FSM_Binary_Analyzer

April 30, 2025

1 File: FSM_Binary_Analyzer.ipynb

Author: Taylor Marrion Prepared for: CSC 786 - Computer Science Problems Professor: Dr. John Hastings Date: 24 April, 2025

2 FSM-Based Vulnerability Detection in Tokenized Assembly

2.0.1 Objective

This project investigates the use of Finite State Machines (FSMs) for binary vulnerability detection. This project aims to determine the validity of this approach in detecting vulnerabilities in compiled binaries **without access to source code** and instead analyzes disassembled binaries tokenized into abstract memory operations (e.g., READ, WRITE, CALL). To limit scope in this proof of concept, we restrict analysis to detecting instances of CWE-457 (Use of Uninitialized Variable) by analyzing tokenized assembly extracted from the Juliet Test Suite.

2.0.2 Dataset Summary

- **Source**: Juliet Test Suite, compiled to ELF binaries with extracted disassembly.
- **Format**: Disassembly `.asm` files organized into `good` and `bad` labeled directories.
- **Features**: Tokenized instructions from disassembled code.
- **Target**: Binary label (`safe`, `vulnerable`) derived from the CWE-457 subset.
- **Total Samples**: 1,852 (50% vulnerable, 50% safe).

2.0.3 Key Tasks / Project Phases

1. **Disassembly Normalization**: Normalize x86_64 disassembly into abstract token sequences (`READ`, `WRITE`, `END`, etc.).
2. **Tokenization**: Extract operands related to stack variables and function arguments for FSM processing.
3. **FSM Simulation**: Emulate variable initialization and access through a simple finite state machine.
4. **Full Dataset Evaluation**:Analyze the full disassembly dataset to compute precision, recall, F1 score, and accuracy.
5. **Results & Discussion**: Present evaluation results and discuss strengths, weaknesses, and future work.
6. **Presentation**: Document methodology, justify design choices, and share results.

2.0.4 Deliverables

- FSM Binary Analyzer Python notebook
- Supporting Python scripts (fsm.py, tokenizer.py, runner.py)
- Write-up explaining implementation and results
- Class presentation summarizing project goals and outcomes

2.0.5 Acknowledgments

This project uses the following resources and projects: - Juliet C/C++ 1.3 - Juliet Test Suite augment

2.1 Phase 0: Environment Setup

> **Note:**
> This notebook assumes you have already manually created and activated a Python virtual environment (`venv/` folder).
>
> Be sure to select the correct Python interpreter (kernel) corresponding to your virtual environment when opening this notebook in VSCode.

To create and activate a virtual environment manually:

```
python3 -m venv venv
source venv/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
```

[1]:
```
# Requirements check - ensures all necessary packages are installed.

!pip install -q -r requirements.txt
```

[2]:
```
# Import main libraries

import sys
import os
import re
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt

sys.path.append(os.getcwd())
```

2.1.1 Dataset Validation

Before proceeding, ensure the disassembled dataset is available.

This project expects: - Disassembled binaries under `juliet-test-suite-c/disasm/` - Files organized by CWE category (`good/` and `bad/` folders inside)

[3]:
```python
# Validate Disassembly Dataset

from pathlib import Path
import glob

# Define dataset paths
disasm_root = Path("juliet-test-suite-c/disasm/")

# Verify root exists
if not disasm_root.exists():
    raise FileNotFoundError(f"Disassembly root not found at {disasm_root.
 ↪resolve()}.\nPlease run dataset generation steps first.")

# Check sample counts
good_samples = list(disasm_root.glob("CWE457/good/*.asm"))
bad_samples = list(disasm_root.glob("CWE457/bad/*.asm"))

print(f"Good samples found: {len(good_samples)}")
print(f"Bad samples found: {len(bad_samples)}")

# Quick preview
if good_samples:
    print(f"\nExample GOOD file: {good_samples[0]}")
if bad_samples:
    print(f"Example BAD file: {bad_samples[0]}")

# Basic assertion
assert len(good_samples) > 0 and len(bad_samples) > 0, "Disassembly samples not␣
 ↪found!"
```

```
Good samples found: 926
Bad samples found: 926

Example GOOD file: juliet-test-suite-
c/disasm/CWE457/good/CWE457_Use_of_Uninitialized_Variable__double_08-good.asm
Example BAD file: juliet-test-suite-
c/disasm/CWE457/bad/CWE457_Use_of_Uninitialized_Variable__struct_array_declare_p
artial_init_10-bad.asm
```

2.2 Phase 1: Disassembly Normalization and Tokenization

This phase converts raw disassembled binaries into normalized abstract tokens.

