Project Title: Reverse Engineering Assisted Vulnerability Discovery

---

Short Project Description:

The tool combines static analysis, control flow reconstruction, and targeted fuzzing to identify security flaws in compiled software.

Component	Role
Binary Decompiler Module	Converts binaries back to high-level code
Control Flow Graph Builder	Reconstructs control flow relationships
Vulnerability Pattern Scanner	Looks for known vulnerability patterns (e.g., stack overflows, etc)
Fuzzer Seed Generator	Generates inputs for dynamic fuzzing based on decompiled code
Reporting Engine	Creates vulnerability discovery reports

Component Details:

1. **Binary Decompiler Module**:
    - Uses tools like Ghidra or Hex-Rays to turn machine code into pseudo-C.
2. **Control Flow Graph Builder**:
    - Rebuilds possible execution paths inside the binary.
3. **Vulnerability Pattern Scanner**:
    - Scans for insecure coding patterns:
        - Unbounded copies (`strcpy`, `memcpy`)
        - Insecure permissions
        - Race conditions
        - Etc
4. **Fuzzer Seed Generator**:
    - Automatically generates valid input seeds for fuzzing, based on reversed data structures.
5. **Reporting Engine**:
    - Lists potential vulnerabilities and fuzzing entry points.

Overall System Flow:

- Input: Binary executable
- Output: List of potential vulnerabilities + fuzzing strategy
- Focus: **Using RE to aid deeper vulnerability discovery**.

Internal Functioning of Each Module:

1. Binary Decompiler Module

- **Decompilers**:
  - Ghidra (Autoanalysis + Decompile mode).
  - Hex-Rays Decompiler (proprietary).
  - Etc
- **Goal**:
  - Translate assembler instructions into C-like pseudocode.
- **Challenges**:
  - Handle obfuscated binaries (packed, virtualized).
  - Resolve indirect call addresses.

---

2. Control Flow Graph Builder

- **Steps**:
  - Identify function entry points.
  - Analyze basic blocks (entry, decision, merge, etc).
  - Build graph:
    - Nodes: basic blocks
    - Edges: jumps, branches

---

3. Vulnerability Pattern Scanner

- **Patterns searched**:
  - Buffer overflows: functions like `strcpy`, `memcpy` without bounds checks.
  - Use-After-Free: missing object reference invalidation.
  - Integer overflows: arithmetic operations without proper validation.
  - Etc
- **Static Taint Analysis**:
  - Track user input through the code to sensitive operations (e.g., system calls).

---

4. Fuzzer Seed Generator

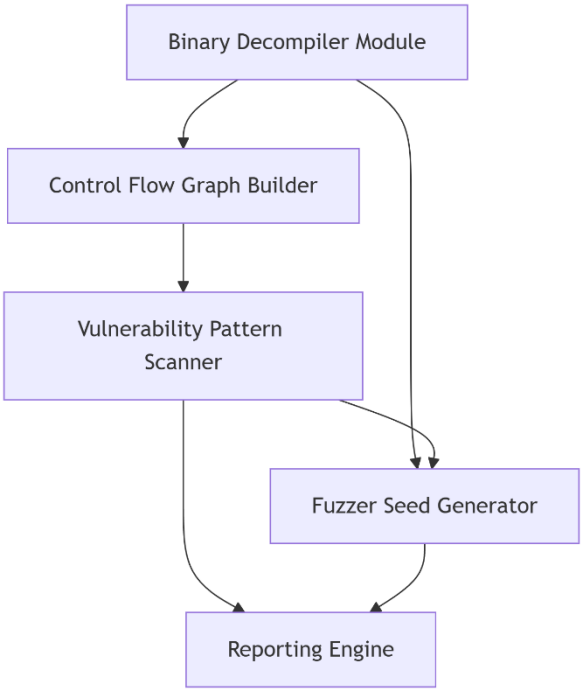
- **Approach**:
  - Identify functions that receive external input (e.g., via `read`, `recv`).
  - Analyze argument structures (e.g., length fields, file parsers, etc).
  - Generate minimal valid inputs to target reachable code paths.
- **Benefit**:
  - Fuzzers become more effective, less blind, and more targeted.

