TraceGuard: Taint-Guided Symbolic Execution

Bachelor Thesis Presentation

Ruben Hutter

July 11, 2025

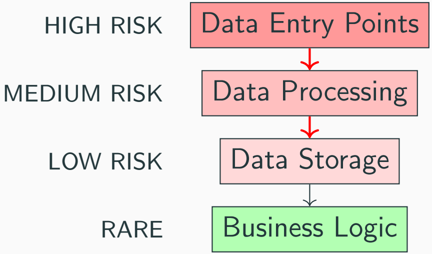
University of Basel, Faculty of Science
Department of Mathematics and Computer Science

Today's Journey

1. **The Challenge:** Software vulnerabilities and current detection limits
2. **The Problem:** Why symbolic execution struggles
3. **The Insight:** Taint-guided exploration concept
4. **The Solution:** TraceGuard's approach and implementation
5. **The Evidence:** Evaluation results and performance gains
6. **Seeing It Work:** Live demonstration
7. **Looking Forward:** Future directions and broader impact

The Problem

Where Do Software Vulnerabilities Actually Hide?



Vulnerability Hotspots:

- **Data Entry:** Network I/O, file parsing, user input
- **Data Processing:** String operations, format parsing, validation
- **Data Storage:** Memory allocation, buffer operations
- **Business Logic:** Complex algorithms, decision trees

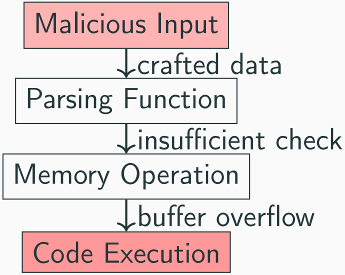
Risk decreases as data moves away from external sources

Common Vulnerability Patterns

Input-Related Vulnerabilities:

- **Buffer overflows:** `strcpy(small_buf, user_input)`
- **Format string bugs:** `printf(user_string)`
- **Injection attacks:** SQL, command injection
- **Integer overflows:** Size calculations from input

Example Attack Flow:

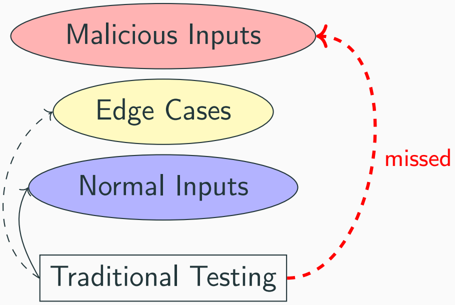


Key Insight: Following the data flow from input to vulnerability

The Challenge: Finding These Vulnerabilities

Traditional Testing Approaches:

- **Manual code review:** Time-intensive, incomplete coverage
- **Unit testing:** Limited to expected inputs
- **Integration testing:** Misses edge cases
- **Static analysis:** High false positive rates



The Gap: Security vulnerabilities often trigger under specific, unexpected input conditions

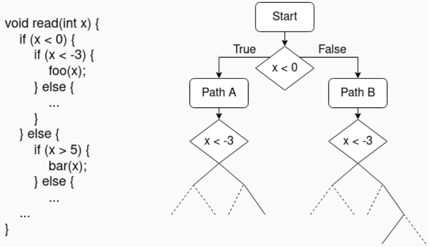
Symbolic Execution - The Promise

Goal: Find all possible bugs automatically

Method: Treat inputs as mathematical symbols

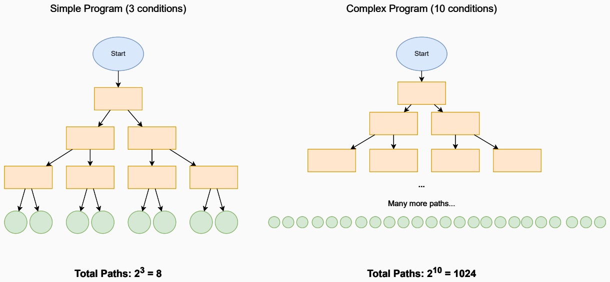
Power: Can generate test cases for any reachable code

