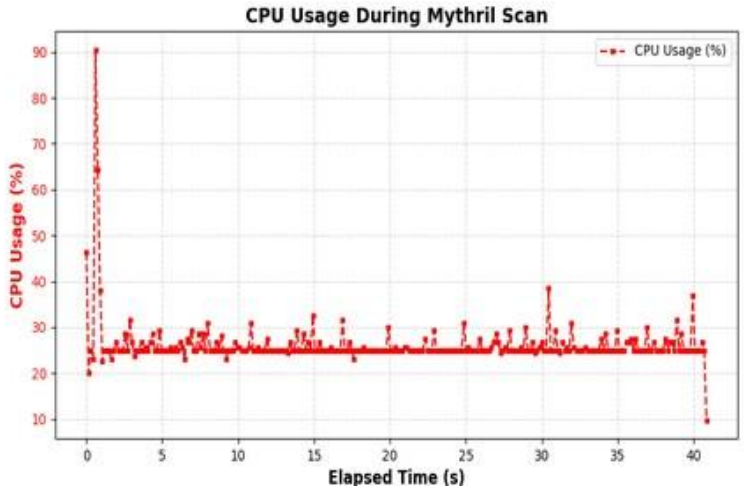

FIGURE 8: CPU usage during a Mythril contract scan in SCBF

.

TABLE 2: Experimental Host and Virtual Machine Configuration

Component	**Specification**
Host OS	Windows 11 Pro (25H2)
Host CPU	Intel Core i9-13900HX
Host RAM	32 GB
Hypervisor	VMware Workstation
Guest OS	Kali Linux Rolling (2024.4)
VM CPU Allocation	4 vCPUs
VM RAM Allocation	4 GB
VM Storage Allocation	80 GB

6.4. Resource Utilisation and Runtime Profiling

All runtime measurements were collected in the controlled host and virtual-machine environment and are summarised in Table 2. This configuration provided a consistent and reproducible basis for evaluating performance and resource behaviour. To assess SCBF's runtime efficiency and stability, empirical profiling was conducted across 383 contract scans. Most scans were completed within a short execution window, while a small number of contracts required substantially longer analysis times because symbolic execution complexity varied with contract structure, branching behaviour, and path depth. This runtime spread is expected in Mythril-driven symbolic analysis, where contracts with more complex execution paths may require additional solving time. Memory utilisation remained effectively stable across the scan set, indicating that the SCBF wrapper released residual Mythril allocations after each scan and avoided cumulative memory growth during batch processing. Collectively, these results show that SCBF maintained a stable memory footprint and practical runtime behaviour under depth-7 symbolic execution, supporting its suitability for large-scale batch analysis. For illustration, Fig. 8 shows the CPU usage profile for a representative contract scan; similar behaviour was observed across the dataset.

TABLE 3: Summary of SCBF results across two public datasets under the reporting scheme adopted in this paper (Dataset 1: findings-based; Dataset 2: consolidated validated totals).

Metric	**Dataset 1 - SmartBugs Curated**	**Dataset 2 Messi-Q**
Files Analysed	269	2,217
Mythril Findings	248	1,239
Fixed	170 (68.5%)	958 (77.3%)
Unfixed	78 (31.5%)	281 (22.7%)
Dominant SWC Class	SWC-110 (121)	SWC-110 (803)
Highest Fix Rate	SWC-107/113 (100%)	Consolidated result reported atdataset level
Lowest Fix Rate	SWC-114 (50.0%)	Consolidated result reported at dataset level

6.5. Comparative Evaluation

Table 3 summarises the key evaluation metrics across both datasets. Under the reporting scheme adopted in this paper, SCBF achieved a findings-based fix rate of 68.5% in Dataset 1 and a consolidated fix rate of 77.3% in Dataset 2. These values should be interpreted in their respective reporting contexts: Dataset 1 reflects findings-based patch outcomes from the current *SmartBugs* Curated run, whereas Dataset 2 reports consolidated, validated outcomes that incorporate previously verified _fixed.sol artefacts under the carryover-aware de-duplication rule. Accordingly, Dataset 1 provides a curated benchmark view, while Dataset 2 reflects a larger-scale consolidated evaluation under more heterogeneous contract conditions. Table 4 presents the per-SWC distribution of fixed and unfixed instances across both benchmark datasets. The results show clear variation in patchability across vulnerability classes. Deterministic and template-compatible classes, such as SWC-107 and SWC-101, achieved consistently high fix rates. In contrast, vulnerabilities such as SWC-110 and SWC-112, representing Exception State and Delegate-call to User-Supplied Address [49]–[51], showed lower or less consistent remediation performance because they often require deeper semantic or contextual reasoning. Dataset 2 also incorporates 295 previously validated _fixed.sol artefacts, reflecting the cumulative effect of prior automated remediation. This breakdown confirms SCBF's strength in pattern-based fixes

