Input Grammars in Practice

Methods for Symbolic Mining, Systematic Evaluation, and Grey-Box Testing

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Leon Vincent Bettscheider

Saarbrücken, 2025

Day of Colloquium	April 24, 2026
Dean of the Faculty	Prof. Dr. Jan Reineke
Chair of the Committee	Prof. Dr. Jan Reineke
Reporters	
First Reviewer	Prof. Dr. Andreas Zeller
Second Reviewer	Prof. Cristian Cadar, Ph.D.
Third Reviewer	Prof. Dr. Sebastian Hack
Academic Assistant	Alexi Turcotte, Ph.D.

Abstract

Automatically inferring input grammars from programs provides significant benefits for test input generation, documentation, and reverse engineering. Existing approaches depend on sample inputs, which are often incomplete or unavailable. This dissertation introduces a novel sample-free approach that infers input grammars using symbolic execution. Tailored to recursive descent parsers, our approach produces highly accurate grammars.

Research on grammar inference is typically evaluated by comparing the languages of the inferred grammar and a ground-truth grammar. We observe that the commonly used metric is biased, as it relies on a fixed sample size and favors short inputs. To address this limitation, we propose a new grammar accuracy metric that systematically explores the grammar space. Using our metric to re-evaluate input grammars inferred by five prior studies, we find that both precision and recall can decrease significantly.

Recent advances in language-based test generation enable the creation of inputs that satisfy both grammatical structure and semantic constraints. However, these constraints are restricted to the input domain. We integrate execution feedback into the input generation process of the Fandango tool, allowing testers to additionally express constraints over the execution domain. Through grey-box fuzzing experiments, we demonstrate that our unified approach achieves better results compared to input or execution constraints alone.

Zusammenfassung

Die automatische Ableitung von Eingabegrammatiken aus Programmen bietet erhebliche Vorteile für die Generierung von Testeingaben, die Dokumentation und das Reverse Engineering. Bestehende Verfahren erfordern Beispieleingaben, die oft unvollständig oder nicht verfügbar sind. Diese Dissertation stellt einen neuen Ansatz vor, der symbolische Ausführung verwendet und keine Beispieleingaben benötigt. Unser Ansatz wurde für rekursive Abstiegsparser entwickelt und erzeugt hochgradig akkurate Grammatiken.

Forschungsarbeiten im Bereich der Grammatikinferenz werden in der Regel evaluiert, indem die Sprachen der abgeleiteten Grammatik und einer idealen Grammatik verglichen werden. Wir stellen fest, dass die gängige Metrik verzerrt ist, da sie eine feste Stichprobengröße verwendet und kurze Eingaben bevorzugt. Um das zu verbessern, schlagen wir eine neue Metrik vor, die den Grammatikraum systematisch abtastet. Unter Verwendung dieser neuen Metrik evaluieren wir die in fünf bestehenden Studien abgeleiteten Eingabegrammatiken erneut und stellen fest, dass Präzision und Recall erheblich sinken können.

Aktuelle Fortschritte in der sprachbasierten Testgenerierung ermöglichen die Erzeugung von Eingaben, die sowohl syntaktisch gültig sind als auch bestimmte semantische Einschränkungen erfüllen. Allerdings können diese Einschränkungen bislang nur über die Eingabedomäne ausgedrückt werden. Wir erweitern dieses Konzept mit Ausführungsinformationen, die wir in den Eingabegenerierungsprozess des Fandango-Werkzeugs integrieren, und ermöglichen es Testern so, zusätzlich Einschränkungen über die Ausführungsdomäne spezifizieren zu können. Anhand einer Reihe von Grey-Box-Fuzzing-Experimenten zeigen wir, dass unser einheitlicher Ansatz bessere Ergebnisse erzielt als die alleinige Berücksichtigung von Eingabe- oder Ausführungseinschränkungen.

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor, Andreas Zeller, for his invaluable guidance throughout this journey. Andreas fostered an ideal research environment characterized by freedom, trust, and enthusiasm that allowed me to grow both professionally and personally. This work would not have been possible without his support.

I am sincerely grateful to Cristian Cadar and Sebastian Hack for serving as examiners, to Jan Reineke for chairing my committee, and to Alexi Turcotte for serving as academic assistant.

I would also like to thank my colleagues for the valuable discussions, constructive feedback, and the many enjoyable moments we shared.

My special thanks go to my parents and family for their unwavering support and confidence in me.

Last but not least, I am especially grateful to Lisa, for her constant encouragement, understanding, and for always being by my side.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, November 27, 2025

gez. / signed

Leon Vincent Bettscheider

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Research Problem . . . 2
1.2 Contributions . . . 3
1.3 Thesis Outline . . . 4
1.4 Publications . . . 5

2 Background **7**
2.1 Input Syntax Specifications . . . 7
2.1.1 Context-Free Grammars . . . 9
2.1.1.1 Generating Inputs . . . 11
2.1.1.2 Parsing Inputs . . . 13
2.1.1.3 Undecidable Properties . . . 13
2.1.1.4 Inferring Grammars . . . 14
2.1.2 Enriching Syntax with Semantics . . . 14
2.2 Automated Software Testing . . . 15
2.2.1 Terminology . . . 16
2.2.2 Black-Box Testing . . . 18
2.2.3 White-Box Testing . . . 19
2.2.3.1 Program Structure Representations . . . 20
2.2.3.2 Fuzz Testing . . . 23
2.2.3.3 Symbolic Execution . . . 25

