# Finite State Machine Based Vulnerability Detection in Binary Executables: A Case Study on CWE-457

Taylor Marrion
*Beacom College of Computer and Cyber Sciences*
*Dakota State University*
Madison, SD, USA
taylor.marrion@trojans.dsu.edu

***Abstract*—This project explores the feasibility of using a Finite State Machine (FSM) approach to detect vulnerabilities in compiled binaries without access to source code. We focus specifically on identifying instances of CWE-457 (Use of Uninitialized Variable) by analyzing tokenized x86-64 disassembly from the Juliet Test Suite. Each function is abstracted into a sequence of memory operation tokens and processed using an FSM designed to simulate variable state tracking. Although the approach achieved modest overall performance—highlighted by an F1 Score of 0.596 and high precision (0.79), the results suggest that FSM-based analysis can reliably flag uninitialized variable usage under certain conditions. However, further research is needed to reduce false negatives, extend detection beyond a single vulnerability class, and evaluate resilience against optimization, obfuscation, and real-world binary variation.**

***Index Terms*—Cybersecurity, Vulnerability Detection, Binary Analysis, FSM, Static Analysis, CWE-457**

## I. Introduction

As cybersecurity threats continue to grow in scale and complexity, the need for scalable vulnerability detection methods that operate directly on compiled binaries is becoming increasingly urgent. In practice, analysts are often required to detect security flaws in software without access to the original source code, especially in closed-source or legacy systems. Traditional approaches such as static analysis, symbolic execution, and manual reverse engineering offer powerful capabilities, but each faces limitations in terms of scalability, coverage, or required domain expertise.

This study investigates a finite state machine (FSM) based approach to static binary analysis, aimed at identifying vulnerable patterns by simulating variable state transitions at the instruction level. FSMs offer an interpretable and deterministic framework for modeling control and data flow, making them a natural candidate for analyzing disassembled code. By abstracting memory operations into semantic tokens such as `READ`, `WRITE`, and `CALL`, the FSM can simulate variable usage and detect violations of expected initialization patterns.

Focusing specifically on Common Weakness Enumeration (CWE)-457 (Use of Uninitialized Variable), this project analyzes x86-64 binaries compiled from the Juliet Test Suite. Each function is disassembled and tokenized before being evaluated by the FSM engine. While the overall results are modest—highlighted by an F1 Score of 0.596—the analysis demonstrates reliable detection in scenarios where initialization behavior is clearly modeled. High precision scores (0.79) suggest that the FSM is conservative in its assessments, favoring correctness over completeness.

The primary goal of this work is not to compete with deep learning-based techniques or production static analyzers, but rather to evaluate the practical viability of FSMs as a lightweight, interpretable tool for binary vulnerability detection. By validating this technique on a constrained subset of known-vulnerable programs, we aim to establish a foundation for more advanced FSM-driven analyses and future hybrid approaches.

## II. Background

### *A. Binary Vulnerability Detection*

Detecting vulnerabilities in binary executables without access to source code is a longstanding challenge in cybersecurity. Static binary analysis techniques aim to identify flaws such as buffer overflows, use-after-free errors, and uninitialized variable usage by reasoning about control flow and data dependencies directly from disassembled code. While tools that utilize symbolic execution and abstract interpretation offer thorough coverage, they are often hindered by high computational costs, path explosion, and the need for intricate modeling of program semantics.

Dynamic analysis methods, such as fuzzing and runtime monitoring, provide valuable insight into actual program behavior but require executable inputs and instrumented environments. These methods may miss latent or state-dependent vulnerabilities. Consequently, scalable static techniques that operate purely at the binary level, particularly those that are explainable and deterministic, are of growing interest in both research and applied security settings.

### *B. Finite State Machines in Static Analysis*

FSMs offer a structured framework for modeling system behavior, making them well-suited for lightweight static analysis. In the context of program security, FSMs have been employed to model input validation, protocol state transitions, and execution patterns in both source and binary code. FSMs are particularly advantageous for their deterministic operation, low computational overhead, and inherent explainability, all of which make them attractive for security auditing and reverse engineering tasks.

In this study, FSMs are used to simulate the flow of variable states within disassembled functions. Memory operations are abstracted into a simplified token vocabulary, allowing the FSM to reason about whether a variable is used before it is safely initialized. While this approach lacks the flexibility of probabilistic models or symbolic reasoning engines, it provides a conservative and interpretable method for detecting a narrow class of vulnerabilities, specifically, uninitialized variable usage.

### C. CWE-457: Use of Uninitialized Variable

CWE-457 is defined as the use of variables that have not been initialized prior to use. This flaw can lead to undefined behavior, data leakage, or exploitable conditions depending on the context. In binaries, uninitialized variable usage often manifests as memory reads from stack or heap locations that have not been written to. Detecting such usage without access to high-level semantics or debugging symbols is difficult, making CWE-457 a suitable target for evaluating FSM-based binary analysis.

