DeObfusca-AI – PRD

Purpose: Simulate automated reverse engineering/deobfuscation of binaries and generate verified C code with iterative refinement.

Summary

A system that ingests an obfuscated binary, runs automated analysis (Ghidra + Code Property Graph), encodes structure with a GNN, generates candidate C with an LLM, verifies behavior with Z3 (RL/verification), and iteratively refines output (diffusion, multi-agent, CoT) until equivalence is reached or a fixed iteration budget is consumed. Produces a report/log of artifacts, verification results, and refinement history.

Context

Reverse engineering and deobfuscation are costly and manual. This platform automates common steps, focuses on correctness via neural-symbolic verification, and offers iterative refinement to reduce manual effort. It relies on pre-existing frameworks/libraries (Ghidra, PyTorch, transformers, Z3) and model scaffolding already in the repo; no new research is claimed beyond integration and workflow hardening.

Key Deliverables

- Hosted microservices (Docker Compose) for Ghidra, CPG, GNN, LLM, RL/Z3 verification, Diffusion, Multi-Agent, CoT, Orchestrator, Frontend, Backend.
- End-to-end flow: upload binary → analyze → encode → generate → verify → refine → output.
- Refinement loop with bounded iterations (e.g., 3) and graceful fallbacks/timeouts.
- Reports/logs: per-step status, verification results, counterexamples, refinement attempts.
- Training orchestration scripts and configs (train_all_models.py, training_config.json).

Basic Flow

1) **Binary Input**: User uploads binary; orchestrator assigns an ID and logs metadata.
2) **Analysis (Ghidra/CPG)**: Ghidra lifts to assembly/P-Code and derives CFG; CPG service fuses CFG + AST + PDG, preserving control (dominators/post-dominators), data (definitions/uses), and syntactic structure so later models see both flow and semantics.

3) **Graph Encoding (GNN)**: A transformer-style GNN encodes nodes/edges with positional/dominator/context signals. The goal is to learn representations that are invariant to low-level obfuscation (junk ops, basic block reordering) while retaining data/control dependencies that matter for semantics.
4) **Initial Decompilation (LLM)**: The LLM consumes graph embeddings (or serialized features) and emits C. Grammar-constrained decoding prunes invalid syntax; sliding-window/context stitching handles larger functions without losing long-range references.
5) **Verification (RL + Z3)**: The RL/verification service frames equivalence as a constraint problem: Z3 checks whether outputs from generated C can differ from binary outputs under any input within sampled/path-bounded scope. Counterexamples (or unsat cores) become feedback. If satisfiable differences are found, refinement is triggered.
6) **Refinement Loop (cap N iterations)**:

- **Diffusion**: Proposes small, denoised edits toward consistency with constraints.
- **Multi-agent**: Generates diverse candidate fixes in parallel to broaden the search space.
- **CoT**: Produces stepwise rationales to pinpoint and fix logical/typing/control issues.
- LLM re-generates with feedback context; verification reruns. Loop stops on proof of equivalence, iteration cap, or time budget.

7) **Output**: Best verified version (or highest-scoring partial if no proof) plus artifacts: graphs, candidates, counterexamples, and refinement history.

Methods

- **Code Property Graphs**: Integrate CFG (control), AST (syntax), and PDG (dataflow) so models see both structural and semantic cues. Useful against control-structure obfuscation because data dependencies persist even when control is rearranged.
- **GNN encoder**: Edge-aware transformer blocks over graphs; incorporates dominator/post-dominator or positional encodings to retain execution order hints despite obfuscation. Aims for invariance to dead-code insertion and basic block shuffling.
- **LLM decompiler**: Conditions on graph embeddings and uses grammar-constrained decoding to avoid invalid C. Sliding-window with overlap preserves variable linkage and control context for larger functions.
- **Neural-Symbolic Verification (Z3)**: RL wrapper around Z3 builds constraints tying binary outputs to generated C; checks satisfiability to detect divergences. Counterexamples are turned into actionable feedback (mismatched outputs, unsatisfied path constraints, typing/overflow hints).
- **Refinement trio**:

  - **Diffusion**: Iteratively denoises toward constraint satisfaction, treating code edits as steps in a latent space.
  - **Multi-agent**: Parallel agents explore alternative fix strategies (e.g., control repair vs. data repair) to reduce local minima.
  - **Chain-of-thought**: Generates explicit reasoning steps to localize and correct logic/typing/control issues.
- **Iteration control**: Bounded iterations (e.g., 3), per-service timeouts, and safe_request wrappers to avoid pipeline stalls; best-effort degradation if a service is slow/unavailable.

Tech Stack

- **Frontend**: React
- **Backend**: Node.js API
- **AI Services (Flask/Python)**: ghidra-service, cpg-service, gnn-service, llm-service, rl-service (Z3), diffusion-service, multi-agent-service, cot-service, orchestrator.
- **Verification**: Z3 (Python bindings) inside RL/verification service.
- **Packaging**: Docker Compose; CPU-compatible LLM (4-bit quant) with optional GPU.

AI Models and Libraries

Model/Service	Role	Libraries/Frameworks
GNN	Graph encoding of CPG/CFG/AST/PDG	PyTorch, PyTorch Geometric/transformers
LLM	C code generation (grammar constrained)	HuggingFace transformers, PyTorch
RL + Z3 verifier	Behavioral equivalence checking	z3-solver, PyTorch
Diffusion	Iterative refinement proposals	PyTorch
Multi-agent	Parallel alternative fix proposals	Python (Flask service)
Chain-of-thought	Stepwise reasoning for fixes	HuggingFace transformers, PyTorch
Orchestrator	Pipeline coordination, timeouts, retries	Flask, requests

Requirements

- **Functional**: upload → analyze → encode → generate → verify → refine; health endpoints; logging; bounded retries; training scripts runnable.

- **Non-functional**: per-service timeouts; resilience to partial failures; CPU-first with optional GPU; ~50GB disk for data; 16GB+ RAM recommended.
- **Security**: input validation on service endpoints; isolation via Docker; auth handled at frontend/backend layer (if enabled).

Risks & Mitigations

- Hallucinated/invalid C → grammar constraints + Z3 checks + refinement loop.
- Slow verification on large functions → timeouts; iteration caps; sampling limited inputs.
- Coverage gaps → multiple refinement proposals (diffusion/multi-agent) and CoT reasoning.
- Model drift/quality → retraining via provided scripts; config-tunable hyperparams.

Success Metrics

- Functional equivalence rate (Z3/RL verified)
- Compile success rate of generated C
- Refinement convergence within 3 iterations
- Reduction in manual RE time (qualitative/benchmarks)

Timeline (example)

- Week 1: Harden endpoints/timeouts; verification tuning.
- Week 2: Refinement quality pass (diffusion/multi-agent/CoT); logging/metrics.
- Week 3: Training run; evaluate metrics; doc updates.
- Week 4: Stabilization; packaging.

Future Enhancements

- Broader architecture support (ARM/MIPS)
- Faster verification heuristics; better counterexample sampling
- Integrated UI for side-by-side binary/graph/code view
- Incremental training with new obfuscation samples