3 Symbolic Input Grammar Mining **35**
3.1 Introduction . . . 36
3.2 Symbolically Exploring Parsers and Generating Execution Traces . . . 40
3.2.1 Assumptions . . . 40
3.2.2 Tracking Input Consumption . . . 41
3.2.2.1 The Mimid Approach . . . 42
3.2.2.2 The STALAGMITE Approach . . . 42
3.2.3 Limiting Loops . . . 43
3.2.4 Limiting Recursion . . . 45
3.2.5 Handling Parsers With a Lexing Stage . . . 45
3.3 Inferring the Input Grammar From Execution Traces . . . 46
3.3.1 Converting Traces to an Input Grammar . . . 47
3.3.2 Generalizing Tokens in the Input Grammar . . . 50
3.3.3 Simplification . . . 53

3.3.4 Reducing Overapproximation . . . 53
3.4 Implementation . . . 54
3.5 Evaluation . . . 55
3.5.1 RQ1: Grammar Accuracy . . . 56
3.5.2 RQ2: Grammar Conciseness . . . 60
3.5.3 RQ3: Scaffolding Effort . . . 61
3.5.4 RQ4: Time Consumption . . . 62
3.5.5 RQ5: Bugs Found . . . 62
3.6 Related Work . . . 64
3.6.1 Language Inference . . . 64
3.6.2 Black-Box Grammar Inference . . . 64
3.6.3 White-Box Grammar Inference . . . 65
3.6.4 Static Grammar Inference . . . 65
3.6.5 Generating Valid Inputs for Parser-Driven Programs . . . 66
3.7 Limitations and Future Work . . . 66
3.8 Conclusion . . . 67
3.9 Data Availability . . . 67

4 Input Grammar Evaluation 69
4.1 Introduction . . . 70
4.2 Background . . . 72
4.2.1 Input Grammar Mining . . . 72
4.2.2 On the Difficulty of Comparing Grammars . . . 72
4.2.3 k-Paths . . . 75
4.3 Measuring Grammar Accuracy . . . 76
4.3.1 Generating Inputs . . . 77
4.3.2 Computing Grammar Accuracy . . . 78
4.4 Implementation . . . 79
4.5 Evaluation . . . 79
4.5.1 Compared Approaches . . . 80
4.5.2 RQ1: Revisiting Grammar Accuracy . . . 85
4.5.3 RQ2: Golden Grammar Coverage . . . 87
4.5.4 RQ3: Qualitative Analysis of Black-Box Approaches . . . 89
4.5.5 RQ4: Growth of k-Paths . . . 91
4.5.6 RQ5: Added Value of k > 1 . . . 93
4.6 Related Work . . . 93
4.7 Threats to Validity . . . 94
4.8 Conclusion . . . 95
4.9 Data Availability . . . 95

5 Input Constraints and Execution Goals 97
5.1 Introduction . . . 98
5.2 Background . . . 101
5.2.1 Language-Based Software Testing . . . 101

5.2.2 Combining Input Specifications with Execution Feedback 101
5.2.3 Modular Fuzzers 102
5.3 Evolutionary Constraint Solving Meets Grey-Box Testing 102
5.3.1 Integrating Soft Constraints 103
5.3.2 Normalizing Soft Constraints 103
5.4 Implementation 105
5.4.1 Instrumentation and Program Analysis 105
5.4.2 Fandango Integration 106
5.5 Domain-Specific Fuzzing Applications 108
5.5.1 Performance Fuzzing 110
5.5.2 Excessive Memory Fuzzing 112
5.5.3 Maximizing Code Coverage 114
5.5.4 Directed Grey-Box Fuzzing 115
5.5.5 Evolving Populations 117
5.5.6 Summary of Results 119
5.6 Threats to Validity 119
5.7 Discussion and Future Work 120
5.8 Conclusion 121

6 Conclusion and Future Work **123**

List of Figures

1 Introduction **1**

2 Background **7**

2.1 High-level model of a program 7
2.2 Illustration of the sets of valid and invalid inputs 8
2.3 High-level model of a program with an input validation stage 8
2.4 Chomsky hierarchy 9
2.5 Context-free grammar for simple arithmetic expressions 10
2.7 Derivation tree for the input *(9*(8+7))*. 12
2.8 Test execution using a comparison-based oracle 17
2.9 Example C function and its control-flow graph 21
2.10 Example C program and its call graph 22
2.11 Test execution using self checks as oracle 24
2.12 Running example: C program for symbolic execution 26
2.13 Control-flow graph of running example 26
2.14 Symbolic execution tree of running example 28
2.15 SMT-LIB constraint example 32
2.16 SMT-LIB constraint to disallow the first solution 32
2.17 SMT-LIB constraint to disallow the second solution 33

3 Symbolic Input Grammar Mining **35**

3.1 STALAGMITE-mined JSON grammar 36
3.2 STALAGMITE overview 38
3.3 Simplified execution trace for *harrydc-json* 41
3.4 Parse tree derived from the simplified execution trace 41
3.5 Snippet from *cjson*: input buffer copy 43
3.6 Harness for *Tiny-C*'s tokenizer 46
3.7 Proxy tokenization function for *Tiny-C* 46
3.8 Function of *calc* that combines parsing and evaluation 48
3.9 Sample execution trace illustrating input consumption identification . . 48
3.10 Input grammar extracted from the simplified execution trace 50
3.11 Token instances aggregated by token ID 52
3.12 Generalized token instances 52
3.14 Optional non-terminal generalization 53
3.17 Grammar accuracy vs. number of input samples 58
3.21 Simplified *mjs* `let` declarations mined by STALAGMITE 63

4 Input Grammar Evaluation **69**

4.1 Nested, ambiguous grammar equivalent to the initial grammar 74
4.2 Graph representation of the grammar in Figure 4.1 75