### D. Related Work

Recent research has explored a variety of methods for binary analysis. Pordanesh and Tan [1] examined the capabilities and limitations of large language models (LLMs) such as GPT-4 in reverse engineering tasks, highlighting the potential for structured reasoning in compiled code contexts. Anderson et al. [2] leveraged machine learning to automate subroutine classification in malware using graph-based features, underscoring the value of structured representations in binary analysis. Asm2Seq by Taviss et al. [3] demonstrated the effectiveness of abstracting assembly code into learnable token sequences for summarization tasks, reinforcing the notion of treating binaries as structured language data. Udeshi et al. [4] developed a symbolic execution framework for recovering equations from binaries, revealing the power of structured state tracking in low-level analysis. Van Zeeland's work [5] on extracting state machines from legacy C programs further supports FSMs as viable tools for modeling execution logic. Additionally, tools like `angr` [6] and Ramblr [7] provide practical frameworks for static binary analysis, illustrating both the promise and the challenges of achieving scalability and precision through control flow and symbolic reasoning. These works collectively motivate this project's investigation into FSMs as a lightweight yet explainable approach to vulnerability detection in binary executables.

## III. Dataset Preparation

### A. Dataset Source

The dataset used for this study is derived from the Juliet Test Suite C/C++ Version 1.3 [8], a widely recognized collection of synthetic test cases designed to evaluate the effectiveness of static and dynamic code analysis tools. Specifically, we utilized the Linux-adapted version provided by Richardson [9], which compiles the test cases into x86-64 ELF binaries suitable for disassembly and analysis.

### B. CWE-457 Filtering

To limit project scope and maintain a highly controlled experimental setting, we filtered the dataset to include only samples corresponding to CWE-457 (Use of Uninitialized Variable). This vulnerability category was selected due to its prevalence in real-world software and its relatively localized manifestation within functions, making it an ideal starting point for evaluating our proof-of-concept framework.

### C. Disassembly and Normalization

Each binary was disassembled using `objdump` with the Intel syntax option, producing x86-64 instruction listings stripped of symbolic or debugging information. To reduce variability and support static analysis, each instruction was parsed and tokenized into one of several abstract operation types:

- **READ**: Stack-relative memory reads (e.g., `mov eax, [rbp-0x4]`)
- **WRITE**: Stack-relative memory writes (e.g., `mov [rbp-0x4], eax`)
- **CALL**: Function calls (e.g., `call <memset>`)
- **END**: Function returns (e.g., `ret`)
- **NOOP**: Non-memory-affecting instructions (e.g., `add eax, 1`)

To improve robustness, a lightweight parsing routine was used to extract operand values and filter for local memory accesses (primarily `[rbp+x]` and `[rsp+x]`). Additional heuristics were used to distinguish between function arguments and local variables based on operand offsets and context.

### D. Motivation for Preprocessing Choices

The abstraction of raw instructions into a small set of semantic tokens was designed to simplify the task of FSM-based analysis. Unlike machine learning models, FSMs operate deterministically and require clearly defined state transitions. Therefore, tokenizing disassembly into memory-affecting actions (READ, WRITE) allows the FSM to reason about initialization state without being distracted by irrelevant instructions such as arithmetic operations, register moves, or optimization artifacts.

Rather than overfitting to opcode patterns or complex control flow, this tokenization strategy ensures that the FSM operates on a clean, semantically meaningful trace of memory interactions, which is essential for reliably identifying uninitialized variable usage in a static context.

## IV. FSM Design and Implementation

### A. Finite State Machine Model

The core of this project is a Finite State Machine designed to track the initialization state of stack variables across a disassembled function. The FSM operates over a sequence of abstracted tokens representing memory interactions, derived from normalized x86-64 instructions. Each variable (represented by a memory operand such as `[rbp-0x4]`) is individually tracked through one of the following implicit states:

- **Unallocated**: The variable has not been referenced by any instruction.
- **Allocated but Uninitialized**: The variable is accessed but has not yet been written to.
- **Initialized**: The variable has been written to.

The FSM maintains two sets: an `allocated` set and an `initialized` set. Variables move between these sets based on observed memory operations (e.g., `WRITE` or `READ`) as the instruction stream is processed.

### B. State Transitions

Transitions are driven by the abstract token stream. For each memory operand encountered:

- A `WRITE` operation adds the operand to both the `allocated` and `initialized` sets.
- A `READ` operation adds the operand to the `allocated` set if not already present. If the operand is not found in the `initialized` set, it is flagged as a use-before-initialization vulnerability.
- A `CALL` operation is heuristically treated as a potential zero-initialization (e.g., `memset`) when the callee matches known safe functions.
- The `END` token triggers a vulnerability check at function return.

