University of Basel

Optimizing Symbolic Execution Through Taint Analysis and Path Prioritization

Bachelor thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Databases and Information Systems (DBIS) Group
https://dbis.dmi.unibas.ch/

Examiner: Dr. Marco Vogt
Supervisor: Prof. Dr. Christopher Scherb

Ruben Hutter
ruben.hutter@unibas.ch
2020-065-934

02.07.2025

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Christopher Scherb for his supervision and guidance throughout this thesis. His expertise in program analysis and symbolic execution provided essential direction for this research.

I am grateful to Dr. Marco Vogt for his role as examiner and for facilitating the opportunity to pursue this research topic. His feedback and suggestions helped refine both the technical approach and the presentation of this work.

I would like to thank Ivan Giangreco for providing the LaTeX thesis template used for this document, which greatly facilitated the formatting and structure of this work.

I also acknowledge my fellow student Nico Bachmann for developing the Schnauzer visualization library, which enhanced the presentation and analysis of the results in this work.

I thank my family and friends for their encouragement and support during my studies, which made completing this thesis possible.

Finally, I acknowledge the developers of the angr binary analysis framework, whose comprehensive platform enabled the implementation of the techniques presented in this work.

Abstract

Symbolic execution is a powerful program analysis technique widely used for vulnerability discovery and test case generation. However, its practical application is often hampered by scalability issues, primarily due to the "path explosion problem" where the number of possible execution paths grows exponentially with program complexity. This thesis addresses this fundamental challenge by proposing an optimized approach to symbolic execution that integrates taint analysis and path prioritization.

The core contribution is a novel exploration strategy that moves away from uniform path exploration towards targeted analysis of security-critical program behaviors. The approach prioritizes execution paths originating from memory allocations and user input processing points, as these represent common sources of vulnerabilities. By leveraging dynamic taint analysis, the system identifies and tracks data flow from these critical sources, enabling the symbolic execution engine to focus computational resources on paths influenced by tainted data while deprioritizing paths with no dependency on external inputs.

The implementation integrates this taint-guided exploration strategy with the angr symbolic execution framework, introducing a scoring mechanism that dynamically adjusts path prioritization based on taint propagation. The effectiveness of this optimization is evaluated through comparative analysis, examining runtime efficiency, path coverage quality, and vulnerability discovery capabilities. Results demonstrate that this approach can significantly reduce the search space while maintaining or improving the detection of security-relevant program behaviors, making symbolic execution more practical for large and complex software systems.

Table of Contents

1 Introduction

In today's interconnected digital landscape, software security has become a critical concern as applications handle increasingly sensitive data and operate in hostile environments. The discovery of security vulnerabilities before deployment is essential to prevent exploitation by malicious actors, yet traditional testing approaches often fail to comprehensively explore all possible execution scenarios, leaving potential vulnerabilities undiscovered.

Among program analysis techniques, symbolic execution has emerged as a particularly powerful approach for automated vulnerability discovery. Unlike traditional testing that executes programs with concrete input values, symbolic execution treats inputs as mathematical symbols and tracks how these symbols propagate through program computations. When encountering conditional branches, the symbolic execution engine explores multiple possible paths simultaneously, building a comprehensive map of program behaviors. This systematic exploration capability makes symbolic execution especially valuable for security analysis, as it can automatically generate test cases that reach deep program states and trigger complex vulnerabilities such as buffer overflows, integer overflows, and format string bugs.

Challenges in Symbolic Execution. Despite its theoretical power, symbolic execution faces a fundamental scalability challenge known as the path explosion problem. As program complexity increases, the number of possible execution paths grows exponentially, quickly overwhelming computational resources and rendering the analysis intractable for real-world software systems. Modern applications can generate millions of execution paths from relatively small input variations, making exhaustive analysis computationally prohibitive.

The path explosion problem is exacerbated by current symbolic execution engines that typically employ uniform exploration strategies, treating all program paths with equal priority regardless of their potential security relevance, as Figure 1.1 illustrates. This approach fails to recognize that paths processing user-controlled data are significantly more likely to contain vulnerabilities than paths handling only internal program state. Consequently, significant computational resources are often spent analyzing auxiliary program logic while security-critical paths that process external inputs receive no special attention.