4.3 Rooted and expanded derivation tree based on a 3-path 78
4.6 Heat map of depth reach for the prevalent sampling technique 86
4.8 Venn diagrams showing covered 3-paths 90
4.9 Growth of k-paths 92

5 Input Constraints and Execution Goals **97**
5.1 Intuitive idea behind FANDANGOGREY 99
5.2 Exponential scaling 104
5.3 Instrumentation workflow with `fcc` 107
5.5 Results for three domain-specific fuzzing tasks 113
5.6 Directed grey-box fuzzing experiment 116
5.7 Coverage-guided fuzzing experiment 118

6 Conclusion and Future Work **123**

List of Tables

1 Introduction **1**

2 Background **7**
2.6 Derivation steps for input generation 12

3 Symbolic Input Grammar Mining **35**
3.13 String classes 52
3.15 Evaluation subjects 56
3.16 Grammar accuracy 58
3.18 Grammar conciseness statistics 60
3.19 Manual effort 61
3.20 Time consumption 62

4 Input Grammar Evaluation **69**
4.4 Comparison of evaluated grammar mining approaches 81
4.5 Grammar accuracy recomputed using GRAMACCU 84
4.7 Golden 1-,2-,3-paths covered by mined grammars 88
4.10 Benefit of k-paths with $k > 1$ 94

5 Input Constraints and Execution Goals **97**
5.4 Execution metrics provided by FANDANGOGREY 107

6 Conclusion and Future Work **123**

1 Introduction

Software is ubiquituous in our everyday life. It helps us with completing our tasks, enables communication and collaboration, and provides access to information. We often take the flawless operation of software for granted, be it on commodity devices such as phones, in private companies that we rely on for services—such as e-mail—in government agencies processing our data, or in control systems for public infrastructure, including airports, power plants or waterworks. However, programs are seldom flawless. They may contain *software bugs*. A bug occurs when a program behaves differently from what was intended, whether by the specification or the developer. The consequences of software bugs vary widely: they may be *minor*—such as a graphical user interface displaying text in a slightly larger font, causing minor inconvenience—*serious*—such as a word processor that occasionally crashes, forcing users to redo unsaved work—or even *disastrous*—such as program failures that put lives at risk or cause substantial financial losses.

Software bugs continue to be a significant challenge in software development. Ideally, we would want to have strict guarantees about the software we use—proofs of its correctness and the absence of bugs. Unfortunately, this is not possible in most cases, as software is often too large and complex for complete formal verification [84]. As an alternative, software is being *tested*. Software testing is one of the most effective and widely used engineering practices to detect bugs. The basic principle is to run the program under test on a given input and to check whether its behavior and output match the expectations. If not, a bug has been found. It really is that simple. On the downside, testing can never guarantee that a program is entirely free of bugs—since each test case checks the program only for a single input at a time—but it can substantially increase software reliability, especially when a large and thoughtfully chosen set of test cases covers both common and uncommon scenarios.

Finding bugs as early as possible in the software development process—preferably before faulty code reaches production—is crucial, because the cost of fixing them significantly increases later on [105]. This is because (1) it may be necessary to make profound changes in order to fix a bug, requiring significant time and effort, (2) bugs may have security-related or business-critical implications that may incur high cost, and (3) fixes need to be sent out to users, which may require additional resources. To address this, developers can write test cases by hand when code changes, but this is tedious and unreliable, which makes automated testing techniques especially valuable.

In recent years, one particular automated software testing technique has gained significant popularity: fuzzing. In essence, a fuzzer[1] tests a program with random or semi-random inputs and checks the program's behavior for crashes or other anoma-

[1] A fuzzer is a program that automatically tests programs with randomly generated inputs.

lies. Fuzzing is successful in practice because it can *automatically* generate large amounts of inputs—either from scratch or based on existing inputs—which are further *evolved* based on code coverage. This enables autonomous exploration of the program's state space without developer interaction.

Despite its success, a key challenge in conventional coverage-guided fuzzing lies in the generation of highly structured inputs that conform to a specific format. To effectively test software that expects highly structured inputs—such as source code or configuration files—grammar-based fuzzing is the prime technique [117]. Grammar fuzzers use a context-free grammar as an input specification to produce random but grammar-conforming inputs. By construction, these inputs are syntactically valid, ensuring that they reach program states beyond parsing, which is crucial for thorough testing.

1.1 Research Problem

The effectiveness of grammar fuzzing comes at a cost: it requires a predefined grammar. Constructing such a grammar manually is tedious and prone to mistakes, particularly because accurately specfiying a program's input format, capturing all the little details, requires a deep understanding of the program.

This limitation gives raise to the field of input grammar mining, which aims to automatically infer an input grammar from a program [125]. Existing approaches can be categorized into two groups: black-box approaches, that only require black-box access to the program—i.e., the ability to execute it on arbitrary inputs and check the output, allowing to construct a *parse oracle* that determines whether an input can be parsed—and white-box approaches, which additionally have access to the program code and runtime information.

All existing approaches share a common limitation: *they require input samples to infer the grammar* [64, 5, 67, 57, 52]. Obtaining representative input samples that cover all edge cases and capture the full details of a program's input format is often challenging. When certain language features are missing in the input samples, existing grammar miners typically fail to learn them, resulting in grammars that are incomplete or inaccurate. This, in turn, can make fuzzing ineffective or inefficient, as parts of the input space—and consequently, program states—remain unexplored.