---

5. Reporting Engine

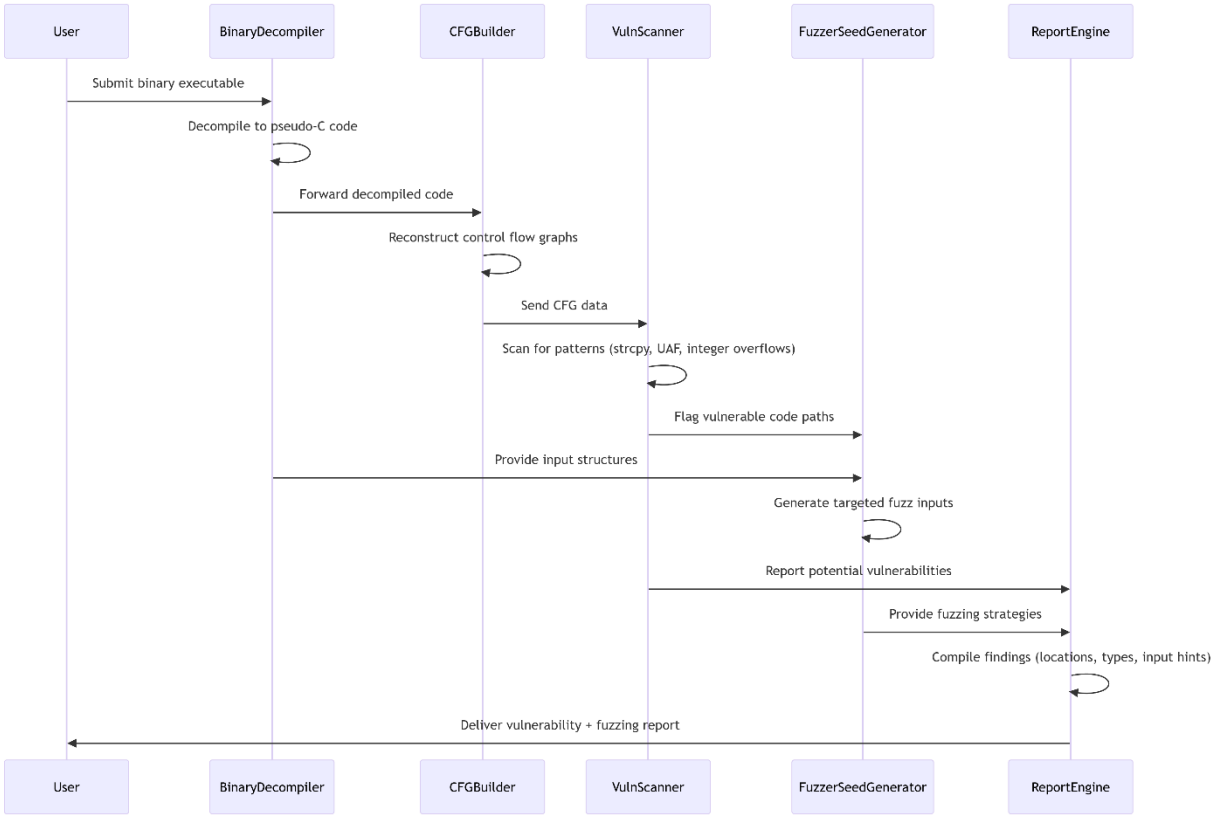
- **Final Output**:
  - Code locations of potential vulnerabilities
  - Types of issues suspected
  - Input crafting hints for fuzzers

Component Diagram



- The **Binary Decompiler Module** feeds decompiled code to both the **Control Flow Graph Builder** and **Fuzzer Seed Generator**.
- The **CFG Builder** reconstructs execution paths for the **Vulnerability Pattern Scanner** to analyze.
- The **Vulnerability Pattern Scanner** identifies insecure code patterns and shares flagged paths with the **Fuzzer Seed Generator**.
- Both the scanner and seed generator contribute data to the **Reporting Engine**, which synthesizes findings into actionable reports.

Sequence Diagram



- The **User** submits a binary, which is decompiled into pseudo-C code.
- The **Control Flow Graph Builder** reconstructs execution paths to guide the **Vulnerability Pattern Scanner**.
- The scanner identifies risky patterns (e.g., `strcpy` usage, etc), while the **Fuzzer Seed Generator** crafts targeted inputs based on decompiled data structures.
- The **Reporting Engine** aggregates vulnerabilities and fuzzing strategies into a prioritized report for the **User**.

Detailed Project Description: Reverse Engineering Assisted Vulnerability Discovery

A system that leverages reverse engineering (RE) to discover vulnerabilities in binary executables. This system combines static analysis, control flow reconstruction, and targeted fuzzing to identify security flaws in compiled software.

---

1. System Overview

The system analyzes binary files (e.g., EXE, ELF) to decompile them into readable code, identify vulnerable patterns, and generate fuzzing strategies. It focuses on **static analysis** and **guided dynamic testing** to uncover issues like buffer overflows, use-after-free, race conditions, etc.

---

2. Component Design & Implementation

2.1 Binary Decompiler Module

Functionality:

- Converts machine code into high-level pseudocode (e.g., C-like) for analysis.

Implementation Steps (e.g.):

1. **Tool Integration**:
    - **Ghidra**: Use headless mode for batch decompilation.
      ```
      ./analyzeHeadless /path/to/project -import malware.exe -postScript decompile.py
      ```
    - **Hex-Rays IDA Pro**: Leverage IDAPython scripting for automated decompilation.
2. **Handling Obfuscation**:
    - Integrate unpacking tools (e.g., **Unipacker** for UPX) and anti-virtualization techniques.

3. **Output**:
    - Pseudocode files (e.g., `malware.c`) with function definitions and data structures.

Tools (e.g.):

- Ghidra, Hex-Rays IDA Pro, Unipacker, etc.

2.2 Control Flow Graph Builder

Functionality:

- Reconstructs execution paths to guide vulnerability detection.