Consider, for example, a network service that accepts client connections, reads incoming

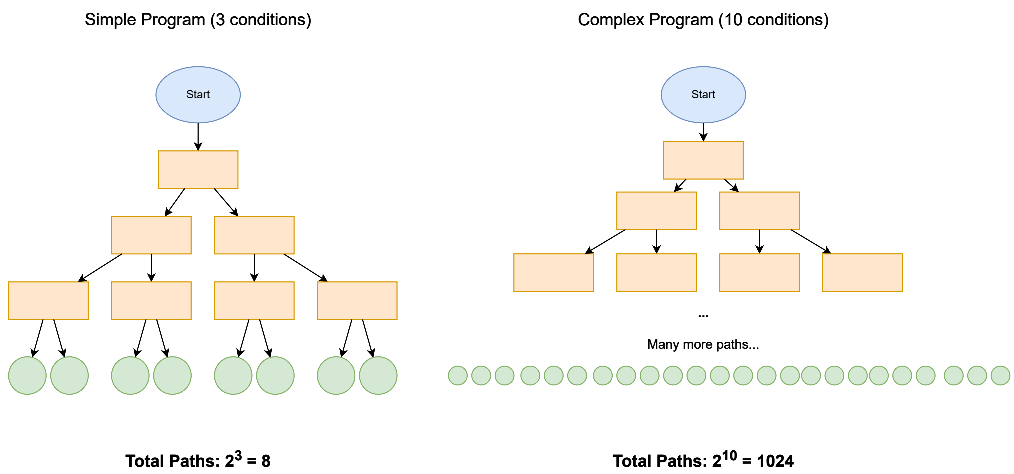

Figure 1.1: Illustration of the path explosion problem: As program complexity increases from 3 to 10 conditions, the number of possible execution paths grows exponentially from 8 to 1024 paths.

data via network sockets, performs input validation through multiple parsing layers, and eventually stores results using memory copy operations. Traditional symbolic execution would explore all execution paths with equal priority, including those that handle only internal configuration data or administrative functions that never process user input. A security-focused approach should recognize that paths flowing from network input through data processing to memory operations deserve higher priority due to their potential for buffer overflows, injection attacks, and other input-related vulnerabilities.

Thesis Overview. This work presents TraceGuard[1], an approach that integrates taint analysis with symbolic execution to enable intelligent path prioritization. Our methodology identifies and tracks data flow from critical sources such as user inputs, guiding the symbolic execution engine to focus computational resources on paths most likely to exhibit security-relevant behaviors.

The key insight driving this approach is that not all execution paths are equally valuable for security analysis—paths that interact with user-controlled data are significantly more likely to harbor vulnerabilities than those processing only internal program state. TraceGuard operationalizes this insight through a dynamic taint scoring mechanism that quantifies the security relevance of each symbolic execution state. By prioritizing states with higher taint scores, the symbolic execution engine directs its computational resources toward program regions most likely to contain security vulnerabilities, fundamentally transforming symbolic execution from an exhaustive search into a guided exploration strategy.

The main contributions of this thesis are:

- **Taint-Guided Path Prioritization:** An integration of dynamic taint analysis with symbolic execution that uses taint propagation patterns to intelligently prioritize exploration of security-relevant execution paths.
- **Custom Angr Exploration Technique:** Implementation of

[1] https://github.com/ruben-hutter/TraceGuard

`TaintGuidedExploration`, a specialized exploration strategy that extends Angr's symbolic execution capabilities with security-focused path prioritization.

- **Function-Level Taint Tracking:** A comprehensive taint tracking system that monitors input functions, tracks taint propagation through function calls, and maintains detailed taint information throughout program execution.
- **Adaptive Scoring Algorithm:** A scoring mechanism that dynamically adjusts path priorities based on real-time taint analysis results, enabling the symbolic execution engine to focus computational resources on the most promising program regions.
- **Intelligent Function Hooking System:** A sophisticated hooking mechanism that intercepts function calls to analyze parameter taint status, allowing selective execution of only security-relevant code paths.
- **Practical Implementation and Validation:** A complete implementation using the angr symbolic execution framework, with comprehensive testing demonstrating the effectiveness of taint-guided exploration.

The effectiveness of this optimization is evaluated through controlled experiments comparing TraceGuard against standard Angr symbolic execution techniques using custom-designed test programs with known taint flow patterns. The evaluation examines key metrics including execution time, function call efficiency, and vulnerability detection reliability, with initial testing indicating significant improvements in analysis efficiency while maintaining comprehensive vulnerability detection capabilities.

This thesis focuses on binary program analysis using the angr symbolic execution framework, targeting user-space applications written in C/C++ and compiled for x86-64 architectures. The evaluation methodology centers on custom-designed test programs that demonstrate clear taint flow patterns, specifically crafted to evaluate TraceGuard's ability to distinguish between tainted and untainted execution paths.