Grammar inference itself is a hard problem, and fundamental results show that learning a perfect input grammar for every possible program is an unattainable goal. For black-box approaches, it has been shown that inferring input grammars based on black-box membership queries is undecidable [3], and hence such approaches cannot learn perfect grammars in all generality. Likewise, white-box approaches are fundamentally limited by Rice's theorem, which states that determining any non-trivial semantic property of a program (such as the language it recognizes) is undecidable [101]. Consequently, grammar mining research is primarily evaluated based on the *accuracy* of the mined

grammars, indicating *how well* they were inferred. Accuracy is assessed by comparing the mined grammar to a golden grammar that acts as the ground-truth reference model.

Comparing grammars is challenging as well. In fact, determining the equivalence of two context-free grammars is an undecidable problem [101]. As an approximation, existing work [64, 5, 67, 52, 23] typically generates a set of inputs from one grammar and attempts to parse them using the other grammar, thereby estimating precision and recall. While this approach is generally reasonable, we observe that the *prevalent implementation for measuring grammar accuracy is not systematic*: it relies on a fixed amount of input samples regardless of grammar size and favors shorter inputs, which can lead to biased results. However, since grammar accuracy metrics are central to evaluating grammar mining techniques, it is crucial that they reliably reflect the true quality of the inferred grammars.

While evaluating the accuracy of mined grammars is important, their primary practical value lies in enabling applications in test input generation [64], particularly in grammar-based fuzzing. With an accurate input grammar, fuzzers can focus on the set of syntactically valid inputs, allowing for more efficient exploration of program code beyond the initial input validation stage. However, many input specifications are not purely context-free, so context-free grammars alone are insufficient to capture arbitrary input formats. To address this limitation, the recently released Fandango [122] test input generator allows to extend context-free grammars with semantic Python-based constraints, such as checksums or def-use relationships. Using an evolutionary algorithm, Fandango generates inputs that are both syntactically valid and satisfy these semantic constraints.

While this allows test inputs to be shaped with unprecedented ease and flexibility, a key challenge remains: Fandango operates as a black-box tool. Many of the most widely used fuzzers, including AFL [121], rely on grey-box execution feedback—such as coverage information—to guide input mutation and drive program exploration. Integrating execution feedback into Fandango would extend its capabilities beyond black-box input generation, equipping Fandango with the unique ability to *generate valid inputs, shaped according to the tester's needs, which are continuously refined to meet execution objectives.*

1.2 Contributions

To address these limitations and research gaps, this dissertation makes the following contributions to the fields of grammar-based fuzzing and grammar mining:

Contribution 1: Symbolic Input Grammar Mining. Most importantly, this thesis introduces STALAGMITE, the first static input grammar mining technique. This technique overcomes the aforementioned limitation of requiring input samples,

which is shared by all existing grammar mining approaches. Being based on symbolic execution, STALAGMITE does not depend on sample inputs for its analysis, and is generally applicable to recursive descent parsers.

Contribution 2: Systematically Measuring Grammar Accuracy. To enable more reliable grammar comparisons, we introduce GRAMACCU, a systematic technique to compare input grammars based on generating k-path covering inputs [54]. By empirically evaluating the input grammars mined by five existing input grammar mining publications using GRAMACCU, we provide new insights into their effectiveness. Not only does our quantitative evaluation suggest that precision and recall values reported in the respective papers are often overestimated, but we do also unlock a qualitative analysis that assesses which language features were (or were not) learned.

Contribution 3: Combining Input Constraints and Execution Feedback. Building on the Fandango tool, our FANDANGOGREY work for the first time enables users to specify both *input constraints* and *execution goals* in a single unified abstraction. This allows users to incorporate their domain knowledge into the fuzzing process and to implement custom grey-box fuzzers using a simple, declarative specification language.

1.3 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 discusses background knowledge that this thesis builds upon, notably context-free grammars and their application in test generation; as well as intermediate representations commonly used in program analysis, such as call graphs and control-flow graphs. The chapter ends with an introduction to symbolic execution, which summarizes the core idea and points out practical challenges.

Chapter 3 presents STALAGMITE, our static approach to infer context-free input grammars from recursive descent parsers based on symbolic execution. This technique does not require input samples, and has the potential to produce very accurate input grammars.

Chapter 4 introduces GRAMACCU, our systematic metric to evaluate mined input grammars, and subsequently empirically evaluates the grammars mined by five existing grammar mining approaches using GRAMACCU.

Chapter 5 presents FANDANGOGREY, our approach to combine input constraints with execution feedback. This gives users unprecedented control over both the input

and execution domains, providing a rich toolkit for designing and implementing customized grey-box fuzzers.

Chapter 6 concludes this thesis with an outlook on future research opportunities.

1.4 Publications

The following list gives an overview over the publications that I co-authored as part of my doctoral studies. Each publication is briefly summarized.

- **Leon Bettscheider** and Andreas Zeller. "Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. ACM, 2024, pp. 522–526. DOI: 10.1145/3663529.3663790.

  In this short paper, we introduce our initial idea of static input grammar inference from recursive descent parsers using a technique that we call *symbolic parsing*. While this paper introduces the core idea of STALAGMITE, the implementation and evaluation are still limited. This work lays the foundation for the following publication, which extends the initial idea, and evaluates it more thoroughly.

- **Leon Bettscheider** and Andreas Zeller. "Inferring Input Grammars from Code with Symbolic Parsing". In: *ACM Transactions on Software Engineering and Methodology* (2025)

  In this paper, we present our STALAGMITE technique, which leverages symbolic execution to statically analyze recursive descent parsers in order to mine input grammars. We thoroughly explain how we realized STALAGMITE based on the KLEE symbolic execution engine [28] and provide an extensive empirical evaluation. Our evaluation subjects span a diverse set of parsers implemented in C and C++ to demonstrate its effectiveness. Chapter 3 focuses on this paper.

