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

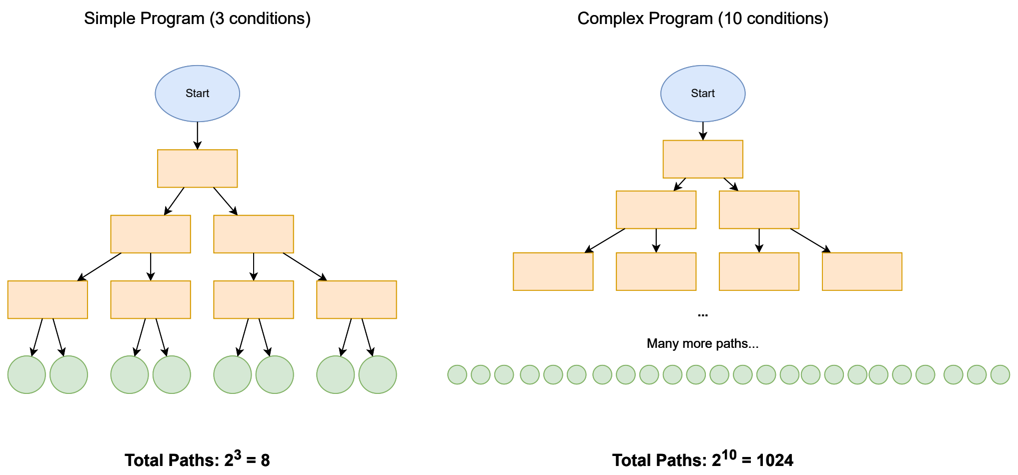
Simple Program (3 conditions)
Start
Total Paths: $2^3 = 8$
Complex Program (10 conditions)
Start
...
Many more paths...
Total Paths: $2^{10} = 1024$

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 implications of this research.

- **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, taint analysis, control flow analysis, and the Angr 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 [8].

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.

The scalability challenges have motivated optimization approaches including state merging [4], veritesting [1], and compilation-based symbolic execution [7]. These optimization techniques form the foundation for more targeted analysis approaches that focus computational resources on security-relevant program behaviors.

2.2 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.

The combination of taint analysis and symbolic execution creates a powerful analysis framework. Symbolic execution can explore multiple program paths while taint analysis identifies which paths involve security-relevant data flows. Research has demonstrated effective integration through pipelined approaches that achieve significant performance improvements while maintaining precision [5]. This integration enables targeted exploration of paths that process untrusted input, significantly improving the efficiency of vulnerability discovery.

2.3 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.

2.4 Angr Framework

Angr is an open-source binary analysis platform providing comprehensive capabilities for static and dynamic program analysis [9]. The platform supports multiple architectures and provides a Python-based interface for research and education [10]. 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. Multiple symbolic execution frameworks address different domains and applications. KLEE [2] focuses on automatic test generation for C programs, SAGE [**?**] specializes in binary whitebox fuzzing for vulnerability discovery, CBMC [3] offers bounded model checking for C/C++ programs with formal verification capabilities, and Java PathFinder [11] provides symbolic execution for Java bytecode with emphasis on model checking applications. Angr's comprehensive binary analysis capabilities and extensible architecture make it particularly suitable for implementing the taint-guided exploration strategies presented in this thesis.

3 Related Work

This chapter examines specific approaches to improving symbolic execution effectiveness for vulnerability discovery, focusing on methodological innovations that address different aspects of the scalability challenge.

3.1 Compositional Analysis: MACKE

MACKE [6] introduces a compositional approach that performs symbolic execution on individual program functions in isolation, then combines results using static analysis and inter-procedural path feasibility checking. The tool employs a three-step process: first analyzing functions separately to achieve higher coverage, then using compositional reasoning to reduce false positives, and finally assigning severity scores to reported vulnerabilities. MACKE demonstrated superior coverage compared to forward symbolic execution on real-world programs including Flex, Grep, and Bzip2, while providing significantly fewer false positives than pure static analysis approaches.

