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

**Challenges in Program Analysis.** Symbolic execution is a powerful program analysis technique used for vulnerability discovery, test case generation, and software property verification.

Beyond security analysis, symbolic execution serves as a foundation for various software verification tasks, including bug detection, correctness checking, and automated program reasoning.

However, in large software systems, often lacking a well-defined analysis entry point, finding suitable starting points for symbolic execution is far from straightforward. Programs frequently serve a wide range of use cases, contain numerous execution paths, and can be expansive enough to obscure which code regions are security-critical and drive the core vulnerability landscape.

Moreover, 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. One recent example demonstrating this challenge is the analysis of complex libraries like OpenSSL, where millions of execution paths can emerge from relatively small input variations, making exhaustive analysis computationally prohibitive.

Despite the importance of efficient symbolic execution, practitioners lack an automated method to intelligently prioritize promising execution paths for security analysis. Manually determining where analysis should begin can be tedious and error-prone. The structure of even medium-sized programs can be extremely cluttered and overwhelming. Current symbolic execution engines typically employ uniform exploration strategies, treating all program paths with equal priority regardless of their potential security relevance, as Figure 1.1 illustrates with a simplified program control flow.

Addressing this issue could significantly enhance both the efficiency and thoroughness of symbolic execution for security testing.

**Thesis Overview.** This work presents a novel 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 and memory allocation sites,

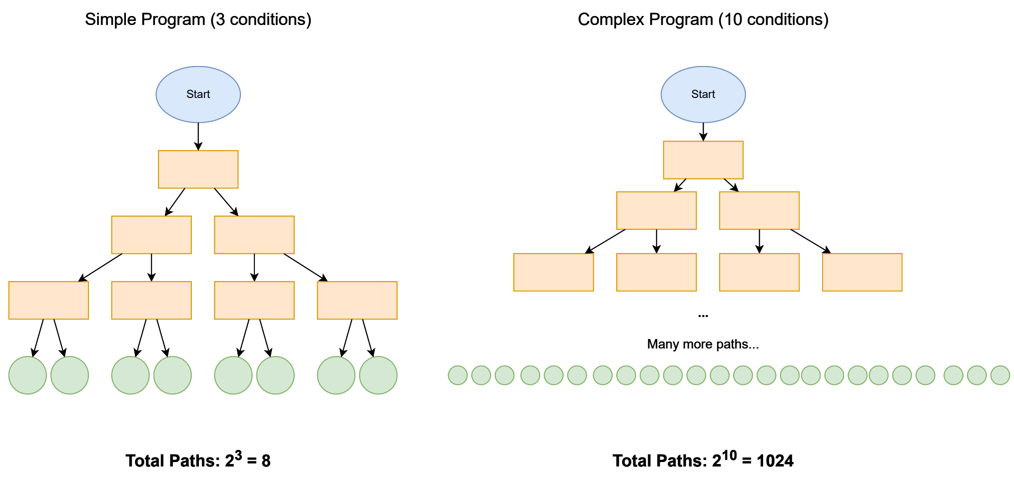


Figure 1.1: Illustration of the path explosion problem: As program complexity increases from 3 to 10 conditions, the number of possible execution paths grows exponentially from 8 to 1024 paths.

guiding the symbolic execution engine to focus computational resources on paths most likely to exhibit security-relevant behaviors. The main contributions of this thesis are:

- **Taint-Guided Path Prioritization:** A novel integration of dynamic taint analysis with symbolic execution that uses taint propagation patterns to intelligently prioritize exploration of security-relevant execution paths.
- **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.
- **Practical Implementation:** A complete implementation of the proposed approach using the angr symbolic execution framework, demonstrating the feasibility and effectiveness of taint-guided exploration in a production-quality tool.
- **Empirical Evaluation:** Comprehensive evaluation comparing the proposed approach against standard symbolic execution techniques, which will measure improvements in analysis efficiency, vulnerability discovery rate, and overall scalability.

The effectiveness of this optimization will be evaluated through extensive experimentation on representative programs, examining key metrics including runtime efficiency, path coverage quality, and vulnerability detection capabilities. Preliminary analysis indicates that the taint-guided approach can significantly reduce analysis time while maintaining or improving the detection of security-relevant program behaviors, making symbolic execution more practical for analyzing large and complex software systems.

The thesis is organized as follows:

- **Chapter 2** reviews essential background concepts including symbolic execution, taint analysis, and the angr framework.

- **Chapter 3** presents the conceptual framework and theoretical algorithms underlying the taint-guided exploration strategy.
- **Chapter 4** details the practical implementation of our approach, including integration with angr and the design of the scoring mechanism.
- **Chapter 5** presents a comprehensive evaluation of the approach, comparing its performance against standard symbolic execution techniques.
- **Chapter 6** discusses related work in symbolic execution optimization and path prioritization.
- **Chapter 7** concludes with a summary of contributions and implications for future research.
- **Chapter 8** explores potential extensions and future research directions.