This FSM design permits lightweight, linear-time processing of function-level disassembly and can be reused across all functions within a binary.

### C. Operand Extraction and Memory Modeling

To support accurate tracking, operand extraction focuses on dereferenced stack locations. Only operands using `rbp` or `rsp` (e.g., `[rbp-0x4]`) are considered, as these most often represent local variables or arguments. Memory operands are normalized to preserve consistent naming and reduce false negatives due to syntactic variation.

Function arguments, often accessed via positive `rbp` offsets, are optionally treated as initialized depending on their context, reducing false positives.

### D. Function Boundary Handling

To avoid analyzing interprocedural dependencies across unknown functions, each analysis pass is restricted to a single function. Calls to external or unknown routines (e.g., `malloc`, `memset`) are handled conservatively: functions with known side effects are modeled as safe, while unknown functions are skipped. Internal calls to other Juliet functions are supported recursively via in-file function boundary detection.

### E. Implementation Details

The FSM is implemented in Python, using a modular design composed of:

- A tokenizer module to normalize and abstract disassembly lines.
- An FSM class to model variable states and transitions.
- A runner module to manage function-level traversal and aggregate results across samples.

This implementation allows flexible testing of heuristics, expansion to new token types, and integration with external metrics and reporting systems. Evaluation is performed using pre-tokenized Juliet disassembly samples in a fully automated pipeline.

## V. Evaluation and Results

### A. Evaluation Methodology

To evaluate the effectiveness of the FSM-based vulnerability detector, we conducted a full sweep across the CWE-457 subset of the Juliet Test Suite. This evaluation focused on binary disassembly without access to source code or compiler metadata. For each disassembled function, our FSM model was executed to determine whether a vulnerability (use of uninitialized variable) was detected.

We measured performance using four standard classification metrics:

- **Accuracy**: The percentage of all correct predictions (safe or vulnerable).
- **Precision**: The proportion of predicted vulnerable functions that were actually vulnerable.
- **Recall**: The proportion of actual vulnerable functions that were correctly identified.
- **F1 Score**: The harmonic mean of precision and recall, indicating the overall balance between false positives and false negatives.

Additionally, we report the confusion matrix to provide a detailed view of classification errors.

### B. Quantitative Results

Evaluation was performed over 1,852 disassembled samples from the CWE-457 dataset:

- 926 **safe** (good) functions
- 926 **vulnerable** (bad) functions

TABLE I
FSM Detection Metrics on CWE-457

| Metric | Score |
|---|---|
| Accuracy | 0.6766 |
| Precision | 0.7935 |
| Recall | 0.4773 |
| F1 Score | 0.5961 |

The confusion matrix in Table II shows the distribution of true positives, false positives, true negatives, and false negatives.

TABLE II
Confusion Matrix

| | Predicted Safe | Predicted Vulnerable |
|---|---|---|
| **True Safe** | 811 | 115 |
| **True Vulnerable** | 484 | 442 |

### *C. Analysis*

The FSM detector achieved high **precision** (0.7935), indicating that when it predicted a vulnerability, it was correct the majority of the time. However, the relatively low **recall** (0.4773) reveals a tendency to miss many true positives. This suggests that the FSM is conservative in its classifications, favoring false negatives over false positives.

This tradeoff is characteristic of lightweight static analysis: the FSM performs well at detecting clear violations (e.g., reads with no corresponding prior write), but lacks the expressiveness to model more complex control flow paths, partial initializations, or indirect register tracking.

### *D. Summary*

Despite its simplicity, the FSM model correctly identified 442 of 926 vulnerable cases while maintaining a low false positive rate. These results demonstrate that FSM-based binary analysis can detect CWE-457 patterns using only tokenized disassembly. While the overall F1 Score (0.5961) is modest, the high precision shows promise for use in low-noise triage or as a pre-processing stage for more advanced analysis tools.

## VI. Limitations and Future Work

### *A. Limitations*

While the FSM-based approach to binary vulnerability detection shows promising results in precision, it is important to acknowledge its limitations:

- **Limited Recall**: The FSM detector missed over half of the vulnerable samples in the dataset, resulting in a recall of just 0.4773. This suggests that the model is not sensitive to all valid forms of uninitialized variable usage, particularly when complex control flow or partial initialization is involved.
- **Simplistic State Model**: The FSM implementation used in this project models only basic memory interactions (e.g., `READ`, `WRITE`) without deeper tracking of control dependencies, register flow, or inter-procedural state transitions. These simplifications reduce detection coverage.
- **Static Disassembly Only**: The analysis was performed solely on disassembled static binaries. No symbolic or dynamic information was used, meaning the FSM could not reason about actual execution paths or conditional logic.
- **No Handling of Indirect Memory Access**: Indirect variable access via register dereferencing or heap memory was only partially handled. Many vulnerable operations do not involve straightforward `rbp/rsp`-relative addressing and were missed as a result.
- **Single-Vulnerability Focus**: The system was designed only for detecting CWE-457 (Use of Uninitialized Variable). It does not generalize to other vulnerability classes such as buffer overflows, use-after-free, or integer overflow.