Implementation Steps (e.g.):

1. **CFG Construction**:
    - Use **angr** or **Radare2** to identify basic blocks and edges.
    - Example:
      ```
      import angr
      proj = angr.Project("malware.exe", auto_load_libs=False)
      cfg = proj.analyses.CFGFast()
      ```
2. **Function Analysis**:
    - Map entry points, loops, and branches.
3. **Visualization**:
    - Export CFGs as GraphML or DOT files for tools like **Gephi**.

Output:

- Structured CFGs (nodes = basic blocks, edges = jumps/branches).

Tools (e.g.):

- angr, Radare2, NetworkX, etc.

2.3 Vulnerability Pattern Scanner

Functionality:

- Identifies insecure code patterns in decompiled pseudocode.

Implementation Steps (e.g.):

1. **Static Analysis Rules**:
    - Define YARA-like rules for vulnerabilities
    - Example:
      ```
      rules:
        - name: "Buffer Overflow"
          pattern: "strcpy(dest, src);"
          condition: "no_size_check(dest, src)"
          severity: "High"
      ```
    - Target patterns:
        - Unbounded `strcpy`/`memcpy` calls.
        - Missing pointer validation (e.g., `free(ptr)` without null check).
2. **Taint Analysis**:
    - Track user input from `read`/`recv` to sensitive sinks (e.g., `system`).
3. **Integration**:
    - Use **BinAbsInspector** (Clang-based static analyzer) for C/C++ pseudocode.

Output:

- List of flagged code locations with vulnerability types.

Tools (e.g.):

- YARA, BinAbsInspector, Clang Static Analyzer, etc.

2.4 Fuzzer Seed Generator

Functionality:

- Creates targeted fuzzing inputs based on decompiled code structures.

Implementation Steps (e.g.):

1. **Input Structure Extraction**:
    - Identify functions accepting external input (e.g., file parsers, network handlers, etc).
    - Extract data formats (e.g., structs, headers, etc) from pseudocode.
2. **Seed Generation**:
    - Use **AFL++** or **libFuzzer** to generate valid initial inputs.
    - Example:
      ```
      afl-clang-fast -o fuzzer fuzzer.c
      afl-fuzz -i seeds/ -o findings/ ./fuzzer
      ```
3. **Code Coverage Guidance**:
    - Prioritize seeds that reach vulnerable code paths identified by the CFG.

Output:

- Corpus of fuzzing inputs (e.g., malformed files, network packets, etc).

Tools (e.g.):

- AFL++, libFuzzer, Python struct module, etc.

2.5 Reporting Engine

Functionality:

- Compiles findings into actionable reports with remediation guidance.

Implementation Steps (e.g.):

1. **Data Aggregation**:
    - Merge vulnerability locations, CFG paths, and fuzzing seeds.
2. **Report Templates**:
    - **HTML**: Use Jinja2 for interactive reports with code snippets.
    - **SARIF**: Export for integration with CI/CD tools like GitHub Code Scanning.

    - **Etc**
3. **Remediation Suggestions**:
    - Provide code fixes (e.g., replace `strcpy` with `snprintf`).

Output:

- Report with:
    - Vulnerability type and location.
    - Fuzzing strategy (e.g., "Target `parse_packet()` with malformed UDP payloads").
    - Severity ratings (Critical/High/Medium).
    - Etc

Tools (e.g.):

- Jinja2, SARIF SDK, WeasyPrint, etc.

---

3. Technology Stack (e.g.)

- **Decompilation**: Ghidra, Hex-Rays IDA Pro, etc.
- **Analysis**: angr, Radare2, BinAbsInspector, etc.
- **Fuzzing**: AFL++, libFuzzer, etc.
- **Reporting**: Jinja2, SARIF, Gephi, etc.

---

4. Evaluation & Validation

1. **Accuracy Testing**:
    - Test on binaries with known vulnerabilities (e.g., CVE-2023-1234).
    - Metrics: True positive rate, false positive rate.
2. **Fuzzing Effectiveness**:
    - Compare code coverage and crash discovery rates with/without seed generation.

3. **Performance**:
    - Measure decompilation speed (e.g., seconds per MB) and CFG construction time.

---

5. Development Roadmap

1. **Phase 1**: Implement Binary Decompiler and CFG Builder.
2. **Phase 2**: Build Vulnerability Scanner and integrate static analysis.
3. **Phase 3**: Develop Fuzzer Seed Generator and Reporting Engine.
4. **Phase 4**: Validate on real-world binaries (e.g., OpenSSL, Curl).

---

6. Challenges & Mitigations (optional)

- **Obfuscation**: Use dynamic taint analysis to resolve indirect jumps.
- **Scalability**: Parallelize decompilation with multiprocessing.
- **False Positives**: Refine rules with manual validation and ML-based filtering.

---

7. Glossary

- **CFG**: Control Flow Graph
- **RE**: Reverse Engineering
- **SARIF**: Static Analysis Results Interchange Format
- **AFL**: American Fuzzy Lop