Advantage: Handles complex conditions that random testing cannot reach



Instead of testing with specific values, explore ALL possible values

The Reality - Path Explosion Problem



- 3 conditions $\rightarrow$ 8 possible paths
- 10 conditions $\rightarrow$ 1,024 possible paths
- 20 conditions $\rightarrow$ 1,048,576 possible paths
- Real programs: Millions of conditions = computational infeasibility

Exponential growth kills practical application

Comparing Approaches - Fuzzing vs. Symbolic Execution

Fuzzing	**Symbolic Execution**
Fast, lightweight	Slow, resource-intensive
Shallow bug discovery	Deep path exploration
Random/guided input generation	Systematic path coverage
Struggles with complex conditions	Handles complex logic well

Challenge

Both struggle with scale, but in different ways:

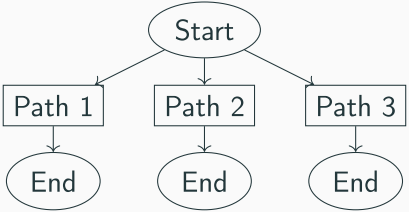
- **Fuzzing:** Hard to reach deep code paths
- **Symbolic Execution:** Exponential path explosion

We need the systematic power of symbolic execution with better efficiency

The Insight

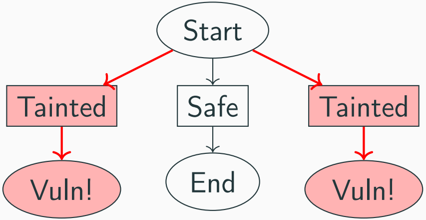
The Core Insight - Taint as a Guide

Classical Exploration:



Explores everything

Taint-Guided Exploration:



Follows the data

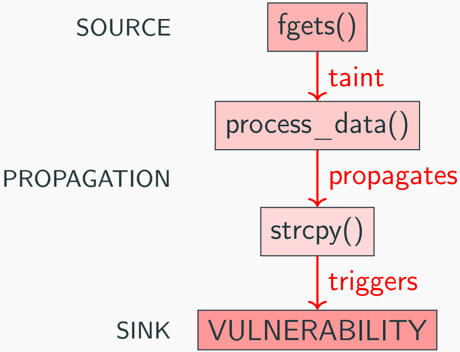
Key Realization

- Not all execution paths are equally likely to contain vulnerabilities
- Paths processing user-controlled data deserve priority

What is Taint Analysis?

Definition: Track data derived from untrusted sources

- **Sources:** User input functions (fgets, scanf, network recv)
- **Propagation:** Through assignments, function calls, memory operations
- **Sinks:** Security-sensitive operations (strcpy, system calls)




```
char buffer[100];
fgets(buffer, 100, stdin);  // TAINT SOURCE
process_data(buffer);       // TAINT PROPAGATES
strcpy(dest, buffer);       // POTENTIAL VULNERABILITY
```

Traditional Approach vs. TraceGuard

Traditional Symbolic Execution:

- Explore all paths uniformly
- Hope to eventually reach vulnerable code
- Often times out before finding bugs
- Wastes resources on irrelevant paths

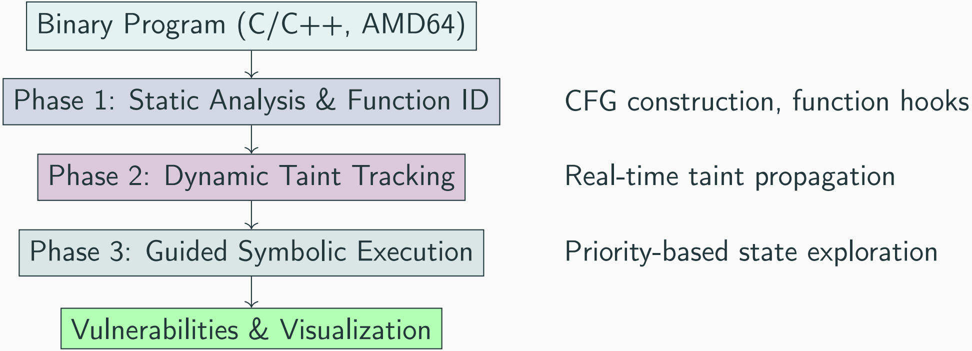
TraceGuard's Taint-Guided Approach:

- Real-time taint tracking during symbolic execution
- Dynamic prioritization based on taint interaction
- Focus computational resources on security-relevant paths
- Find more vulnerabilities with focused exploration

Key Innovation: Integration, not post-processing

The Solution

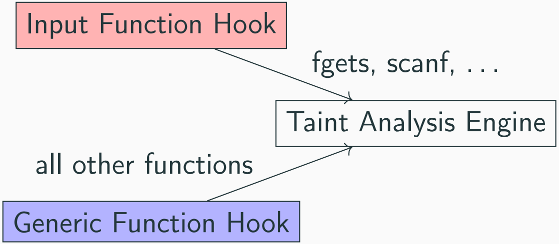
TraceGuard Architecture - Overview



Integration Point

- Built on Angr symbolic execution framework

How TraceGuard Works - Function Hooking

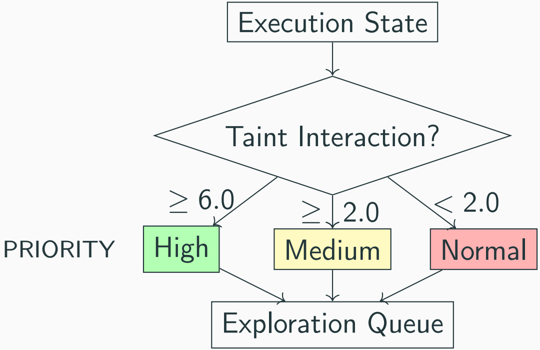


Function Hooking Strategy:

- **Input Function Hooks:** Detect taint introduction
- **Generic Function Hooks:** Monitor taint propagation
- **Real-time Detection:** Analyze register/memory contents
- **Taint Marking:** Create symbolic variables with taint IDs

Example: When `fgets()` is called → Create `"taint_source_fgets_001"` symbolic variable

Dynamic State Scoring Algorithm



Scoring Components:

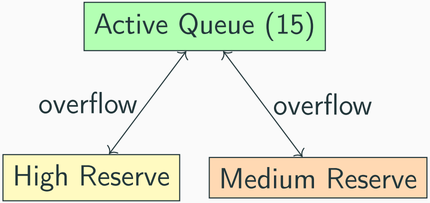
- **Base Score:** From taint interactions ($+20.0$ for input functions)
- **Bonuses:** Execution within tainted functions ($+3.0$)
- **Penalties:** Excessive execution depth ($\times 0.95$ for deep paths)
- **Classification:** High (≥ 6.0), Medium (≥ 2.0), Normal (< 2.0)

Result: Three-tier exploration queue prioritizing security-relevant states

Exploration Strategy

Bounded Exploration:

- Maximum 15 active states
- Priority queues: High → Medium → Normal
- Dynamic replacement: New high-priority states replace low-priority ones
- Overflow management: Store excess states in reserve pools