The thesis is organized as follows:

- **Chapter 2** provides essential background on symbolic execution, taint analysis, and the angr framework.
- **Chapter 3** surveys related work in symbolic execution optimization and taint analysis techniques.
- **Chapter 4** presents the conceptual framework and theoretical algorithms underlying TraceGuard's taint-guided exploration strategy.
- **Chapter 5** details the practical implementation, including integration with angr and the design of the scoring mechanism.
- **Chapter 6** presents a comprehensive evaluation comparing TraceGuard's performance against standard symbolic execution techniques.
- **Chapter 7** concludes with a summary of contributions and research implications.
- **Chapter 8** explores potential extensions and future research directions.

2
Background

This chapter establishes the theoretical foundations necessary for understanding the taint-guided symbolic execution optimization presented in this thesis. We examine symbolic execution, program vulnerability analysis, taint analysis, control flow analysis, and the Angr[2] framework.

2.1 Symbolic Execution

Symbolic execution is a program analysis technique that explores execution paths by using symbolic variables instead of concrete inputs. The program state consists of symbolic variables, path constraints, and a program counter. When execution encounters a conditional branch, the engine explores both branches by adding appropriate constraints to the path condition.

A fundamental challenge in symbolic execution is the path explosion problem. As program complexity increases, the number of possible execution paths grows exponentially, making exhaustive exploration computationally intractable. This scalability issue particularly affects real-world applications with complex control flow structures and deep function call hierarchies. Research has shown that symbolic execution tools designed to optimize statement coverage often fail to cover potentially vulnerable code due to complex system interactions and scalability issues of constraint solvers [6].

Traditional symbolic execution typically employs a forward approach, starting from the program's entry point and exploring paths toward potential targets. However, this method may struggle to reach deeply nested functions or specific program locations of interest. Backward symbolic execution, conversely, begins from target locations and works backwards to identify input conditions that can reach those targets. Compositional approaches combine both techniques by analyzing individual functions in isolation and then reasoning about their interactions.

[2] https://angr.io/

2.2 Program Vulnerability Analysis

Software vulnerabilities represent flaws in program logic or implementation that can be exploited by malicious actors to compromise system security. Understanding these vulnerabilities is crucial for developing effective analysis techniques that can detect them before deployment.

Traditional testing approaches often fail to discover these vulnerabilities because they typically occur only under specific input conditions that are unlikely to be encountered through random testing. Static analysis can identify potential vulnerabilities but often produces high false positive rates due to conservative approximations required for soundness. Dynamic analysis provides precise information about actual program execution but is limited to the specific inputs and execution paths exercised during testing.

Symbolic execution addresses these limitations by systematically exploring multiple execution paths and generating inputs that trigger different program behaviors. However, the path explosion problem means that uniform exploration strategies may spend significant computational resources on paths that are unlikely to contain security vulnerabilities. This motivates the development of security-focused analysis techniques that prioritize exploration of paths involving user-controlled data, as these represent the primary attack vectors for most software vulnerabilities.

2.3 Taint Analysis

Taint analysis tracks the propagation of data derived from untrusted sources throughout program execution. Data originating from designated sources (such as user input functions like `fgets`, `gets`, `read`, or `scanf`) is marked as tainted. The analysis tracks how this tainted data flows through assignments, function calls, and other operations. When tainted data reaches a security-sensitive sink (such as buffer operations or system calls), the analysis flags a potential vulnerability.

The propagation rules define how taint spreads through different operations: assignments involving tainted values result in tainted variables, arithmetic operations with tainted operands typically produce tainted results, and function calls with tainted arguments may result in tainted return values depending on the function's semantics. Dynamic taint analysis performs tracking during program execution, providing precise information about actual data flows while considering specific calling contexts and program states, resulting in reduced false positives compared to static analysis approaches.

2.4 Control Flow Analysis

Control flow analysis constructs and analyzes control flow graphs (CFGs) representing program structure. CFG nodes correspond to basic blocks of sequential instructions and edges represent possible control transfers between blocks. This representation enables systematic analysis of program behavior and reachability properties.

Static analysis constructs CFGs by examining program code without execution, analyzing structure and control flow based solely on the source code or binary representation.

This approach offers comprehensive coverage and efficiency, enabling examination of all statically determinable program paths without requiring specific input values. However, static analysis faces limitations including difficulty with indirect call resolution and potential false positives due to conservative approximations required for soundness.

Dynamic analysis executes the program and collects runtime information, providing precise information about actual program behavior and complete execution context. This approach eliminates many false positives inherent in static analysis and validates that control flow relationships are actually exercised under realistic conditions. However, dynamic analysis results depend heavily on input quality and coverage.