while identifying vulnerability classes that require future enhancement through expanded repair templates, improved detection coverage, and deeper semantic patching strategies.

TABLE 4: Per-SWC vulnerability distribution across both datasets

SWC	**Vulnerability**	**Dataset 1**			**Dataset 2**			**Fix Rate (1/2) %**
		Fixed	Unfixed	Total	Fixed	Unfixed	Total	
SWC-110	Exception State	98	83	181	432	371	803	54.1 / 53.8
SWC-107	Re-entrancy	29	0	29	39	0	39	100 / 100
SWC-104	Unchecked Return Value from External Call	18	9	27	20	4	24	66.7 / 83.3
SWC-116	Dependence on Predictable Environment Variable	18	7	25	17	108	125	72.0 / 13.6
SWC-114	Transaction Order Dependence	11	11	22	14	11	25	50.0 / 56.0
SWC-120	Dependence on Predictable Environment Variable	10	8	18	0	0	0	55.6 / 0.0
SWC-101	Integer Arithmetic Bugs	13	0	13	92	0	92	100 / 100
SWC-112	*Delegate-call* to User-Supplied Address	0	0	0	14	54	68	0 / 20.6
SWC-105	Unprotected Ether Withdrawal	7	6	13	14	5	19	53.8 / 73.7
SWC-115	Dependence on tx.origin	9	0	9	5	0	5	100 / 100
SWC-113	Multiple Calls in a Single Transaction	3	0	3	8	0	8	100 / 100
SWC-123	Requirement Violation	0	0	0	1	11	12	0 / 8.3
SWC-124	Write to Arbitrary Storage Location	0	0	0	7	12	19	0 / 36.8

6.6. Comparison With Existing Tools

To contextualise SCBF's contribution, Fig. 9 compares the framework with representative smart contract analysis and repair tools across key operational dimensions, including detection coverage, remediation support, workflow integration, reporting capability, compliance alignment, maintainability, extensibility, and explain ability. The comparison is based on a structured qualitative assessment of the primary literature, available tool documentation, project repositories, and reported workflow characteristics. The ordinal scores used in the radar visualisation are therefore literature-grounded comparative judgements rather than direct experimental measurements obtained from a single unified benchmark. Existing tools such as Oyente, Mythril, Slither, Securify, SmartCheck, ContractWard, and MadMax remain valuable for vulnerability discovery, but they primarily operate as detection-oriented systems and generally require separate manual or external remediation workflows. Bytecode-level repair approaches, such as SmartShield, demonstrate that automated patching is feasible, although they offer reduced source-level transparency and a narrower scope for repairs. Template-based, ML-based, and LLM-based approaches extend automation in different ways, but their reproducibility,

standards alignment, explain ability, and operational reporting capabilities vary across implementations. In contrast, SCBF connects Mythril-based vulnerability detection with SWC-guided patch routing, deterministic OpenZeppelin-aligned repair templates, post-patch verification, and dashboard-based reporting. This positions SCBF as a reproducible remediation workflow rather than a detection-only analyser. The purpose of the comparison is therefore not to claim universal superiority over existing tools, but to show how SCBF addresses the detection-remediation gap by integrating automated repair, verification, and traceable reporting within a modular framework that can be extended with additional analysis and verification tools in future work.