- **Leon Bettscheider** and Andreas Zeller. "How Good are Input Grammar Miners? An Empirical Study". In: *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering, 2026, Rio de Janeiro, Brazil, April 12-18, 2026*. 2026

  In this empirical study, we systematically investigate the quality of input grammars mined by five existing grammar mining approaches. We show how the accuracy metric used by existing approaches is flawed, suggesting that the grammar accuracies reported by the respective authors are biased. To investigate this issue further, we propose a systematic metric based on k-paths [54], utilizing it to thoroughly re-evaluate the grammars mined by the existing approaches. Chapter 4 represents this publication.

- **Leon Bettscheider**, Marius Smytzek, and Andreas Zeller. "Combining Input Constraints with Execution Goals". In: *IEEE Conference on Software Testing, Verification and Validation, ICST 2026, Daejeon, South Korea, May 18–22, 2026*. 2026

  In this paper, we develop a new fuzzing technique that combines input constraints and execution feedback in a unified framework based on the Fandango [122] tool, allowing users to declaratively specify *input properties* as well as *execution properties* in the same abstraction. We empirically evaluate this technique on a number of real-world programs and fuzzing scenarios. Chapter 5 presents this work.

Additionally, I contributed to the following publication, which explores topics related to this thesis but are not part of its content:

- **Leon Bettscheider**. "Learning Test Input Constraints from Branch Conditions". In: *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 248–250. DOI: 10.1109/ICSE-COMPANION58688.2023.00067

2 Background

In this chapter, we review the background knowledge this dissertation builds upon. Section 2.1 focuses on *context-free grammars*, the core formalism we use to define input specifications. Beyond formal definitions, we discuss related topics that are relevant when using grammars[1] for fuzz testing, including parsing and derivation trees, test input generation, as well as fundamental limitations of grammars. We briefly discuss fundamental results and the state of the art in grammar inference, as well as practical directions in enriching grammars with semantics. In Section 2.2, we first outline fundamental limitations of program analysis, and motivate the need for automated software testing. We present and define basic concepts in automated testing—such as *test cases* and *test oracles*—that help to contextualize our work. We proceed to detail two distinct testing methodologies—black-box and white-box testing. For the latter, we first introduce program analysis concepts such as *control-flow graphs* and *call graphs*, which we build upon to discuss fuzz testing and symbolic execution.

2.1 Input Syntax Specifications

In this section, we highlight the importance of formal input specifications, particularly for generating test inputs for programs that process structured data. We also discuss a simple program model and review the relationship between input, computations, and output.

At a fundamental level, programs operate by receiving *input*, performing a sequence of *computations*, and producing *output* that represents the result of these computations. Inputs may be read by the program from various sources, such as files, the network, or the user. In this dissertation, we look at inputs from an abstract standpoint, treating all sources uniformly. We assume that a program's behavior, including both intermediate and final computational results, depends solely on its input. Correspondingly, a simple model of a program is shown in Figure 2.1.

Figure 2.1: High-level model of a program

Exploring and analyzing the behavior of programs is important. For instance, automated software testing (see Section 2.2) checks whether a program behaves as intended by executing it on a large amount of inputs. Doing this requires a method to generate

[1] Unless specified otherwise, we use the terms "grammar" and "context-free grammar" interchangeably.

inputs, which is often facilitated by a *specification* that defines the possible forms inputs can take.

A simple, general model of inputs could be the set of all bit strings, or equivalently, the set of natural numbers. In principle, *enumerating all natural numbers* allows one to *enumerate all possible program inputs*, thereby providing a means to explore all program behaviors and outputs. However, this approach does not scale to realistic programs[2], since they often have extremely large or even infinite input spaces.

A more practical approach is to specify the set of valid inputs accepted by the program. By focusing only on *valid inputs*, the input space is greatly reduced (though it may still be infinite), as illustrated in Figure 2.2.

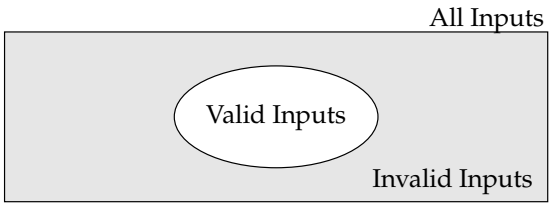


Figure 2.2: Illustration of the sets of valid and invalid inputs

From a testing perspective, while a program must, in principle, handle arbitrary (including invalid) inputs correctly, invalid inputs are usually rejected early, limiting the exploration of the program's behavior to the *input validation code*. When a specification of valid inputs is available, it can be used to generate *valid inputs* that exercise the parts of the program executed *after input validation*, referred to as *evaluation*. These parts might otherwise be reached only rarely by random testing. In this dissertation, we adopt a refined program model in which inputs are first validated and then evaluated if valid, as shown in Figure 2.3.

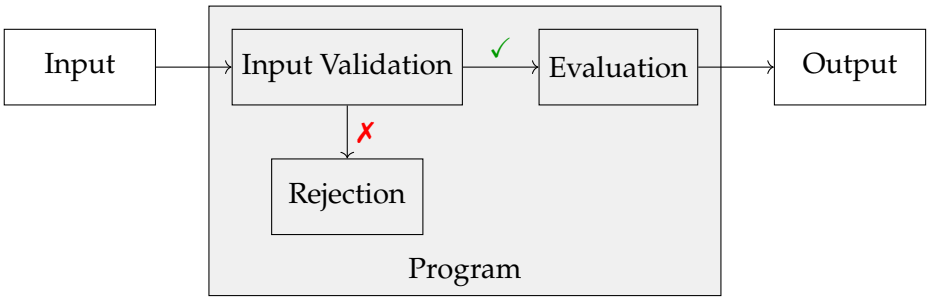


Figure 2.3: High-level model of a program with an input validation stage

[2]The *infinite monkey theorem* states that a monkey typing forever on a typewriter, provided with eternal life, would eventually come up with all of Shakespears' work, and more. Yet, realistically, producing non-trivial texts (or inputs) this way is expected to exceed the lifespan of the universe [118]. Thus, alternative strategies are necessary.

