Course Report: EVM Transaction Debugger & Analyzer

Module: SC6107 Blockchain Development Fundamentals (Part 2)
Programme: MSc (Blockchain), Nanyang Technological University
Academic Year: 2025–26, Trimester 1
Project Option: Option 7 — EVM Transaction Debugger & Analyzer
Report Type: Final Project Report

---

1. Executive Summary

This report describes the design, implementation, and evaluation of an **Ethereum transaction debugging and analysis platform** that helps developers understand transaction behaviour at the EVM level. The system parses execution traces, profiles gas usage, visualises state changes, and applies rule-based vulnerability detection, with a web frontend, a REST API, and support for both mock and live RPC-based analysis. The project satisfies the course requirements for core features, testing, security, and documentation, and is scoped to be achievable within a six-week timeline while remaining defensible in a presentation and Q&A.

---

2. Project Goal and Scope

2.1 Goal

The goal is to build a **production-quality** Ethereum transaction debugger and analyzer that:

- Parses Ethereum transaction execution traces (external and internal calls).
- Analyses gas usage and identifies gas-intensive operations.
- Visualises state changes (storage, balances, token transfers).
- Detects common smart contract vulnerabilities using original analysis logic.
- Presents results in a developer-friendly way and supports exportable reports.

The system is **not** a thin wrapper around existing tools; it implements **original analysis logic** (trace parsing, gas aggregation, heuristic vulnerability rules) and a clear system design.

2.2 Scope

- **In scope:** Trace analysis, gas profiling, state-diff visualisation, vulnerability detection, REST API, Next.js frontend, Hardhat/Foundry contracts and tests, frontend tests, documentation, export (JSON/Markdown).
- **Out of scope:** Full ABI-based decoding, prestate/poststate tracer integration, batch or comparative analysis, real-time monitoring, and integration with Slither/Mythril (mentioned as optional).

---

3. Core Features Implemented

3.1 A. Transaction Trace Analysis

- **Input:** Transaction hash.
- **Behaviour:**
  - When `RPC_URL` is set: the API uses `debug_traceTransaction` (callTracer) to fetch the trace, then a **shared Node.js analysis library** flattens the call tree into a list of calls with type (CALL/DELEGATECALL/STATICCALL/CREATE/CREATE2), from/to, value, gas used, and success/failure.
  - When RPC is not set: the API returns a **mock** trace for demo.
- **Output (frontend):** Execution Trace view with total calls, contracts involved, max depth, and an expandable list of calls with type badges and gas.

Deliverables:

- Solidity libraries: `TraceAnalyzer.sol` (call structures, types).
- Node: `frontend/src/lib/analyzeTransaction.ts` (trace fetch + flattening).
- Frontend: `TraceView.tsx` .

3.2 B. Gas Profiling

- **Behaviour:**
  - Gas usage is aggregated by function selector from the trace; per-call and per-function gas is computed.
  - Efficiency (gas used / gas limit) and optional global hints (e.g. low efficiency) are derived.
- **Output (frontend):** Gas Profile view with top gas-consuming functions (chart), gas breakdown by category (when data exists), optimisation hints, and summary metrics (efficiency, total gas).

Deliverables:

- Solidity: `GasProfiler.sol`.
- Node: gas aggregation in `analyzeTransaction.ts`.
- Frontend: `GasProfileView.tsx` (Recharts).

3.3 C. State Diff Visualisation

- **Behaviour:**
  - With **mock** data: storage changes, balance changes, and token transfers are shown.
  - With **live** RPC analysis: state diff is currently empty (prestate/poststate tracer not integrated); structure is in place for future extension.
- **Output (frontend):** State Changes view with tabs (Storage / Balances / Transfers), counts, and before/after/delta where applicable.

Deliverables:

- Solidity: `StateDiffAnalyzer.sol`.
- Frontend: `StateDiffView.tsx`.

3.4 D. Vulnerability Detection

- **Behaviour:**
  - **Rule-based** logic in the Node analysis library: e.g. external calls with value in the trace are flagged as potential reentrancy; counts and severity are aggregated.
  - Solidity `VulnerabilityDetector.sol` defines structures and patterns used in tests and

documentation.

- **Output (frontend):** Vulnerability Report view with severity counts and per-issue description and recommendation.

Deliverables:

- Solidity: `VulnerabilityDetector.sol`, test contracts (`VulnerableContract`, `SafeContract`) in `TestContracts.sol`.
- Node: `detectReentrancyHints()` in `analyzeTransaction.ts`.
- Frontend: `VulnerabilityReportView.tsx`.
- Documentation: `docs/security-analysis.md`.

---

4. Technical Stack and Architecture

4.1 Stack

- **Blockchain:** Ethereum (local mainnet fork or Sepolia).
- **Smart contracts:** Solidity 0.8.x, OpenZeppelin Contracts 5.x (optional via Foundry).
- **Backend / tooling:** Hardhat (primary for Windows-friendly setup), Foundry optional; Node.js for API and analysis library.
- **Frontend:** Next.js 14 (App Router), React 18, TypeScript, Tailwind CSS, ethers.js v6, Recharts.
- **Testing:** Jest + React Testing Library (frontend), Hardhat tests (JS), Foundry tests (Solidity) when Foundry is used.