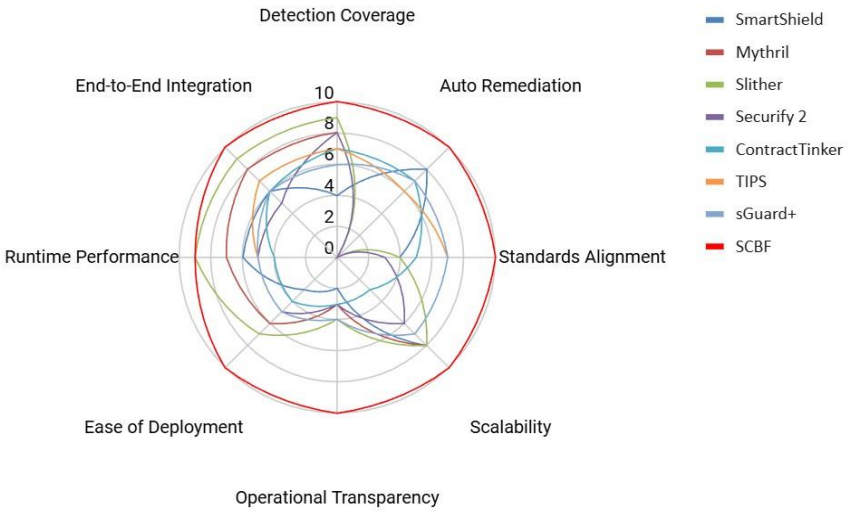

FIGURE 9: Multi-tool capability radar across eight operational dimensions.

6.7. Discussion

Across both benchmark evaluations, SCBF consistently demonstrated robust detection coverage, automated remediation capability, and reproducible fix behaviour, particularly for deterministic vulnerability types such as re-entrancy and arithmetic-related vulnerabilities. The framework maintained strong performance across heterogeneous datasets, with empirical fix rates of 68.5% for *SmartBugs* Curated and 77.3% for *Messi-Q* under the findings-based and consolidated reporting scheme adopted in this paper. These results indicate that SCBF's class-specific repair capability is strongest for SWC patterns with established remediation templates. In contrast, more complex, state-dependent, or inter-procedural vulnerabilities, such as transaction-order dependence and *delegate-call* misuse, remain challenging to address. The system's integrated pipeline, from Mythril-based detection through patching, post-patch verification, and dashboard-driven reporting, supports a traceable remediation workflow rather than detection alone. Comparative analysis against leading academic and industrial tools further highlights SCBF's position as a remediation-oriented framework that connects vulnerability detection to deterministic patch generation and structured reporting. This distinguishes SCBF from detection-only analysers and provides a practical basis for extending broader tool support and vulnerability coverage. However, the results should be interpreted within the current scope and limitations of the framework. SCBF currently relies on Mythril as its primary vulnerability detection engine, so patching is attempted only when Mythril reports a supported finding. Therefore, unfixed cases may arise because Mythril did not detect the vulnerability, no matching repair routine was available, or the generated patch failed post-patch verification. In addition, the current validation process is based primarily on recompilation and Mythril re-scanning under the same

configuration. This provides practical evidence that the targeted issue is no longer reported, but it does not constitute full proof of semantic equivalence or functional correctness across all execution paths. Persistent challenges with certain vulnerability classes indicate that future SCBF development should focus on deeper semantic trace reasoning, stronger context modelling, broader verification support, and explainability-aware security analysis [57]. In particular, future work should integrate additional detection and verification engines, such as Slither and Manticore, expand SWC-specific and DeFi-oriented repair templates, and incorporate stronger correctness checks, including unit testing, invariant checking, differential testing, and formal verification where applicable. Collectively, these findings position SCBF as a generalizable, reproducible, and auditable framework for smart contract security remediation, while also clarifying the main limitations and future directions needed to strengthen its use in research and production environments.

7. CONCLUSION

This work introduced SCBF as a framework for automated smart contract vulnerability remediation, connecting Mythril-based detection, SWC-guided fix routing, deterministic patch generation, verification, and structured reporting into a reproducible workflow. In its current implementation, SCBF addresses the gap between vulnerability identification and actionable repair for selected classes of Ethereum smart contract vulnerabilities by applying deterministic, OpenZeppelin-aligned remediation patterns to supported Mythril findings. Empirical evaluation on two public benchmark datasets indicates that SCBF can support automated remediation at scale while maintaining traceability and audit-oriented reporting. Using the counting scheme adopted in this paper, SCBF achieved a fixed rate of 68.5% (170/248 findings) on *SmartBugs* Curated and a consolidated fix rate of 77.3% (958/1,239 findings) on *Messi-Q*. These results indicate that the framework is particularly effective for vulnerability classes amenable to deterministic, template-based remediation, whereas more complex semantic, state-dependent, and interdependent cases remain challenging. Compared with detection-oriented tools, SCBF provides a more integrated remediation workflow by combining deterministic patch generation, post-fix verification, dashboard-based reporting, forensic traceability, and CSV, JSON, and SQL exports. Overall, SCBF provides a reproducible framework for automated smart contract remediation with source-level, standards-aligned repairs for supported vulnerability classes. Its modular architecture and deployment workflow provide a practical basis for future extension toward broader vulnerability coverage and additional security and verification tools.