We define a simple but effective tokenization: - `READ`: Reading from memory (e.g., mov reg, [mem]) - `WRITE`: Writing to memory (e.g., mov [mem], reg) - `CALL`: Function call - `END`: Function return (`ret`) - `NOOP`: Any other instruction (ignored for vulnerability modeling)

This abstraction allows a Finite State Machine (FSM) to reason about memory usage without needing architecture-specific syntax or source code.

[4]:
```
# Import tokenizer functions from the project code
from fsm_binary_analyzer.tokenizer import normalize_disassembly, extract_operand
```

[5]:
```
import random

# Helper function to preview a tokenized disassembly
def preview_tokenization(file_path):
    with open(file_path, "r") as f:
        lines = f.readlines()
    tokens = normalize_disassembly(lines)

    print(f"File: {file_path.name}")
    print(f"Total tokens: {len(tokens)}\n")
    print("First 50 tokens:")
    print(tokens[:50])
    print("-" * 80)

# Pick 1 random GOOD and 1 random BAD
sample_good = random.choice(good_samples)
sample_bad = random.choice(bad_samples)

# Preview them
preview_tokenization(sample_good)
preview_tokenization(sample_bad)
```

```
File: CWE457_Use_of_Uninitialized_Variable__twointsclass_array_declare_partial_i
nit_05-good.asm
Total tokens: 1097

First 50 tokens:
['NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', ('CALL', 'call
rax'), 'NOOP', 'END', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP']
--------------------------------------------------------------------------------
File:
CWE457_Use_of_Uninitialized_Variable__new_double_array_partial_init_16-bad.asm
Total tokens: 822

First 50 tokens:
['NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', ('CALL', 'call
rax'), 'NOOP', 'END', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP', 'NOOP',
'NOOP', 'NOOP']
```

2.3 Phase 2: FSM Modeling and Full Dataset Evaluation

Goal: Simulate a Finite State Machine (FSM) over the tokenized disassemblies to detect vulnerabilities.

We will: - Walk each function starting from main - Track memory reads and writes - Detect cases where a READ occurs before a WRITE - Compute overall detection metrics (Precision, Recall, F1)

[6]:
```
# Import FSM class and functions from the project code
from fsm_binary_analyzer.fsm import SimpleFSM
```

[7]:
```
# Import runner functions from the project code
from fsm_binary_analyzer.runner import analyze_disassembly
```

[8]:
```
from tqdm import tqdm

def evaluate_full_dataset(good_samples, bad_samples):
    y_true = []
    y_pred = []

    print("\n[*] Evaluating GOOD files...")
    for path in tqdm(good_samples):
        result = analyze_disassembly(path)

        y_true.append(0)  # GOOD = 0
        y_pred.append(int(result))

    print("\n[*] Evaluating BAD files...")
    for path in tqdm(bad_samples):
        result = analyze_disassembly(path)

        y_true.append(1)  # BAD = 1
        y_pred.append(int(result))

    return y_true, y_pred
```

[9]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score

def report_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

    print("\n===== FSM Vulnerability Detection Results =====")
```

```
    print(f"Accuracy:  {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall:    {recall:.4f}")
    print(f"F1 Score:  {f1:.4f}")
    print("================================================\n")
```

2.3.1 Evaluation Metric Definitions

True Positives (TP): Correctly flagged vulnerable binaries False Positives (FP): Safe binaries incorrectly flagged as vulnerable False Negatives (FN): Vulnerable binaries missed by the FSM True Negatives (TN): Correctly identified safe binaries Accuracy: "How many predictions were correct out of all predictions?" (TP + TN) / Total Precision: "How much of what we said was vulnerable is actually vulnerable?" (TP) / (TP + FP) Recall: "How many vulnerable cases did we detect?" (TP) / (TP + FN) F1: Balancing precision and recall 2 * ((P x R) / (P + R))

[10]:
```
# Full run
y_true, y_pred = evaluate_full_dataset(good_samples, bad_samples)

# Report metrics
report_metrics(y_true, y_pred)
```

```
[*] Evaluating GOOD files…
100%|      | 926/926 [00:04<00:00, 224.12it/s]