3.2 Compilation-Based Execution: SymCC

SymCC [7] revolutionized symbolic execution performance through a compilation-based approach that embeds symbolic execution directly into program binaries during compilation. Rather than interpreting intermediate representations at runtime, SymCC inserts symbolic computation code alongside concrete execution, achieving significant performance improvements with 10-12 × speedup over traditional approaches like KLEE and QSYM. This optimization enabled higher coverage analysis and led to the discovery of vulnerabilities in heavily tested projects such as OpenJPEG.

4 Taint-Guided Exploration

Having established the need for more efficient symbolic execution in Chapter 1, we now turn to the theoretical foundation of TraceGuard's solution. 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. The following sections present the conceptual algorithms and design decisions that enable this taint-guided exploration strategy.

4.1 Theoretical Foundation

Traditional symbolic execution engines suffer from the path explosion problem, where the number of possible execution paths grows exponentially with program complexity. TraceGuard addresses this fundamental challenge by introducing a taint-guided exploration strategy that prioritizes paths based on their interaction with user-controlled data.

The central concept revolves around calculating a dynamic ”taint score” for each symbolic execution state. This score quantifies how closely a given execution path interacts with tainted data originating from external inputs. By prioritizing states with higher taint scores, the symbolic execution engine directs its computational resources toward program regions that are most likely to contain security vulnerabilities.

The taint score serves as a heuristic measure of security relevance. States that process user inputs, manipulate tainted data, or reach security-critical functions receive higher scores, while states that operate on untainted data or perform auxiliary computations receive lower scores. This approach fundamentally transforms symbolic execution from an exhaustive search into a guided exploration strategy.

4.2 Taint Source Identification

The foundation of TraceGuard's approach lies in accurately identifying taint sources within the analyzed program. Taint sources represent points where external, potentially malicious data enters the program execution flow. These sources serve as the starting points for taint propagation analysis.

Algorithm 1 Taint Source Identification

```
Require: Program control flow graph $CFG$, Function set $F$
Ensure: Set of identified taint sources $T$
$T \leftarrow \emptyset$
$InputFunctions \leftarrow$ {fgets, scanf, getchar, read, recv, ...}
$MemoryFunctions \leftarrow$ {malloc, calloc, realloc, ...}
for all function $f \in F$ do
    if $f.name \in InputFunctions$ then
        $T \leftarrow T \cup \{f\}$
        MarkAsTaintSource($f$, INPUT)
    else if $f.name \in MemoryFunctions$ then
        $T \leftarrow T \cup \{f\}$
        MarkAsTaintSource($f$, MEMORY)
    end if
end for
for all system call $syscall$ in $CFG$ do
    if IsIOSystemCall($syscall$) then
        $T \leftarrow T \cup \{syscall\}$
        MarkAsTaintSource($syscall$, SYSCALL)
    end if
end for
return $T$
```

TraceGuard identifies taint sources through static analysis of the program's control flow graph and function call relationships. The system recognizes several categories of taint sources:

Input Functions: Functions that read data from external sources, including standard input/output operations, file operations, and network communications. These functions represent direct pathways for attacker-controlled data to enter the program.

Memory Allocation Functions: Dynamic memory allocation operations are treated as potential taint sources because they create memory regions that may subsequently store user-controlled data. While not inherently tainted, these allocations become relevant when combined with input operations.

System Call Interfaces: Low-level system calls that interact with the operating system kernel, particularly those involved in inter-process communication, file system operations, and network communications.

4.3 Taint Propagation and Tracking

Once taint sources are identified and initial data is marked as tainted, TraceGuard employs a sophisticated taint propagation mechanism to track how tainted data flows through the program's execution. This process involves monitoring data transfers, function calls, and memory operations to maintain accurate taint information.