8. FUTURE WORK

The SCBF framework demonstrates substantial progress in automated detection and patching of smart contract vulnerabilities, attaining a consolidated fix rate of 77.3% on the *Messi-Q* dataset and 68.5% on the *SmartBugs* Curated dataset under the counting scheme adopted in this paper. Nevertheless, several research challenges remain, offering opportunities to advance automated remediation at scale.

8.1. Semantic and Complex Vulnerability Classes

While SCBF effectively addresses deterministic vulnerability classes, semantic and non-deterministic vulnerabilities such as transaction-order dependence, *delegate-call* misuse, and complex re-entrancy scenarios require further investigation. In particular, integrating additional analysis and verification tools, such as Slither and Manticore [52], as future SCBF modules will strengthen the detection and remediation of *delegate-call* vulnerabilities. Automated patching strategies that leverage OpenZeppelin standards can be extended to cover SWC-112 and related

patterns, enabling more comprehensive and proactive protection against these critical attack vectors [20], [53].

8.2. Comparative Benchmarking and Competitive Analysis

Future work will conduct an in-depth comparative study with related frameworks, such as Elysium [10] and SoliAudit [11], to benchmark and refine SCBF's automated repair logic, patch determinism, verification workflow, and adherence to security standards. This evaluation will extend the qualitative comparison shown in Fig. 9 by incorporating additional emerging tools and, where feasible, controlled benchmark experiments across shared datasets and reporting criteria.

REFERENCES

[1] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," IEEE Transactions on Network Science and Engineering, vol. 8, no. 2, pp. 1133–1144, 2020.

[2] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[3] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 67–82.

[4] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020, pp. 23–34.

[5] R. Kiani and V. S. Sheng, "Automated repair of smart contract vulnerabilities: A systematic literature review," Electronics, vol. 13, no. 19, p. 3942, 2024.

[6] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, "sguard+: Machine learning guided rule-based automated vulnerability repair on smart contracts," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 5, pp. 1–55, 2024.

[7] S. A. Amri, L. Aniello, and V. Sassone, "A review of upgradeable smart contract patterns based on openzeppelin technique," The Journal of The British Blockchain Association, 2023.

[8] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE). ACM, 2020, pp. 530–541.

[9] J. Feist, G. Grieco, and A. Groce, "Mythril: Security analysis of ethereum smart contracts," in Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE/ACM, 2018.

[10] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, 2022, pp. 115–128.

[11] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing," in 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, 2019, pp. 458–465.

[12] H. Liu, D. Wu, Y. Sun, H. Wang, K. Li, Y. Liu, and Y. Chen, "Using my functions should follow my checks: understanding and detecting insecure openzeppelin code in smart contracts," in 33rd USENIX Security Symposium (USENIX Security 24), USENIX Association, 2024, pp. 3585–3601.

[13] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," arXiv preprint arXiv:1812.05934, 2018.

[14] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019, pp. 8–15.