# 2
# Background

This chapter establishes the theoretical foundations necessary for understanding the optimization techniques presented in this thesis. We examine symbolic execution, taint analysis, program analysis techniques, and the Angr framework.

## 2.1 Symbolic Execution

Symbolic execution is a static analysis technique that explores program 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 [7].

Various techniques address the scalability challenge, including state merging [4], constraint optimization, and compositional analysis methods. Recent advances include veritesting approaches that combine dynamic and static symbolic execution [2] and compilation-based symbolic execution achieving orders of magnitude performance improvements [6]. Path prioritization strategies represent another important direction, with modern approaches including coverage-guided exploration, target-directed search, and machine learning-based path selection [3][1].

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.

[1] https://github.com/ksluckow/awesome-symbolic-execution

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.

## 2.2 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].

## 2.3 Program Analysis Techniques

Static analysis inspects program code without executing it, analyzing structure, data flow, and control flow based solely on the source code or binary representation. This approach offers comprehensive coverage and efficiency, enabling examination of all 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 potential issues are actually exploitable under realistic conditions. However, dynamic analysis results depend heavily on input quality and coverage, and achieving exhaustive coverage can

[2] https://github.com/badnack/angr_taint_engine

require exponential time for complex programs.

Modern tools often combine both approaches. Symbolic execution represents a hybrid technique, combining static analysis of program structure with dynamic-style exploration of execution paths. Hybrid approaches have proven particularly effective, such as combining fuzzing with selective symbolic execution for comprehensive vulnerability discovery [10].

A Call Graph (CG) 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[3].

## 2.4 Angr Framework

Angr is an open-source binary analysis platform providing comprehensive capabilities for static and dynamic program analysis [8]. The platform supports multiple architectures and provides a Python-based interface for research and education [9]. Key components include[4]: 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 [1]. Static analysis provides efficiency but may miss indirect calls, while dynamic analysis offers completeness at higher computational cost[5]. The static approach analyzes the binary without execution, making it efficient for initial program understanding, while dynamic analysis executes the program with sample inputs to discover reachable code, providing more complete coverage of actual execution paths and better handling of indirect calls.

Angr represents program states with register values, memory contents, path constraints, and execution history. The framework provides APIs for state inspection and manipulation, along with fine-grained execution control through step functions and exploration techniques[6]. The step function advances execution by single instructions, enabling precise control over exploration processes, while various exploration strategies guide path selection including depth-first search, breadth-first search, and custom heuristics. Configuration options control execution behavior, such as handling of unconstrained memory and registers, allowing researchers to customize analysis behavior for specific research requirements.

The next chapter will present the conceptual framework for integrating these approaches into TraceGuard's optimized symbolic execution strategy.

[3] https://docs.angr.io/en/latest/built-in-analyses/#call-graph
[4] https://docs.angr.io/en/latest/core-concepts/
[5] https://docs.angr.io/en/latest/built-in-analyses/
[6] https://docs.angr.io/en/latest/core-concepts/states/

# 3 Improving Symbolic Execution Through Taint Analysis

This is a short conclusion on the thesis template documentation. If you have any comments or suggestions for improving the template, if you find any bugs or problems, please contact me.

How does it work conceptually? (not implementation) which path i choose and why. limiter -¿ je weiter unter desto schwieriger dass ich eine vulnerability triggere Pseudo code of the algorithm

# 4
# Practical Implementation

This is a short conclusion on the thesis template documentation. If you have any comments or suggestions for improving the template, if you find any bugs or problems, please contact me.

How does the script work? Implementation details, how to use it, how to run it, how to set up the environment. Strategies that I use, LoopSeer...

# 5 Evaluation

Compare my work to default angr strategy. Implement a benchmark tool to compare the performance of the taint-guided symbolic execution against the default path exploration strategy in angr. The benchmark should include:

- A set of test programs with known vulnerabilities.
- Metrics for evaluation:
  - Execution time
  - Number of paths explored
  - Vulnerabilities detected
  - Path coverage
- Comparison of results between the taint-guided approach and the default angr strategy.

# 6 Related Work

Evtl. already mentioned in background chapter

Here other approaches to symbolic execution, like the Paper: "MACKE: Compositional Analysis of Low-Level Vulnerabilities with Symbolic Execution"

# 7 Conclusion

This thesis introduced a novel approach to optimizing symbolic execution through the integration of taint analysis and path prioritization. Our 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.

We 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, our 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. We demonstrated how our 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, our 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] angr CFG analysis, 2025. URL https://github.com/angr/angr/tree/master/angr/analyses/cfg.

[2] 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.

[3] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1426–1440. ACM, 2021.

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

[7] 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.

[8] 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*. IEEE, 2016.

[9] 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.

[10] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller:

Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, NDSS 2016. Internet Society, 2016.

# A

# Appendix

**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.*