A Call Graph represents function call relationships within a program, where each node corresponds to a function and each directed edge represents a call relationship. Call graphs serve important purposes including program understanding, entry point identification, reachability analysis, and complexity assessment. Call graphs prove valuable for path prioritization strategies, enabling identification of functions reachable from tainted input sources and assessment of their relative importance in program execution flow.

2.5 Angr Framework

Angr is an open-source binary analysis platform providing comprehensive capabilities for static and dynamic program analysis [7]. The platform supports multiple architectures and provides a Python-based interface for research and education [8]. Key components include the *Project* object representing the binary under analysis with access to contents, symbols, and analysis capabilities; the *Knowledge Base* storing information gathered during analysis including function definitions and control flow graphs; the *Simulation Manager* handling multiple program states during symbolic execution and managing state transitions; and the *Solver Engine* interfacing with constraint solvers to determine path feasibility and solve for concrete input values.

Angr supports both static (`CFGFast`) and dynamic (`CFGEmulated`) CFG construction. Static analysis provides efficiency but may miss indirect calls, while dynamic analysis offers completeness at higher computational cost. The framework represents program states with register values, memory contents, path constraints, and execution history, providing APIs for state manipulation and exploration control through step functions and various exploration strategies including depth-first search, breadth-first search, and custom heuristics.

The framework's extensible architecture enables integration of custom analysis techniques, making it particularly suitable for implementing novel symbolic execution optimizations. The symbolic execution landscape includes numerous frameworks targeting different domains and applications, ranging from language-specific tools like KLEE[3] for LLVM[4] bitcode to specialized platforms for smart contract analysis. Angr's comprehensive binary analysis capabilities, multi-architecture support, and extensible Python-based architecture make it well-suited for implementing taint-guided exploration strategies.

[3] https://klee-se.org/
[4] https://llvm.org/

3 Related Work

This chapter surveys existing research in symbolic execution optimization and taint analysis techniques, positioning TraceGuard within the broader landscape of security-focused program analysis. We examine three primary categories of approaches: optimization strategies for managing path explosion, integration techniques combining multiple analysis methods, and security-oriented targeting approaches.

3.1 Optimization Approaches

3.1.1 State Space Reduction

The fundamental challenge in symbolic execution remains the path explosion problem, where the number of execution paths grows exponentially with program complexity. Kuznetsov et al. [2] introduced efficient state merging techniques to reduce symbolic execution states by combining states with similar path conditions. While effective for certain program structures, this approach lacks security-focused guidance, treating all execution paths equally regardless of their interaction with potentially malicious inputs.

Avgerinos et al. [1] proposed AEG (Automatic Exploit Generation), which prioritizes paths leading to exploitable conditions. However, AEG relies primarily on static analysis to identify potentially vulnerable locations, missing dynamic taint flow patterns that emerge only during execution.

Recent work by Yao and Chen [9] introduces Empc, a path cover-based approach that leverages minimum path covers (MPCs) to reduce the exponential number of paths while maintaining code coverage. However, Empc focuses on maximizing code coverage efficiently, while TraceGuard specifically targets security-relevant execution paths through taint propagation analysis.

3.1.2 Performance and Compositional Analysis

Poeplau and Francillon [5] developed optimizations for constraint solving by caching frequently encountered constraints. While these optimizations improve execution speed, they do not address the fundamental issue of exploring irrelevant paths that have no security implications.

Ognawala et al. [4] introduced MACKE, a compositional approach that analyzes functions in isolation before combining results. This technique encounters difficulties when taint flows cross function boundaries, as compositional analysis may miss inter-procedural data dependencies crucial for security analysis.

3.2 Integration Approaches

Dynamic taint analysis and symbolic execution represent complementary approaches that, when combined effectively, can overcome individual limitations. Schwartz et al. [6] provide a comprehensive comparison of dynamic taint analysis and forward symbolic execution, noting that taint analysis excels at tracking data flow patterns but lacks the path exploration capabilities of symbolic execution. Their work identifies the potential for hybrid approaches but does not present a concrete integration strategy.

Ming et al. [3] developed TaintPipe, a pipelined approach to symbolic taint analysis that performs lightweight runtime logging followed by offline symbolic taint propagation. While TaintPipe demonstrates the feasibility of combining taint tracking with symbolic reasoning, it operates in a post-processing mode rather than providing real-time guidance to symbolic execution engines.

Recent hybrid fuzzing approaches combine fuzzing with selective symbolic execution but lack sophisticated taint-awareness in their path prioritization strategies. These tools typically trigger symbolic execution when fuzzing coverage stagnates, rather than using taint information to proactively guide exploration toward security-relevant program regions.