This program model is common in practice. For instance, compilers and web browsers first check the validity of their inputs (e.g., programs) before proceeding with evaluation [56]. More generally, programs that process *structured inputs* typically first parse their inputs, and reject those that are invalid.

Although our focus is on testing, input specifications are also valuable beyond that context. When included in the *documentation* of a program, they benefit both users and developers. Users can consult them to understand which input features the software supports, while developers can use them to determine which inputs their software must handle—for instance, to ensure interoperability between systems.

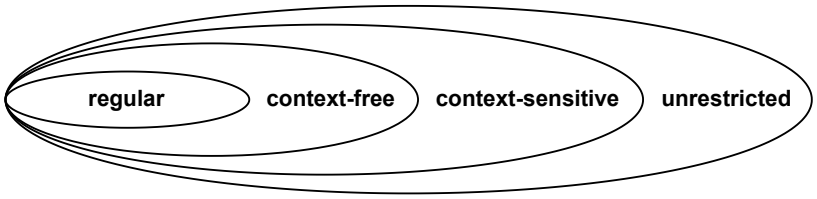


Figure 2.4: Chomsky hierarchy

Input specifications come in different shapes. While natural language descriptions may aid human understanding of the input format, machine-readable, formal specifications are usually preferable: they are more precise and enable automated reasoning as well as input generation. In formal language theory, the *Chomsky hierarchy* [34] (see Figure 2.4) classifies formal languages by their expressive power. In this dissertation, we focus on context-free languages, a particular class of formal languages, which we discuss in the following section.

2.1.1 Context-Free Grammars

Context-free grammars are a commonly used formalism all across computer science. Their popularity stems from the balance they strike between expressivity and complexity. In the Chomsky hierarchy, they stand between the less complex *regular expressions* and the more complex *context-sensitive grammars*. In contrast to regular expressions, context-free grammars support *recursion*, allowing them to describe *nested* languages, such as the language of balanced parentheses. This is not merely an academic advantage; nested structures are at the core of most programming and data description languages, such as C and JSON. In contrast to context-sensitive grammars, there exist efficient algorithms to *recognize* whether a given sentence is part of the language [36], a process known as parsing. In addition, context-free grammars are reasonably easy to work with, making them a good fit for various practical applications. For instance, grammars are used for test input generation already since the 1960s [27, 88].

In the following, we formally define context-free grammars and related concepts, and discuss generating and parsing inputs in detail, which are crucial techniques not only for compiler construction but also for grammar-based test generation. We define context-free grammars and languages as follows (adapted from [101]):

Definition 1: Context-Free Grammar

A *context-free grammar* is a tuple (N, T, R, S), consisting of

1. N, a finite set of *non-terminal symbols*,
2. T, a finite set of *terminal symbols*, which is disjoint from N,
3. R, a finite set of *production rules*, where each production rule is a relation $N \times (N \cup T)^*$ mapping a single non-terminal symbol to a string of terminal and non-terminals,
4. S, the *start symbol*, which is a non-terminal symbol.

To make this definition more concrete, consider the following grammar, which defines the language of parenthesized arithmetic expressions using two binary operations—addition and multiplication—over single-digit operands:

```
<start> ::= <expr>1
<expr>  ::= <digit>2
          | "(" <expr> ")"3
          | "(" <expr> "+" <expr> ")"4
          | "(" <expr> "*" <expr> ")"5
<digit> ::= "0"6 | "1"7 | "2"8 | "3"9 | "4"10
          | "5"11 | "6"12 | "7"13 | "8"14 | "9"15
```

Figure 2.5: Context-free grammar for simple arithmetic expressions

Grammars, such as this one, are often written in Backus-Naur form (BNF). In BNF, non-terminal symbols are wrapped in angular brackets to highlight them visually and to distinguish them from terminal symbols, which are wrapped in quotes. The symbol `::=` means "derives" and the symbol `|` means "also derives" [36]. In other words, `::=` introduces a production rule, and `|` separates alternative production rules.

This form implicitly defines the grammar tuple (N, T, R, S). The set of non-terminal symbols N is made up of all the left-hand sides (here: `<start>`, `<expr>`, and `<digit>`). Terminal symbols T are wrapped in quotes, e.g., `"("` and `"+"`. Production rule alternatives R for a given non-terminal symbol are grouped with an `|`. Unless specified otherwise, `<start>` is the start symbol S.

2.1.1.1 Generating Inputs

Grammars are essentially a finite set of production rules, each tied to a single non-terminal symbol, that collectively define a (potentially infinite) *context-free language*. This context-free language consists of all the *sentences* that can be derived from the grammar start symbol. To derive a sentence, one starts with a string—called a *sentential form*—that initially contains only the start symbol of the grammar but in general consists of terminal and non-terminal symbols. In an iterative process, one then replaces one non-terminal in the sentential form at a time with one of its production rules. At any point, the sentential form represents an intermediate or final step towards a valid derivation. Once the sentential form consists only of terminal symbols, we call it a *sentence*, which is part of $L(G)$ [36].

Definition 2: Context-Free Language

The *context-free language* of a context-free grammar $G = (N, T, R, S)$ is the set

$$L(G) = \{ \omega \in T^+ \mid S \Rightarrow^* \omega \}$$

of sentences that can be derived from the start symbol.
Here, $\Rightarrow^*$ means "can be derived in one or more steps".