4.2 Architecture

- **Frontend (Next.js):**
  - Single page: transaction hash input → call to `/api/analyze` → render trace, gas, state diff, vulnerability report; export JSON/Markdown.
- **API (Next.js API Routes):**
  - `GET /api/analyze?txHash=<hash>&rpcUrl=<optional>`: validates hash; if `rpcUrl` or `RPC_URL` is set, calls the shared analysis library and returns live result; otherwise returns mock.
- **Analysis library (Node):**

    - `frontend/src/lib/analyzeTransaction.ts` : ethers provider, `getTransaction` , `getTransactionReceipt` , `debug_traceTransaction` (callTracer); flattens trace, builds gas profile, runs heuristic vulnerability checks; returns a single `TransactionAnalysisResult` object.
- **Contracts (Solidity):**
    - Used for data structures and in-contract analysis logic; trace fetching and aggregation are implemented in Node for flexibility and RPC compatibility.

Detailed diagrams and data flow are in `docs/architecture.md` .

---

5. Testing

5.1 Frontend Tests (Jest)

- **Location:** `frontend/src/__tests__/` .
- **Coverage:**
    - API route: valid/invalid hash, response shape.
    - Components: `TransactionInput` , `TransactionAnalysis` , `TraceView` , `GasProfileView` , `StateDiffView` , `VulnerabilityReportView` , and Home page (success/error flows with mocked fetch).
    - Types: analysis type shapes.
- **Result:** 50 tests, 9 suites; **line coverage ~87%**, above the 80% requirement.
- **Commands:** `npm test` , `npm run test:coverage` (from `frontend/`).

5.2 Contract Tests

- **Hardhat:** `contracts/test-hardhat/TraceAnalyzer.test.js` (runtime and signer checks).
- **Foundry (optional):** `contracts/test/*.t.sol` for TraceAnalyzer, GasProfiler, StateDiffAnalyzer, VulnerabilityDetector and test contracts.
- **Commands:** `npx hardhat test` , or `forge test -vv` when Foundry is installed.

5.3 Gas and Scripts

- Gas reporting: `forge test --gas-report` (Foundry).
- CLI analysis: `node scripts/analyze-transaction.js <txHash> [rpcUrl]` for trace retrieval and file output when the node supports `debug_traceTransaction`.

---

6. Security

- **Input validation:** Transaction hash is validated (format and length) before RPC or mock.
- **Error handling:** API returns appropriate status codes and messages without exposing internals.
- **Contracts:**
  - Test contracts demonstrate **reentrancy** (VulnerableContract) and **checks-effects-interactions** (SafeContract).
  - Documentation in `docs/security-analysis.md` describes vulnerabilities and mitigations.
- **Vulnerability detection:** Implemented as heuristic hints (e.g. reentrancy) from trace structure; not a replacement for formal audits or dedicated tools.

---

7. Documentation

- **README.md:** Overview, structure, quick start, installation (Windows Hardhat path and Foundry path), features, testing, security.
- **docs/architecture.md:** Components, data flow, design decisions, **known limitations**, scalability and future work.
- **docs/security-analysis.md:** Security patterns in test contracts and vulnerability detection.
- **docs/gas-optimization.md:** Gas optimisation guidance.
- **docs/INSTALLATION_WINDOWS.md:** Windows installation (Hardhat and Foundry options).
- **docs/DEMO_STEPS_CN.md:** Step-by-step demo in Chinese.
- **docs/COURSE_REPORT_EN.md:** This course report in English.

---

8. Advanced / Bonus Features Delivered

- **Exportable reports:** Buttons “Export JSON” and “Export Markdown” on the analysis view; download of full result (JSON) or human-readable report (Markdown).
- **Real RPC-based analysis:** When `RPC_URL` (or query `rpcUrl`) is set, the API performs live trace fetch and original analysis instead of mock.
- **Dual toolchain:** Installation and tests can run with **Hardhat only** (Windows-friendly) or with **Foundry**; documented in README and installation docs.

---

9. Known Limitations and Future Work

- **RPC:** Live analysis requires a node supporting `debug_traceTransaction` (e.g. Geth, Erigon, or archive RPC).
- **State diff:** Not populated from RPC in the current implementation; would require prestate/poststate tracer support.
- **Function names:** Without ABIs, only selectors are shown; ABI resolution would improve readability.
- **Vulnerability detection:** Rule-based and heuristic; not a substitute for Slither/Mythril or professional audits.
- **Scale:** Single-transaction analysis; no batching or comparison.

Planned improvements (see `docs/architecture.md`): prestate/poststate tracer integration, ABI registry, caching, and optional batch/async processing.

---

10. Conclusion

The EVM Transaction Debugger & Analyzer project delivers a **complete, well-scoped** platform that meets the course requirements:

- **Core features (A–D):** Trace analysis, gas profiling, state-diff visualisation, and vulnerability detection are implemented with original logic and a clear separation between Solidity structures and Node.js analysis.
- **Testing:** Frontend coverage exceeds 80%; contract and API behaviour are covered by unit and integration tests.

- **Security:** Input validation, safe error handling, and documented security patterns and limitations.
- **Documentation:** Architecture, security, gas optimisation, installation, demo steps, and this course report.

The system is suitable for **demonstration and Q&A**: behaviour is consistent, limitations are stated, and extensions (e.g. state diff from RPC, ABI decoding) are documented as future work. It is ready for submission and presentation within the SC6107 course framework.

End of Report

Document version: 1.0 | Aligned with repository state and docs.