Algorithm 2 Taint Propagation Analysis

```
Require: Symbolic execution state $s$, Operation $op$, Operands $\{op_1, op_2, ..., op_n\}$
Ensure: Updated taint information
tainted ← FALSE
for all operand $op_i$ in operands do
    if IsTainted($s, op_i$) then
        tainted ← TRUE
        break
    end if
end for
if tainted then
    result ← ExecuteOperation($op, operands$)
    MarkAsTainted($s, result$)
    UpdateTaintScore($s$, TAINT_INTERACTION)
else
    result ← ExecuteOperation($op, operands$)
end if
return result
```

Algorithm 3 Function Call Taint Tracking

```
Require: Function call $f$, Parameters $\{p_1, p_2, ..., p_k\}$, State $s$
Ensure: Updated function taint status and score
hasTaintedParams ← FALSE
taintedParamCount ← 0
for all parameter $p_i$ in parameters do
    if IsTainted($s, p_i$) then
        hasTaintedParams ← TRUE
        taintedParamCount ← taintedParamCount + 1
    end if
end for
if hasTaintedParams then
    MarkFunctionAsTainted($f$)
    if $f \in InputFunctions$ then
        UpdateTaintScore($s$, INPUT_FUNCTION_BONUS)
    else
        UpdateTaintScore($s$, TAINTED_CALL_BONUS)
    end if
    PropagateToReturnValue($s, f$)
else
    UpdateTaintScore($s$, FUNCTION_CALL_PROGRESS)
end if
```

Data Flow Tracking: TraceGuard tracks taint propagation through register transfers, memory operations, and arithmetic computations. When tainted data participates in an operation, the result inherits taint status according to predefined propagation rules.

Function Call Propagation: Taint information propagates across function boundaries through parameter passing and return value mechanisms. When a function receives tainted parameters, the function itself becomes associated with taint processing.

Taint Score Calculation: The taint score for each symbolic execution state is calculated based on multiple factors including direct taint interaction, tainted function execution, input function calls, and exploration progress. The score employs a decay mechanism to prevent infinite accumulation and maintain relative priorities.

4.4 Adaptive Path Prioritization Algorithm

TraceGuard's path prioritization algorithm represents the core innovation of the taint-guided approach. This algorithm dynamically reorders the symbolic execution exploration queue based on calculated taint scores, ensuring that the most promising paths receive priority attention.

Algorithm 4 Taint-Guided Path Prioritization

```
Require: Set of symbolic execution states $S$
Ensure: Prioritized list of states $P$
for all state $s \in S$ do
    $score[s] \leftarrow \textsc{CalculateTaintScore}(s)$
    $\textsc{UpdateTaintHistory}(s, score[s])$
end for
$P_{high} \leftarrow \{s \in S : score[s] \geq 6.0\}$
$P_{medium} \leftarrow \{s \in S : 2.0 \leq score[s] < 6.0\}$
$P_{normal} \leftarrow \{s \in S : score[s] < 2.0\}$
$P \leftarrow \textsc{Sort}(P_{high}) \cup \textsc{Sort}(P_{medium}) \cup \textsc{Sort}(P_{normal})$
return $P[1 : \textsc{MaxActiveStates}]$
```

The algorithm classifies states into three priority categories based on their taint scores. High-priority states (score $\geq$ 6.0) represent execution paths with strong taint interactions, medium-priority states (2.0 $\leq$ score $<$ 6.0) show moderate taint relevance, and normal-priority states (score $<$ 2.0) have minimal taint interaction.

State Classification Logic: The priority thresholds are carefully chosen based on empirical analysis of taint score distributions. High-priority states typically correspond to paths that directly process user input or execute within security-critical functions. Medium-priority states often represent paths that indirectly interact with tainted data or explore new program regions. Normal-priority states generally handle untainted data or perform auxiliary computations.