Next, we will demonstrate this process on a concrete example. During the derivation process, we need to make a choice whenever we select a production rule for the substituted non-terminal. In this example, we will show a random derivation which can be used for random grammar-based input generation in the context of automated testing. Note that since the left-hand sides of production rules of context-free grammars are always single non-terminals, *the order in which we perform derivations does not matter*. This is what lends context-free grammars their name—derivations are independent of the context.

Our example is based on the grammar shown in Figure 2.5, and the generation steps are shown in Table 2.6. We start the iterative derivation process with the start symbol as the sentential form. In each step, we select the leftmost non-terminal symbol in the sentential form and substitute it with one of its production rules, chosen at random. The chosen production rules correspond to their numbers shown in Figure 2.5. The process terminates when the sentential form consists only of terminals—at that point, we generated a sentence, i.e., a test input.

The input generation process can also be performed at the abstraction level of a *derivation tree*, which maintains the structure of derivations, and thus is a richer representation than the *flat* sentential forms. We exemplarily show the resulting derivation tree in Figure 2.7.

Table 2.6: Derivation steps for input generation

Sentential Form	Rule Application
`<start>`	1
`<expr>`	5
`"(" <expr> "*" <expr> ")"`	2
`"(" <digit> "*" <expr> ")"`	15
`"(9*" <expr> ")"`	4
`"(9*(" <expr> "+" <expr> "))"`	2
`"(9*(" <digit> "+" <expr> "))"`	14
`"(9*(8+" <expr> "))"`	2
`"(9*(8+" <digit> "))"`	13
`"(9*(8+7))"`	

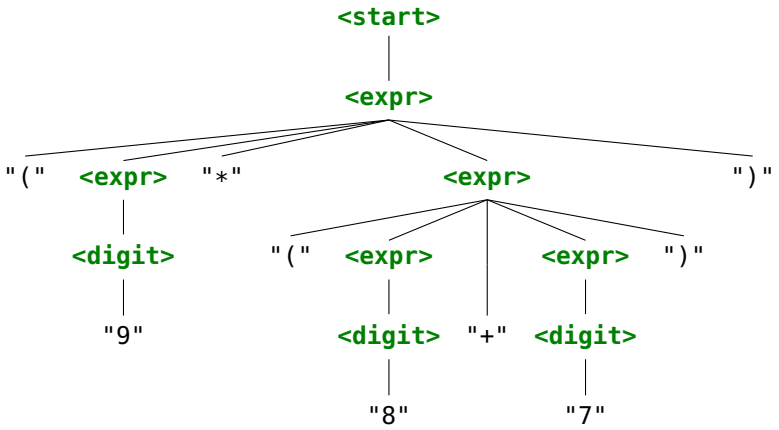


Figure 2.7: Derivation tree for the input *(9*(8+7))*.

One challenge with purely random input generation is the risk of combinatorial explosion, especially when a production rule with multiple recursive expansions is selected (here, for example, `"(" <expr> "*" <expr> ")"`). To deal with this issue, input generation algorithms may favor *cheap* production rules, which are less recursive, once the current expansion reaches a specified depth.

Beyond random generation, input generation may also be guided by *coverage criteria* [101], such as ensuring that each production rule is used at least once, allowing for a more systematic exploration of the input space. One such principled input generation approach is discussed in Chapter 4 as part of our work on comparing input grammars.

Additional grammar-based input generation techniques include *probabilistic grammar fuzzing*, where production rules are assigned *probabilities* to guide input generation. For a comprehensive overview of related methods, we refer to *The Fuzzing Book* [126].

2.1.1.2 Parsing Inputs

Parsing is the task of checking whether a given sentence (i.e., an input) belongs to the language $L(G)$ defined by a grammar G. A *parser* performs this task automatically. In case of a positive result, the parser typically produces a *derivation tree* (or *parse tree*), which shows how the sentence can be derived from the grammar.

In compiler construction, *parsing* takes place in the *frontend* of the compiler, positioned between *lexical analysis* and *semantic analysis*. The resulting derivation tree then serves as an *intermediate representation* of the program, providing a structured form that simplifies further analysis and processing.

Parsing is fundamentally more complex than *input generation*. The *Earley parsing algorithm* can handle *any* context-free grammar, with a worst-case time complexity of $O(n^3)$, where n is the length of the input string. For certain restricted subsets of context-free grammars, more efficient parsing algorithms exist. Two widely used subsets are `LR(1)` and `LL(1)` grammars. These are *unambiguous* grammars (see Section 2.1.1.3) that can be parsed in linear time using only a single token of lookahead. `LR(1)` grammars can be parsed using *bottom-up parsing* and allow left recursion. `LL(1)` grammars form a subset of `LR(1)` grammars. They cannot contain left recursion and can be parsed using *top-down parsers*. A common implementation of `LL(1)` parsing is the *hand-coded recursive descent parser*. Such a parser is built from a set of mutually recursive functions, where each function is responsible for recognizing the language generated by one non-terminal symbol of the grammar [36].

In this work, parsing is important in two ways: First, we present a technique for *statically inferring context-free grammars from recursive descent parsers* in Chapter 3. More broadly, parsers—in particular Earley parsing—play a central role throughout this dissertation for *mutating inputs that conform to a grammar* and for *comparing different grammars*.

2.1.1.3 Undecidable Properties

While context-free grammars provide a simple yet expressive formalism to define languages, many problems involving context-free grammars are undecidable. Below we summarize two problems that are undecidable for context-free grammars [101], which are most relevant for the scope of this dissertation.