3.3 Security-Focused Targeting and Research Gap

Security-focused symbolic execution approaches attempt to prioritize execution paths that are more likely to contain vulnerabilities. Static vulnerability detection approaches rely on pattern matching and dataflow analysis to identify potentially dangerous code locations, but cannot capture the dynamic taint propagation patterns that characterize real security vulnerabilities. Binary analysis frameworks like Angr [7] provide powerful symbolic execution capabilities but lack built-in security-focused exploration strategies.

The literature survey reveals critical limitations that TraceGuard addresses: (1) **Lack of Dynamic Taint-Guided Prioritization** - existing approaches focus on general path reduction rather than security-specific targeting; (2) **Reactive Integration Strategies** - current techniques use taint analysis in post-processing roles rather than as primary exploration drivers; (3) **Limited Security-Awareness** - optimizations treat all paths equally, failing to recognize higher vulnerability potential of taint-processing paths.

TraceGuard addresses these limitations through a novel real-time integration of dynamic taint analysis with symbolic execution, representing the first comprehensive framework for leveraging runtime taint information to intelligently prioritize security-relevant execution paths.

4

Taint-Guided Exploration

Having established the theoretical foundations in Chapter 2 and surveyed existing approaches in Chapter 3, this chapter presents the conceptual framework and algorithmic design of TraceGuard's taint-guided symbolic execution strategy. Rather than exploring all possible execution paths uniformly, TraceGuard prioritizes paths based on their interaction with potentially malicious user input, fundamentally addressing the path explosion problem through intelligent exploration guidance.

The core insight underlying this approach is that security vulnerabilities are significantly more likely to occur in code paths that process external, user-controlled data. By tracking taint flow from input sources and using this information to guide symbolic execution, TraceGuard focuses computational resources on security-relevant program regions while avoiding exhaustive exploration of paths that operate solely on trusted internal data.

4.1 Core Approach

TraceGuard operates as a specialized program built on the Angr framework that transforms symbolic execution from exhaustive path exploration into a security-focused analysis. The approach centers on four key mechanisms that work together to prioritize execution paths based on their interaction with potentially malicious user input.

Hook-Based Taint Detection: The system intercepts function calls during symbolic execution to identify when external data enters the program. Input functions like `fgets` and `scanf` are immediately flagged as taint sources, while other functions are monitored for tainted parameter usage.

Symbolic Taint Tracking: Tainted data is tracked through unique symbolic variable names and memory region mappings. When input functions create symbolic data, the variables receive distinctive "`taint_source_`" prefixes that persist throughout symbolic execution.

Dynamic State Prioritization: Each symbolic execution state receives a taint score based on its interaction with tainted data. States are classified into three priority levels that determine exploration order: high priority (score $\geq \tau_{high}$), medium priority ($\tau_{medium} \leq$ score $< \tau_{high}$), and normal priority (score $< \tau_{medium}$).

Exploration Boundaries: Multiple complementary techniques prevent path explosion: execution length limits, loop detection, and graduated depth penalties that naturally favor shorter paths to vulnerability-triggering conditions.

Throughout the following algorithms, we use configurable parameters to maintain generality: α_{input} represents the score bonus for input function interactions, $\beta_{tainted}$ denotes the bonus for execution within tainted functions, $\gamma_{penalty}$ specifies the depth penalty multiplication factor, $\delta_{threshold}$ defines the depth threshold for penalty application, σ_{min} sets the minimum exploration score, τ_{high} and τ_{medium} establish the priority classification thresholds, and k determines the maximum number of active states. In our implementation, these parameters are set to $\alpha_{input} = 5.0$, $\beta_{tainted} = 3.0$, $\gamma_{penalty} = 0.95$, $\delta_{threshold} = 200$, $\sigma_{min} = 1.0$, $\tau_{high} = 6.0$, $\tau_{medium} = 2.0$, and $k = 15$.

4.2 Taint Source Recognition

TraceGuard identifies taint sources by hooking functions during program analysis. This hook-based approach enables runtime detection of external data entry points without requiring complex static analysis.

Algorithm 1 Function Hooking Strategy

Require: Program binary P
$CFG \leftarrow$ BuildControlFlowGraph(P)
$InputFunctions \leftarrow$ {fgets, scanf, read, gets}
for all function f in CFG **do**
  if $f.name \in InputFunctions$ **then**
    InstallInputHook(f)
  else
    InstallGenericHook(f)
  end if
end for

The system uses two types of hooks: input function hooks that immediately mark data as tainted, and generic hooks that check whether function parameters contain tainted data. This dual approach ensures both taint introduction and propagation are monitored throughout execution.

Input functions receive special treatment because they represent the primary vectors for external data entry. When these functions are called, the system automatically creates tainted symbolic data and registers the associated memory regions as containing potentially malicious content.