[*] Evaluating BAD files…
100%|      | 926/926 [00:05<00:00, 184.56it/s]

===== FSM Vulnerability Detection Results =====
Accuracy:  0.6766
Precision: 0.7935
Recall:    0.4773
F1 Score:  0.5961
================================================
```

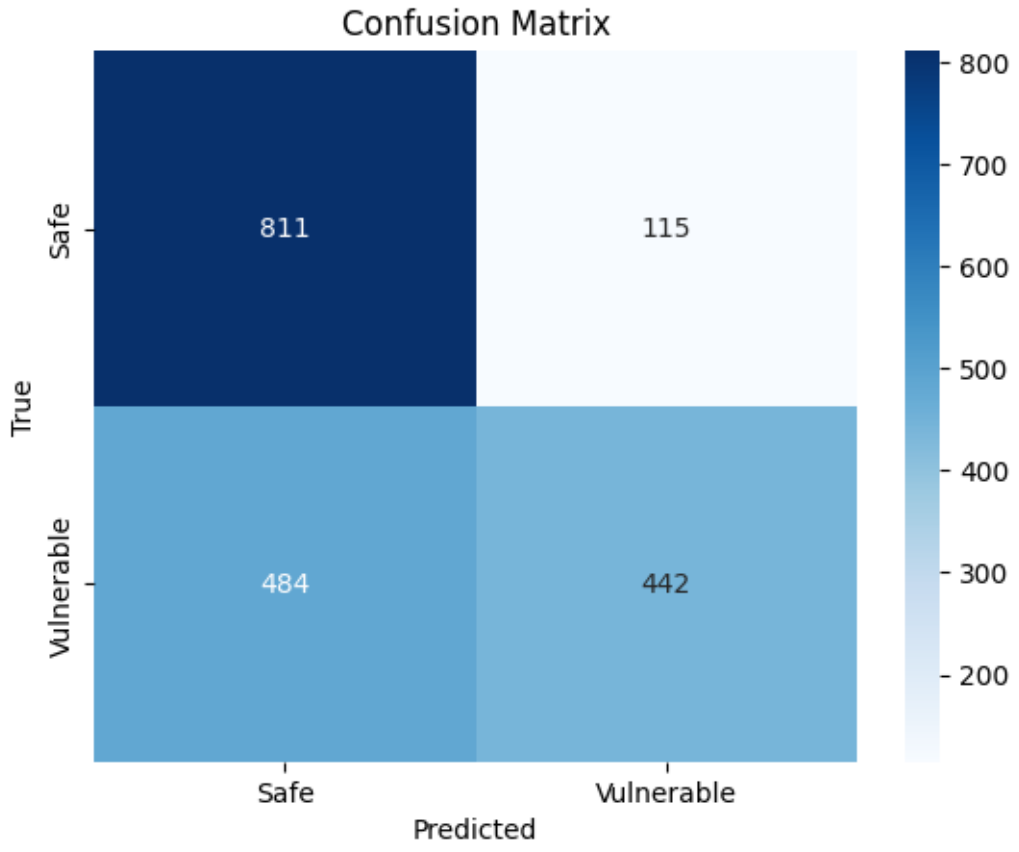
[11]:
```
# Report precision, recall, F1, and confusion matrix
from sklearn.metrics import classification_report, confusion_matrix

print(classification_report(y_true, y_pred, target_names=["Safe",␣
 ↪"Vulnerable"]))

cm = confusion_matrix(y_true, y_pred)
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=["Safe",␣
 ↪"Vulnerable"], yticklabels=["Safe", "Vulnerable"])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

	precision	recall	f1-score	support
Safe	0.63	0.88	0.73	926
Vulnerable	0.79	0.48	0.60	926
accuracy			0.68	1852
macro avg	0.71	0.68	0.66	1852
weighted avg	0.71	0.68	0.66	1852



2.3.2 Results:

- Precision is relatively high (79%), meaning when the FSM predicts a vulnerability, it's often correct.
- Recall is moderate (47%), meaning the FSM misses some vulnerabilities (false negatives).
- F1 Score balances both and reflects overall FSM effectiveness (~60%).

2.4 Phase 3: Presentation

–> You are here <–

[12]:
```
print("Goodbye!")
```

```
Goodbye!
```