Dynamic Reordering: At each symbolic execution step, the algorithm recalculates taint scores and reorders the exploration queue. This dynamic approach ensures that the prioritization adapts to changing taint conditions as execution progresses. States that gain taint relevance are promoted, while states that lose taint interaction are demoted.

State Management: To prevent resource exhaustion, the algorithm limits the number of active states to a configurable maximum (typically 15-20 states). Excess states are moved to secondary queues based on their priority levels, allowing for potential reactivation if computational resources become available.

4.5 Exploration Depth Management

A critical design decision in TraceGuard involves implementing depth limitations to prevent the symbolic execution from pursuing paths that are unlikely to yield valuable security insights. This design choice reflects the fundamental observation that vulnerability discovery efficiency decreases significantly as exploration depth increases.

Algorithm 5 Exploration Depth Control Strategy

```
Require: State $s$, Maximum depth $D_{max}$, Warning threshold $D_{warn}$
Ensure: Exploration decision and depth penalty
$current\_depth \leftarrow$ GetExecutionDepth($s$)
$depth\_penalty \leftarrow 0$
if $current\_depth > D_{max}$ then
    TerminatePath($s$)
    return TERMINATE
else if $current\_depth > D_{warn}$ then
    $depth\_penalty \leftarrow$ CalculateDepthPenalty($current\_depth, D_{warn}, D_{max}$)
    ApplyScorePenalty($s, depth\_penalty$)
    return CONTINUE_WITH_PENALTY
else
    return CONTINUE_NORMAL
end if
```

Algorithm 6 Dynamic Depth Penalty Calculation

```
Require: Current depth $d$, Warning threshold $D_{warn}$, Maximum depth $D_{max}$
Ensure: Depth penalty factor
$depth\_ratio \leftarrow \frac{d - D_{warn}}{D_{max} - D_{warn}}$
$penalty\_factor \leftarrow 0.5 + 0.5 \times depth\_ratio^2$
return $penalty\_factor$
```

Theoretical Justification: The rationale for depth limitation stems from the security principle that most vulnerabilities occur within a limited distance from user input sources. As execution paths diverge further from input processing code, the likelihood of discovering security-relevant behavior diminishes exponentially, while computational cost grows substantially.

Resource Optimization: By preventing excessive depth exploration, TraceGuard concentrates computational resources on paths that are more likely to contain vulnerabilities. This approach significantly improves analysis efficiency compared to exhaustive exploration strategies.

Vulnerability Coverage: Empirical evidence suggests that analyzing the first N levels of execution depth captures the majority of security-relevant program behavior. Deep execution paths often involve complex program logic that is less likely to contain direct security vulnerabilities.

Graduated Depth Control: The depth control algorithm implements a graduated approach where paths approaching the depth limit receive score penalties that reduce their priority, while paths exceeding the limit are terminated. This ensures smooth transitions and prevents abrupt exploration termination.

Algorithm 7 Depth-Limited Exploration Control

```
Require: State $s$, Maximum depth $D$
Ensure: Continue exploration decision
current_depth ← GetExecutionDepth($s$)
if current_depth $> D$ then
    return TerminatePath
else if current_depth $> 0.8 \times D$ then
    ApplyDepthPenalty($s$)
    return ContinueWithPenalty
else
    return ContinueNormal
end if
```

The depth control algorithm implements a graduated approach to exploration limitation. Paths approaching the depth limit receive score penalties that reduce their priority, while paths exceeding the limit are terminated. This graduated approach ensures smooth transitions and prevents abrupt exploration termination.

4.6 Symbolic Execution Integration

TraceGuard's taint-guided exploration operates as an enhancement to existing symbolic execution engines rather than as a standalone system. The theoretical integration approach focuses on extending the standard symbolic execution workflow through strategic intervention points that preserve the underlying analysis semantics while adding security-focused guidance.

Exploration Technique Extension: The taint-guided approach functions as a specialized exploration strategy that can be integrated into symbolic execution engines that support pluggable exploration techniques. This design allows the system to leverage existing constraint solving and symbolic reasoning capabilities while directing the search toward security-relevant paths.