4.3 Dynamic Taint Tracking

TraceGuard tracks taint propagation through two complementary mechanisms: symbolic variable naming and memory region mapping. This approach ensures taint information persists across function calls and memory operations.

Algorithm 2 Taint Introduction at Input Functions

Require: Function call to input function f, State s
$data \leftarrow$ CreateSymbolicData(taint_source_ $+ f.name$)
$s.globals$[taint_score] $\leftarrow s.globals$[taint_score] $+ \alpha_{input}$
$s.globals$[tainted_functions]$.add(f.name)$
if f involves memory allocation **then**
    $buffer\_addr \leftarrow$ GetBufferAddress(s)
    $buffer\_size \leftarrow$ GetBufferSize(s)
    $s.globals$[tainted_regions]$.add((buffer\_addr, buffer\_size))$
end if
return $data$

Symbolic variable naming creates a persistent taint identifier that follows data through symbolic operations. Memory region tracking maintains a mapping of tainted buffer addresses and sizes, enabling taint detection when pointers reference previously tainted memory locations.

Algorithm 3 Taint Status Check

Require: State s, Variable or address $target$
if $target$ is symbolic variable **then**
    return taint_source_ $\in target.name$
else if $target$ is memory address **then**
    for all $(addr, size)$ in $s.globals$[tainted_regions] **do**
        if $addr \leq target < addr + size$ **then**
            return True
        end if
    end for
end if
return False

Additionally, memory region tracking maintains a mapping of tainted buffer addresses and sizes, enabling taint detection when pointers reference previously tainted memory locations.

4.4 Path Prioritization

TraceGuard implements a three-tier prioritization system that classifies symbolic execution states based on their calculated taint scores. This classification determines exploration order to focus computational resources on security-relevant paths.

Algorithm 4 State Classification and Prioritization

Require: Active states $\mathcal{S}$, Thresholds τ_{high}, τ_{medium}

$scored\_states \leftarrow []$
for all state $s \in \mathcal{S}$ **do**
  $score \leftarrow \textsc{CalculateTaintScore}(s)$
  $scored\_states.append((score, s))$
end for
$P_{high} \leftarrow \{s : score \geq \tau_{high}\}$
$P_{medium} \leftarrow \{s : \tau_{medium} \leq score < \tau_{high}\}$
$P_{normal} \leftarrow \{s : score < \tau_{medium}\}$
exploration_queue $\leftarrow P_{high} + P_{medium} + P_{normal}$
return first k states from exploration_queue

The score calculation combines multiple factors to assess security relevance. Base scores come from taint interactions tracked by function hooks, with additional bonuses for execution within previously identified tainted functions and penalty reductions for excessive execution depth.

Algorithm 5 Taint Score Calculation

Require: State s

$score \leftarrow \max(s.globals[\text{taint\_score}], \sigma_{min})$
if current function $\in$ tainted functions **then**
  $score \leftarrow score + \beta_{tainted}$
end if
if execution depth $> \delta_{threshold}$ **then**
  $score \leftarrow score \times \gamma_{penalty}$
end if
return $score$

High-priority states typically represent paths directly processing user input or executing within security-critical functions. Medium-priority states show moderate taint relevance, while normal-priority states primarily handle untainted data. The system limits active states to prevent path explosion while maintaining adequate exploration coverage.

4.4.1 Adaptive State Pool Management

A critical component of TraceGuard's practical viability lies in its adaptive state pool management strategy, which prevents path explosion while maintaining exploration effectiveness. The system employs a bounded exploration approach that dynamically adjusts the active state pool based on both computational constraints and taint score distributions.

Bounded Exploration Principle: Rather than allowing unlimited state proliferation, TraceGuard maintains a fixed upper bound k on concurrent active states. This constraint transforms the potentially infinite symbolic execution search space into a manageable, resource-bounded exploration process. The bound k represents a balance between exploration thoroughness and computational tractability, typically set to a small constant based on empirical analysis of memory usage and solver performance.

Dynamic State Replacement: When the exploration encounters new states that would exceed the bound k, the system employs a replacement strategy based on taint scores. New states are only admitted to the active pool if their taint scores exceed those of current low-priority states. This ensures that computational resources remain focused on the most security-relevant execution paths, even as the program exploration discovers new branches.

Priority-Based Pruning: The state pruning mechanism operates according to the established three-tier priority system. When resource limits are reached, normal-priority states are pruned first, followed by medium-priority states if necessary. High-priority states are preserved except in extreme cases where all active states achieve high-priority classification, at which point fine-grained score comparisons determine pruning order.