Advantage

- Prevents path explosion while maintaining security focus

The Results

Evaluation Methodology

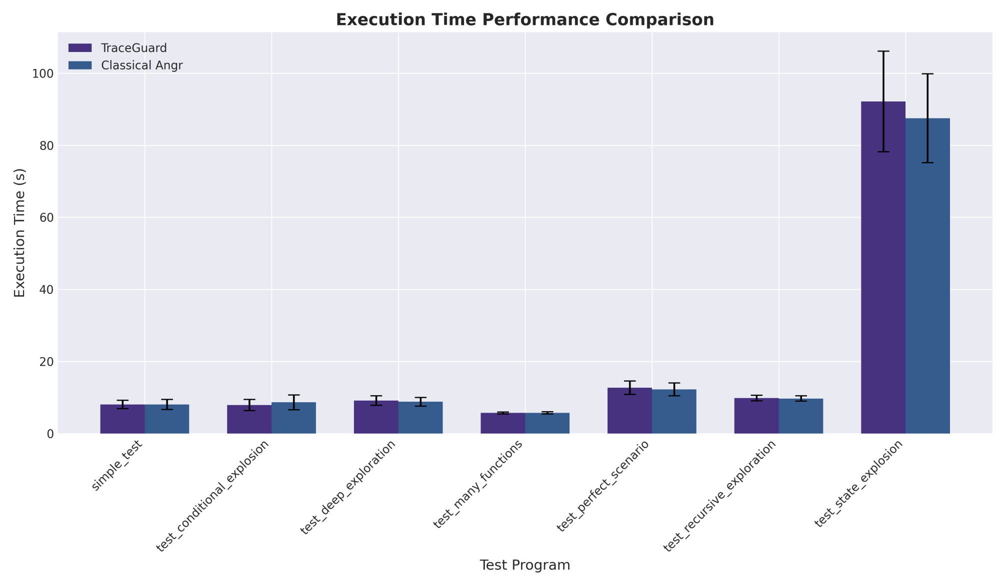
Test Suite Design

- **7 synthetic programs** targeting different challenges
- **Known vulnerabilities** with clear taint flow patterns
- **Controlled comparison:** TraceGuard vs. Classical Angr strategy

Programs Test:

- Simple baselines, conditional explosions, deep exploration
- Multi-function analysis, perfect scenarios, recursive calls
- State explosion stress test

Key Results - Execution Time Performance

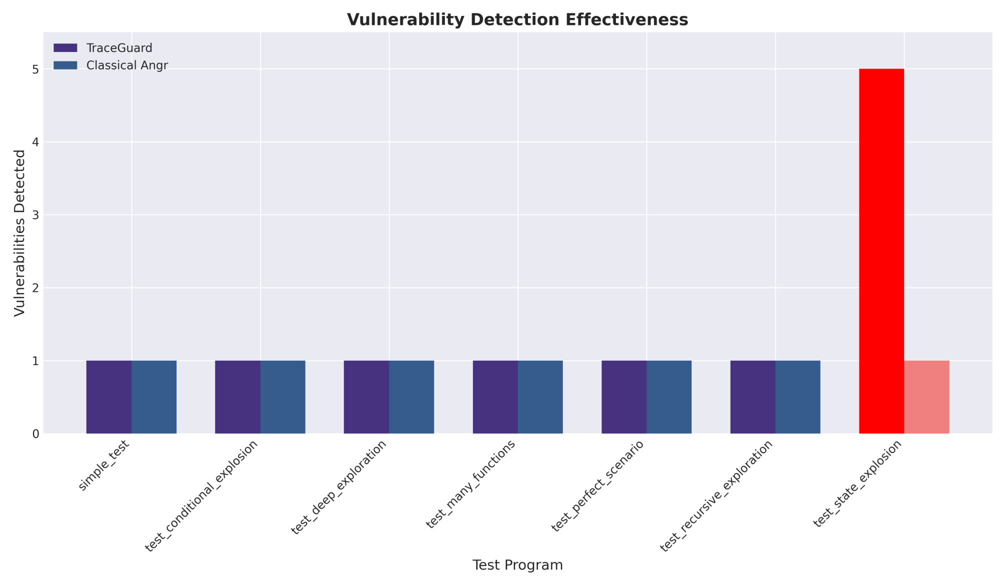


Key Results - Execution Time Performance

Key Insights

- **Competitive Performance:** TraceGuard shows -5.3% to +8.5% time variation compared to Classical Angr.
- **Complex Branching Improvement:** Achieved an 8.5% time improvement in the `test_conditional_explosion` scenario.
- **Scalability in Stress Test:** Maintained comparable execution time in the `test_state_explosion` scenario, despite its complexity.