State Management Integration: The integration strategy involves intercepting the symbolic execution state management process to inject taint scoring and prioritization logic. At each exploration step, the system evaluates taint scores for active states and reorders the exploration queue before the symbolic execution engine processes the next batch of states.

Hook-Based Monitoring: Taint propagation tracking integrates through a hooking mechanism that monitors function calls and data operations without modifying the core symbolic execution semantics. This approach allows the system to observe program behavior and maintain taint information while preserving the accuracy of the underlying symbolic analysis.

Complementary Optimization Techniques: The taint-guided strategy operates alongside other symbolic execution optimizations such as depth limiting, loop detection, and search heuristics. This cooperative approach ensures that multiple optimization strategies can work together to improve overall analysis efficiency.

The theoretical integration model demonstrates that security-focused guidance can be added to symbolic execution engines without fundamental architectural changes, making the approach broadly applicable to different symbolic execution frameworks and analysis scenarios.

4.7 Theoretical Foundation

4.7.1 Formal Problem Definition

Let P be a program with control flow graph $CFG = (V, E)$ where V represents basic blocks and E represents control flow transitions. Let $T \subseteq V$ be the set of taint sources and $S \subseteq V$ be the set of security-relevant program locations (sinks).

Definition 1 (Taint-Guided Path): A path $\pi = v_0, v_1, ..., v_n$ in CFG is taint-guided if $\exists i, j$ such that $v_i \in T$ and there exists a data dependency from v_i to v_j where $j > i$.

Definition 2 (Taint Score): For a symbolic execution state s at program location v, the taint score $\tau(s)$ is defined as:

$$\tau(s) = \alpha \cdot I(s) + \beta \cdot M(s) + \gamma \cdot D(s) + \delta \cdot C(s)$$

where $I(s)$ is input interaction score, $M(s)$ is memory operation score, $D(s)$ is depth penalty, and $C(s)$ is coverage bonus.

4.8 Complete Taint Scoring Framework

Algorithm 8 Dynamic Taint Score Calculation

```
Require: State $s$, Taint information $\mathcal{T}$, Execution history $H$
Ensure: Updated taint score $\tau(s)$
$score \leftarrow 0$
$input\_bonus \leftarrow \textsc{CalculateInputInteraction}(s, \mathcal{T})$
$memory\_bonus \leftarrow \textsc{CalculateMemoryOperations}(s, \mathcal{T})$
$call\_bonus \leftarrow \textsc{CalculateFunctionCalls}(s, \mathcal{T})$
$depth\_penalty \leftarrow \textsc{CalculateDepthPenalty}(s)$
$coverage\_bonus \leftarrow \textsc{CalculateCoverageBonus}(s, H)$
$score \leftarrow input\_bonus + memory\_bonus + call\_bonus - depth\_penalty + coverage\_bonus$
$\tau(s) \leftarrow \max(0, score)$                    ▷ Ensure non-negative score
return $\tau(s)$
```

4.9 Complexity Analysis

4.9.1 Time Complexity

The taint-guided exploration has time complexity $O(|S| \cdot \log|S| + |T|)$ per symbolic execution step, where $|S|$ is the number of active states and $|T|$ is the number of taint tracking operations.

4.9.2 Space Complexity

Taint tracking requires additional memory proportional to the number of symbolic variables, adding $O(|V|)$ space overhead where $|V|$ is the set of tracked symbolic values.

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*, pages 1083–1094. ACM, 2014.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[3] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, pages 368–371. ACM, 2003.

[4] Vitaly 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.

[5] 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.

[6] 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.

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

[8] 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). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331. IEEE, 2010. doi: 10.1109/SP.2010.26.

[9] 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.

[10] 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.

[11] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. In *Automated Software Engineering*, pages 3–12. IEEE, 2003.

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.

September 2023