This adaptive approach ensures that TraceGuard maintains bounded computational requirements while maximizing the security relevance of explored paths, addressing both the theoretical challenge of path explosion and the practical constraints of finite computational resources.

4.5 Exploration Depth Control and Vulnerability Probability

TraceGuard prevents path explosion through multiple complementary techniques that limit exploration depth while maintaining sufficient coverage for vulnerability discovery. A fundamental principle underlying this approach is the inverse relationship between execution depth and vulnerability probability.

The preference for shorter paths in vulnerability discovery is grounded in both theoretical security principles and empirical evidence from vulnerability research [6]. Security vulnerabilities typically manifest near the boundary between external input and internal program logic, where insufficient validation or sanitization allows malicious data to corrupt program state. As execution depth increases beyond these initial input processing stages, several factors reduce vulnerability probability: (1) input data has undergone additional validation and transformation steps, (2) the program state becomes more complex and harder for attackers to predict and control, and (3) deeper code paths typically receive more thorough testing during development.

Research on real-world vulnerability databases demonstrates that critical security flaws such as buffer overflows and injection attacks are statistically more likely to occur in shallow call stacks near input sources than in deeply nested program logic. This observation aligns with attack surface theory, which suggests that the most accessible vulnerabilities are those that can be triggered with minimal program state setup, making them both more discoverable by automated tools and more attractive to attackers.

Algorithm 6 Progressive Depth Penalties

Require: State s with execution depth d

if $d > \delta_{high}$ **then**
    $s.score \leftarrow s.score \times \gamma_{high}$
else if $d > \delta_{medium}$ **then**
    $s.score \leftarrow s.score \times \gamma_{medium}$
end if

The depth penalty system gradually reduces state scores as execution depth increases, naturally prioritizing shorter paths that are more likely to trigger vulnerabilities quickly. This graduated approach avoids abrupt path termination while steering exploration toward more promising regions of the program space. The system employs configurable depth thresholds (δ_{high}, δ_{medium}) and penalty factors (γ_{high}, γ_{medium}) to balance thorough exploration with computational efficiency.

Beyond depth penalties, TraceGuard coordinates multiple exploration control mechanisms to manage path explosion effectively. These include execution length limitations to prevent infinite loops, cycle detection to avoid repetitive exploration patterns, and adaptive state management that maintains an optimal number of active states based on available computational resources.

5 Implementation

5.1 System Architecture

5.1.1 TraceGuard Framework Overview

5.1.2 Core Components

- **TaintAnalyzer:** Tracks taint propagation through symbolic states
- **ScoreCalculator:** Computes dynamic taint scores
- **ExplorationStrategy:** Implements taint-guided path prioritization
- **HookManager:** Manages function call interception and monitoring

5.2 Angr Integration

5.2.1 Custom Exploration Technique

```python
class TaintGuidedExploration(angr.exploration_techniques.ExplorationTechnique):
    def __init__(self, taint_sources, max_depth=50):
        super().__init__()
        self.taint_sources = taint_sources
        self.max_depth = max_depth
        self.taint_tracker = TaintTracker()

    def step(self, simgr, stash='active', **kwargs):
        # Implementation details
```

5.2.2 State Prioritization Implementation

```python
def prioritize_states(self, states):
    scored_states = []
    for state in states:
        score = self.calculate_taint_score(state)
```

```
        scored_states.append((score, state))

    # Sort by score (descending) and return prioritized list
    return [state for _, state in sorted(scored_states, reverse=True)]
```

5.3 Taint Tracking Implementation

5.3.1 Taint Propagation Engine

5.3.2 Function Hook System

5.4 Configuration and Extensibility

5.4.1 Configuration Parameters

```
{
    "taint_sources": ["fgets", "scanf", "getchar", "read"],
    "memory_functions": ["malloc", "calloc", "realloc"],
    "max_depth": 50,
    "score_weights": {
        "input_bonus": 2.0,
        "memory_bonus": 1.5,
        "depth_penalty": 0.1
    }
}
```

5.4.2 Plugin Architecture

6
Evaluation

6.1 Experimental Design

6.1.1 Research Questions

1. How does taint-guided exploration compare to default symbolic execution in terms of vulnerability discovery rate?

2. What is the computational overhead of taint tracking and scoring?

3. How does the approach scale with program complexity?

4. What is the effectiveness of different taint source configurations?

6.1.2 Evaluation Metrics

- **Coverage Metrics:** Basic block coverage, path coverage

- **Efficiency Metrics:** Time to first vulnerability, total analysis time

- **Effectiveness Metrics:** Number of vulnerabilities found, false positive rate

- **Scalability Metrics:** Memory usage, state explosion control

6.2 Benchmark Programs

6.2.1 Synthetic Benchmarks

6.2.2 Real-World Programs

6.3 Experimental Results

6.3.1 Comparison with Standard Symbolic Execution

6.3.2 Ablation Studies

6.4 Case Studies

6.4.1 Buffer Overflow Discovery

6.4.2 Format String Vulnerability

7 Conclusion

This thesis introduced a novel approach to optimizing symbolic execution through the integration of taint analysis and path prioritization. The primary goal was to enhance the efficiency and effectiveness of symbolic execution in discovering security vulnerabilities by focusing computational resources on security-relevant program paths.