Vulnerability Detection Effectiveness

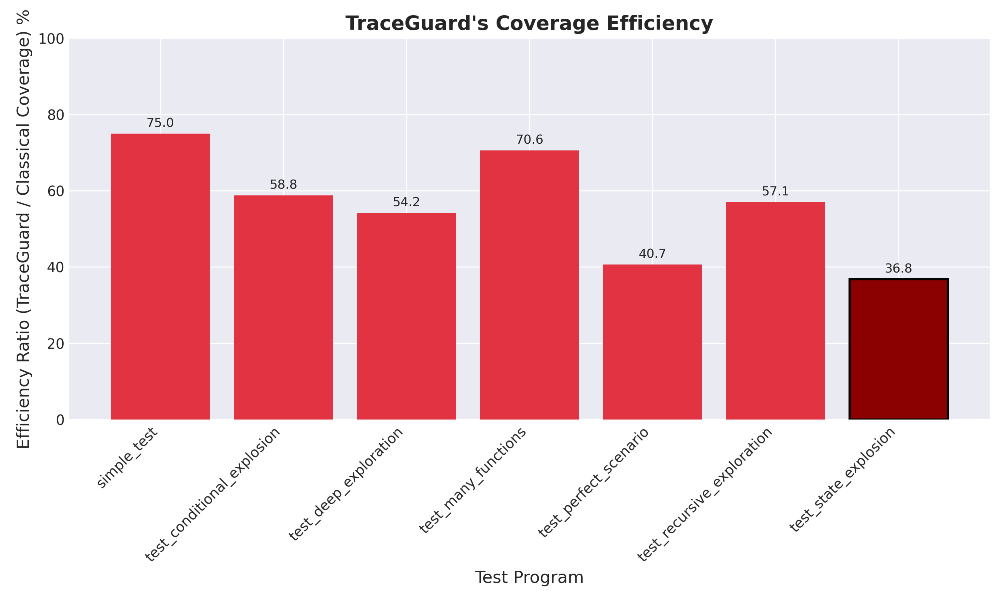


Vulnerability Detection Effectiveness

Perfect Detection with Superior Performance

- **100% Vulnerability Coverage**
- **Strategic Advantage in Complexity:** TraceGuard found $5\times$ **more vulnerabilities** in the challenging `test_state_explosion` scenario.
- **Consistent and Enhanced Reliability:** TraceGuard maintains consistent detection for all test types while providing a significant leap in highly complex environments.

Coverage vs. Effectiveness



Efficiency: Quality Over Quantity

Paradigm Shift

- **Traditional Metric:** More coverage $\Longrightarrow$ better analysis
- **Our Finding:** Focused exploration $\Longrightarrow$ more critical vulnerabilities
- **The Proof (Example: `test_state_explosion`):**
  - TraceGuard found 5$\times$ **more vulnerabilities** with only **36.8% of Classical's coverage**.

Quality of exploration matters more than quantity

Live Demo

Live Demo - TraceGuard in Action

Demo Setup

Target: `examples/program6`

What we'll see:

1. Taint source detection (fgets)
2. Real-time taint propagation tracking
3. Guided exploration prioritization
4. Vulnerability discovery and reporting
5. Interactive visualization

[LIVE DEMO]

Future Directions

Future Research Directions

Enhanced Configuration:

- **Header file integration:** Replace meta files with automatic C/C++ header parsing
- **Adaptive parameters:** Dynamic configuration based on program complexity
- **Custom entry points:** Support user-defined analysis starting points
- **Performance tuning:** Systematic optimization of scoring parameters

Architecture & Scale:

- **Multi-architecture:** Complete ARM and x86 platform support
- **Library analysis:** Support for analyzing library interfaces
- **Real-world validation:** Large-scale commercial software testing
- **Input source expansion:** Network protocols, file formats, IPC

Broader Impact & Applications

Academic Impact

- **Methodology:** Security-aware program analysis paradigm
- **Tool:** Open source platform for continued research
- **Validation:** Empirical evidence for taint-guided approaches

Conclusion

Summary and Conclusion

What I Achieved

- **Addressed fundamental limitation:** Path explosion in symbolic execution through intelligent prioritization
- **Demonstrated effectiveness:** 100% vulnerability detection with a remarcable improvement in challenging scenarios
- **Practical implementation:** Complete system built on Angr framework, ready for deployment
- **Research foundation:** Platform for continued security-aware program analysis research

Summary and Conclusion

Key Results

- **Increased vulnerability discovery:** Found more vulnerabilities while exploring only 36.8% to 75.0% of basic blocks
- **Competitive performance:** Maintained execution times within 10% of classical approaches
- **Validated approach:** Proved that security-focused exploration outperforms uniform path coverage

Questions?

Thank You

Questions and Discussion

TraceGuard: Taint-Guided Symbolic Execution
for Enhanced Binary Analysis

Ruben Hutter
University of Basel
Supervisor: Prof. Dr. Christopher Scherb

Tool: https://github.com/ruben-hutter/TraceGuard