[15] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in Proceedings of the 35th IEEE/ACM international conference on automated software engineering, 2020, pp. 1349–1352.
[16] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in Proceedings of the 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), 2018, pp. 9–16.
[17] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects," IEEE Transactions on Software Engineering, vol. 50, no. 6, pp. 1360–1373, 2024.
[18] C. Wang, J. Zhang, J. Gao, L. Xia, Z. Guan, and Z. Chen, "Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts," in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 2350–2353.
[19] G. Iuliano and D. Di Nucci, "Smart contract vulnerabilities, tools, and benchmarks: An updated systematic literature review," arXiv preprint arXiv:2412.01719, 2024.
[20] S. Bobadilla, M. Jin, and M. Monperrus, "Do automated fixes truly mitigate smart contract exploits?" IEEE Transactions on Software Engineering, 2025.
[21] P. Fang, P. Gao, Y. Peng, Q. Zhang, T. Xie, D. Song, P. Mittal, S. R. Kulkarni, Z. Liu, and X. Xiao, "Contractfix: A framework for automatically fixing vulnerabilities in smart contracts," arXiv preprint arXiv:2307.08912, 2023.
[22] Y. Wang, S. Sheng, and Y. Wang, "A systematic literature review on smart contract vulnerability detection by symbolic execution," in International Conference on Blockchain and Trustworthy Systems. Springer, 2024, pp. 226–241.
[23] Darvishi, I., Asare, B.T., Musa, A., Yeboah-Ofori, A., Oseni, W. and Ganiyu, A., 2024, August. Blockchain technology and vulnerability exploits on smart contracts. In 2024 11th International Conference on Future Internet of Things and Cloud (FiCloud) (pp. 160-167). IEEE.
[24] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and defi security tools: Do they meet the needs of practitioners?" in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
[25] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 2, pp. 515–529, 2024.
[26] B. Jia, X. Zhang, X. Zhang, T. Chen, and W. Lian, "Enhancing security and acuity of smart contract vulnerability detection based on federated learning and bilstm-attention," ACM Trans. Softw. Eng. Methodol., Jun. 2025, just Accepted.
[27] G. A. D. Malavolta, P. Moreno-Sanchez, A. Kate, and K. G. Paterson, "SoK: Communication Across Distributed Ledgers," in 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 414–433.
[28] C. D. Baets, B. Suleiman, A. Chitizadeh, and I. Razzak, "Vulnerability detection in smart contracts: A comprehensive survey." [Online].Available: http://arxiv.org/abs/2407.07922
[29] A. Arusoaie and S.-C. Susan, "Towards trusted smart contracts: A comprehensive test suite for vulnerability detection," Empirical Software Engineering, vol. 29, no. 5, p. 117, 2024.
[30] I. Ashraf and W. Chant, "An empirical study on the effects of entry function pairs in fuzzing smart contracts," in 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2022, pp. 1716–1721.
[31] P. Azad, C. Akcora, and A. Khan, "Machine learning for blockchain data analysis: Progress and opportunities," Distrib. Ledger Technol., Apr. 2025, just Accepted.
[32] R. Gupta, S. Tanwar, F. Al-Turjman, P. Italiya, A. Nauman, and S. W. Kim, "Smart contract privacy protection using ai in cyber-physical systems: tools, techniques and challenges," IEEE Access, vol. 8, pp. 24746–24772, 2020.
[33] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," IEEE Access, vol. 8, pp. 99552–99564, 2020.
[34] J. Chen, Z. Shao, S. Yang, Y. Shen, Y. Wang, T. Chen, Z. Shan, and Z. Zheng, "Numscout: Unveiling numerical defects in smart contracts using llm-pruning symbolic execution," IEEE Transactions on Software Engineering, vol. 51, no. 5, pp. 1538–1553, 2025.

[35] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," IEEE Transactions on Knowledge and Data Engineering, vol. 35, no. 2, pp. 1296–1310, 2023.

[36] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in IJCAI, 2020, pp. 3283–3290.

[37] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in IJCAI, 2021, pp. 2751–2759.

[38] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 295–306.

[39] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," arXiv preprint arXiv:2301.03943, 2023.

[40] B. Wang, X. Yuan, L. Duan, H. Ma, B. Wang, C. Su, and W. Wang,"Defiscanner: Spotting defi attacks exploiting logic vulnerabilities on blockchain," IEEE Transactions on Computational Social Systems, vol. 11, no. 2, pp. 1577–1588, 2024.

[41] Y. Feng, E. Torlak, and R. Bodik, "Summary-based symbolic evaluation for smart contracts," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1141–1152.

[42] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 227–239.

[43] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 945–956.

[44] Z. A. Khan and A. S. Namin, "A survey of vulnerability detection techniques by smart contract tools," IEEE Access, vol. 12, pp. 70870–70910, 2024.

[45] C. Torres, S. Bhattacharya, A. Zumberi, A. Pinna, R. Díaz, and G. Roig, "Smartbugs: A framework to analyze solidity smart contracts," in 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE, 2021, pp. 57–66, dataset available at https://github.com/smartbugs/ smartbugs-curated. [Online]. Available: https://github.com/smartbugs/ smartbugs-curated

[46] P. Qian, Z. Liu, Y. Yin, and Q. He, "Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode," in Proceedings of the ACM web conference 2023, 2023, pp. 2220–2229.