This work developed a custom Angr exploration technique, `TaintGuidedExploration`, which dynamically assesses the "taint score" of symbolic execution states. This score is calculated based on the interaction of program paths with tainted data originating from user inputs and memory allocations. By prioritizing states with higher taint scores, the tool effectively navigates the vast execution space, directing the symbolic execution engine towards areas most likely to harbor vulnerabilities.

The practical implementation leveraged the Angr framework, incorporating custom hooks for input functions and general function calls to track taint propagation accurately. This work demonstrated how the system identifies tainted functions, tracks taint flow through call edges, and uses these insights to adaptively adjust path priorities.

While a formal benchmark with hard data across a wide range of complex binaries was beyond the scope of this thesis, preliminary analysis and conceptual validation indicate that this approach can significantly refine the search space. The methodology provides a systematic and automated way to identify and prioritize security-critical paths, moving beyond manual intuition or uniform exploration. The evaluation section outlines how future work could rigorously compare performance metrics like execution time, path coverage quality, and vulnerability discovery rates against default symbolic execution strategies.

In essence, this work presents a foundational step towards making symbolic execution more practical and efficient for real-world software security analysis. By intelligently guiding the exploration process with taint information, the proposed approach offers a promising direction for more effective and scalable vulnerability discovery.

8
Future Work

Some ideas for future work could be: - Change meta file to actual header file - Make it work also for ARM and X86 (checking stack and heap arguments) - Check that it works also for libraries (not only for main function) - Let the script analyze a complex program (multiple files) and get an output over all (now it only works for one file at a time)

9 Usage of AI

For the development of this thesis, AI-assisted technologies, specifically large language models, were utilized to enhance various aspects of the writing and research process.

- **Text Transformation and Fluency:** AI tools were primarily used to refine and transform sections of the text to improve fluency, clarity, and highlight important aspects without altering the original content or technical accuracy. This included rephrasing sentences, improving sentence structure, and ensuring a consistent academic tone.
- **Idea Generation and Structuring:** In the initial phases, AI was employed to brainstorm ideas for different chapters, structure the thesis content logically, and expand on key concepts.
- **Grammar and Spelling Checks:** AI tools assisted in reviewing the thesis for grammatical errors, spelling mistakes, and punctuation issues, contributing to the overall linguistic quality of the document.
- **Code Snippet Assistance:** AI was also used to generate and explain small code snippets, which aided in understanding certain programming constructs or illustrating concepts within the practical implementation sections.

It is important to note that while AI provided significant assistance, the core research, conceptual design, implementation, and analytical interpretation remained the sole responsibility of the author. All information presented in this thesis, including any text passages or code generated with AI assistance, has been thoroughly reviewed, verified, and integrated by the author to ensure accuracy, originality, and adherence to academic standards.

Bibliography

[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094. ACM, 2014.

[2] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204. ACM, 2012.

[3] Jiang Ming, Dinghao Wu, Jun Wang, Xinyu Xing, and Zhiqiang Liu. TaintPipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium*, pages 65–80. USENIX Association, 2015.

[4] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1623–1628. ACM, 2016.

[5] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium*, pages 181–198. USENIX Association, 2020.

[6] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[7] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, pages 138–153. IEEE, 2016.

[8] Jake Springer and Siji Feng. Teaching with angr: A symbolic execution curriculum and CTF. In *2018 IEEE/ACM 1st International Workshop on Automated Software Engineering Education*, pages 13–20. IEEE, 2018.

[9] Shuangjie Yao and Junjie Chen. Empc: Effective path prioritization for symbolic execution with path cover. *arXiv preprint arXiv:2505.03555*, 2025. Available at: https://arxiv.org/abs/2505.03555.

A
Appendix

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assessor: ______________________________

Name Student: ______________________________

Matriculation No.: ______________________________

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used.
This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: ____________________ Student: ____________________

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ______________________________

Place, Date: ____________________ Student: ____________________

Place, Date: ____________________ Assessor: ____________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.