### *B. Future Work*

Several avenues exist to extend and improve this approach:

- **Enhanced Operand Tracking**: Introduce symbolic or SSA-based operand tracking to follow variable initialization across control branches and register transfers.
- **Control Flow Awareness**: Incorporate basic control flow analysis to understand variable state under conditional execution and across multiple function scopes.
- **FSM Extension for Partial Initialization**: Extend the FSM to explicitly model partial initialization, especially for arrays and structs.
- **Integration with Other Analyses**: Combine FSM-based heuristics with more expressive techniques such as taint analysis, abstract interpretation, or machine learning models for hybrid detection.
- **Multiple CWE Support**: Expand the FSM library to detect patterns corresponding to other common vulnerability types, using CWE templates as formal specifications.
- **Real-World Binary Evaluation**: Test the FSM system on optimized or obfuscated binaries from real software projects to better assess practical performance.

This project demonstrates the viability of using FSMs for lightweight static detection of uninitialized variables, but additional refinements are necessary to increase recall, broaden applicability, and adapt the system to more realistic code environments.

## VII. Conclusion

This project explored the feasibility of using a Finite State Machine framework for detecting vulnerabilities in compiled binaries without access to source code. Focusing specifically on CWE-457 (Use of Uninitialized Variable), we implemented a static analysis tool that tokenizes x86-64 disassembly and simulates variable state transitions across function scopes.

Evaluation across 1,852 disassembled samples from the Juliet Test Suite revealed a precision of 0.7935, indicating the FSM system was often correct when flagging vulnerabilities. However, the recall rate of 0.4773 revealed a tendency to miss many vulnerable cases, likely due to simplifications in the operand tracking and lack of control-flow reasoning.

These results suggest that FSMs can be used to identify certain types of vulnerabilities with a relatively high degree of confidence, but further enhancements are required to improve sensitivity and generalizability. Despite modest overall performance, the project provides a foundation for future hybrid vulnerability detection systems that combine deterministic state modeling with richer forms of analysis.

This proof of concept highlights the potential of FSM-based detection as a lightweight, explainable, and easily extensible tool for binary analysis, particularly when used in conjunction with more expressive static or dynamic techniques.

## Acknowledgments

The author would like to thank the creators and maintainers of the Juliet Test Suite for providing a standardized benchmark for software vulnerability research, as well as

Alexander Richardson (arichardson) for his work in adapting the Juliet Test Suite for Unix-like environments, which greatly streamlined data preparation and automation for this project. Appreciation is also given to the broader research community whose work on symbolic execution, abstract interpretation, and state machine–based program analysis informed the direction of this project. Their contributions have laid the groundwork for exploring scalable and interpretable binary analysis techniques.

## References

[1] S. Pordanesh and B. Tan, "Exploring the efficacy of large language models (gpt-4) in binary reverse engineering," 2024. [Online]. Available: https://arxiv.org/abs/2406.06637

[2] B. Anderson, C. Storlie, M. Yates, and A. McPhall, "Automating reverse engineering with machine learning techniques," in *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, ser. AISec '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–112. [Online]. Available: https://doi.org/10.1145/2666652.2666665

[3] S. Taviss, S. H. H. Ding, M. Zulkernine, P. Charland, and S. Acharya, "Asm2seq: Explainable assembly code functional summary generation for reverse engineering and vulnerability analysis," *Digital Threats*, vol. 5, no. 1, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3592623

[4] M. Udeshi, P. Krishnamurthy, H. Pearce, R. Karri, and F. Khorrami, "Remaqe: Reverse engineering math equations from executables," *ACM Trans. Cyber-Phys. Syst.*, vol. 8, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3699674

[5] D. van Zeeland, "Reverse-engineering state machine diagrams from legacy c-code," Master's Thesis, Eindhoven University of Technology, 2009. [Online]. Available: https://research.tue.nl/en/studentTheses/reverse-engineering-state-machine-diagrams-from-legacy-c-code

[6] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157. [Online]. Available: https://ieeexplore.ieee.org/document/7546500

[7] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *2017 NDSS*, 01 2017. [Online]. Available: https://www.researchgate.net/publication/316913243$_{R}amblr_{M}aking_{R}eassembly_{G}reat_{A}gain$

[8] N. I. of Standards and Technology, "Juliet test suite," https://samate.nist.gov/SARD/test-suites/112, 2024, accessed: 2025-04-25.

[9] A. Richardson, "Juliet test suite c (linux port)," https://github.com/arichardson/juliet-test-suite-c, 2024, accessed: 2025-04-25.