[47] J. Cohen, Statistical power analysis for the behavioral sciences. routledge, 2013.

[48] I. O. for Standardization, ISO/IEC 25024: 2015: Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-Measurement of Data Quality. ISO/IEC, 2015.

[49] C. Ruggiero, P. Mazzini, E. Coppa, S. Lenti, and S. Bonomi, "Sok: a unified data model for smart contract vulnerability taxonomies," in Proceedings of the 19th International Conference on Availability, Reliability and Security, 2024, pp. 1–13.

[50] M. Di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Evolution of automated weakness detection in ethereum bytecode: a comprehensive study," Empirical Software Engineering, vol. 29, no. 2, p. 41, 2024.

[51] F. R. Vidal, N. Ivaki, and N. Laranjeiro, "Vulnerability detection techniques for smart contracts: A systematic literature review," Journal of Systems and Software, vol. 217, p. 112160, 2024.

[52] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1186–1189.

[53] C. Sendner, L. Petzi, J. Stang, and A. Dmitrienko, "Vulnerability scanners for ethereum smart contracts: A large-scale study," arXiv preprint arXiv:2312.16533, 2023.

[54] N. H. Le Viet, L. Phan, H. Ngo Van, and T. Trinh Quang, "Deep learning solutions for Source code vulnerability detection," Int. J. Netw. Secur. Its Appl., vol. 17, no. 3, pp. 33–44, May 2025, doi: 10.5121/ijnsa.2025.17303.

[55] S. Alajmani, E. Aljuaid, B. Soh, and R. Y. Alyami, "Enhancing Malware Detection and Analysis using Deep Learning and Explainable AI (XAI)," Int. J. Netw. Secur. Its Appl., vol. 17, no. 2, pp. 01–19, Mar. 2025, doi: 10.5121/ijnsa.2025.17201.
[56] J. Selasi Agbesi, A. Nanayaa Otchill, R. Horlalie Tay, and N. K. Bamfo, "Machine learning for network intrusion detection in usa critical infrastructure: challenges and opportunities ," Int. J. Netw. Secur. Its Appl., vol. 18, no. 1, pp. 55–73, Jan. 2026, doi: 10.5121/ijnsa.2026.18104.
[57] A. Mater Aljohani and I. Elgendi, "The interest of hybridizing explainable ai with rnn to resolve ddos attacks: a comprehensive practical study," Int. J. Netw. Secur. Its Appl., vol. 16, no. 2, pp. 65–83, Mar. 2024, doi: 10.5121/ijnsa.2024.16205.

AUTHORS

Iman Darvishi is a cybersecurity researcher whose work focuses on smart contract security, blockchain-based authentication, Internet of Things (IoT) security, and secure web application development. His research explores vulnerability detection and automated remediation in smart contracts, decentralised access control, and the integration of blockchain technologies with cyber-physical systems. He has designed and implemented access control systems involving blockchain smart contracts and web application platforms for IoT-based access control. He has also developed an automated framework for vulnerability scanning and remediation to audit blockchain security. His broader research interests include web application security, penetration testing, privacy-preserving authentication, and blockchain-enabled protection for connected and autonomous environments.

Dr Alireza Esfahani (Member, IEEE) is currently a Senior Lecturer in cyber security at the University of West London, U.K. Previously, he was a Research Associate with the SnT-CritiX Group at the University of Luxembourg. He made significant contributions to major EU-funded cybersecurity pilot projects, including SPARTA and CyberSec4EU. He has authored more than 50 peer-reviewed scientific publications in the field of cybersecurity. He has also served as the Principal Investigator for two UKRI-funded CyberASAP projects. His primary research interests include software-defined networking (SDN) and open RAN (ORAN) technologies, with a particular focus on privacy-preserving solutions for connected and autonomous vehicles (CAVs). He has served as a guest editor, a technical program committee member, and a reviewer for numerous international conferences, journals, and IEEE TRANSACTIONS.

Dr Hadeel Alsolai is an associate professor at the College of Computer Science and Information Systems at Princess Nourah Bint Abdulrahman University, Riyadh, Saudi Arabia. She currently serves as the manager of the Research, Development, and Innovation Unit at the university. Dr Alsolai's research interests focus on data science, machine learning, and artificial intelligence. Her professional experience includes supervising research programs, managing innovation-driven projects, and conducting workshops on artificial intelligence.