Language Equality. In general, determining whether two context-free grammars are equivalent—in the sense that they generate the same language—is undecidable.

Grammar Ambiguity. A grammar is called *ambiguous* if it can parse the same string in different ways, resulting in structurally different parse trees. Some grammars are inherently ambiguous; others can be equivalently represented by an unam-

biguous grammar. In general, it is undecidable if a given grammar is *ambiguous* or not.

2.1.1.4 Inferring Grammars

While the inference of context-free grammars based on black-box membership queries is undecidable in general [3], useful approximations may be learned in certain contexts. Hence, inferring grammars from code, samples, or both, is an active field of research. We present a static approach to learn context-free grammars in Chapter 3.

2.1.2 Enriching Syntax with Semantics

Using context-free grammars to specify inputs restricts one to *syntactic* properties. As illustrated in the Chomsky hierarchy (Figure 2.4), more expressive formalisms exist, such as *context-sensitive grammars* and *unrestricted grammars*, which, in theory, support expressing *semantic* properties. In practice, however, these formalisms are rarely used. Context-sensitive grammars are computationally expensive to parse, making many potential applications prohibitively slow, and they cannot (easily) capture many semantic properties [36]. Unrestricted grammars, which are as powerful as Turing machines, are also not widely used in practice. However, most programming languages, being Turing-complete, offer the same expressive power, but writing ad-hoc input specifications in a programming language like Python is cumbersome [126].

Recently, hybrid approaches have emerged that combine the structure and efficiency of context-free grammars with the expressivity of Turing-complete programming languages. ISLa [106] pioneered this approach, allowing users to specify input languages with a context-free grammar augmented by declarative constraints over grammar nonterminals, expressed in a domain-specific language based on SMT-LIB. For example, ISLa allows one to define an XML specification where opening and closing tags must match—a property that is not context-free. Using ISLa's constraint solving-based input generation technique, one can then generate inputs that satisfy the specification on demand. Checking whether an input conforms to the specification is straightforward, as the constraints can be evaluated concretely without invoking a solver.

A major limitation of ISLa is that constraints must be expressed within specific *solver theories*, which can restrict expressivity or make specifications cumbersome. In addition, the complexity of SMT solving varies by theory, ranging from NP-hard to undecidable [10]. Section 2.2.3.3 provides further background on SMT solving, which is also relevant to symbolic execution.

Fandango [122] builds on ISLa's idea of combining context-free grammars with declarative constraints but replaces the SMT solver with an *evolutionary algorithm*. This allows constraints to be expressed flexibly in Python. We describe Fandango in more detail in the context of our FandangoGrey work in Chapter 5.

2.2 Automated Software Testing

In an ideal world, software could be developed as follows: A developer comes up with a perfect plan for writing the program, meeting all requirements, and then goes on to flawlessly implement the plan in code, making not a single mistake, finally resulting in a program that never needs to be changed again.

Unfortunately, in reality, this is not feasible for non-trivial programs. The most important argument is probably that *humans make mistakes*—for instance, during planning and coding activities, resulting in defective software. Moreover, software often needs to be continuously extended to meet ever-changing requirements. This motivates the need for automated techniques that reason about programs and verify their behavior, a field known as program analysis.

Since the inception of computer science, reasoning about programs—and the limits thereof—has been among the most important research topics. The undecidability of the *halting problem* is the most fundamental result in the field, and was independently proved by Alonzo Church and Alan Turing in 1936 [35, 113]. Informally, the result implies that there cannot be an algorithm that decides for an arbitrary program and an arbitrary input, whether this program will ever halt when run on the given input. This result was later generalized by *Rice's theorem*, which states that determining any non-trivial semantic property of the languages recognized by an arbitrary Turing machine is undecidable [101]. Here, a *non-trivial* property is one that holds for some Turing machines (i.e., programs) but not for others; a *semantic* property is one that can be completely defined with respect to the program executions—in contrast to *syntactic* properties, which depend only on the program code [95]. As a consequence, it is impossible to prove that an arbitrary program satisfies *any interesting execution property*, such as the absence of run-time errors [95].

Despite this, meaningful reasoning about programs properties is achievable. *Static program analysis* techniques such as formal verification, may, for instance, sacrifice automation by requiring the user to supply *invariants* that support analysis efforts. Moreover, they might relax the analysis by permitting *inaccuracies* in terms of *soundness* and *completeness*. For example, a sound but incomplete analysis ensures that whenever it concludes a program satisfies a given property, the conclusion is definitely correct. However, because it is *conservative*, it may fail to recognize that the property holds on some programs where it actually does. In contrast, an unsound but complete analysis will always identify when a program truly satisfies a property. However, it may also mistakenly report (i.e., overclaim) that the property holds for programs where it does not [95]. It is important to note that the meanings of *soundness* and *completeness* can vary depending on context [84]—specifically, whether the property in question refers to ensuring correctness (as in formal verification) or detecting bugs (as in testing). Hence, in testing contexts, the meaning of the two terms may be swapped.

In this dissertation, we focus mostly on *dynamic program analysis* techniques, specifically automated software testing. In principle, testing is simple: one provides an input to the program under test, and subsequently checks whether the output matches one's expectation. If not, the test failed. In manual testing, this process is performed by a human. In contrast, our focus is on *automated* software testing, where tests are executed in an automated way. For some automated testing approaches—such as unit tests—the tests are typically written manually, but executed automatically. In others—like fuzzing and symbolic execution, which we focus on—tests are also *generated* automatically.

Next, we introduce the terminology and basic concepts of automated software testing, followed by a detailed discussion of black-box and white-box testing techniques—specifically fuzzing and symbolic execution.

2.2.1 Terminology

Unless stated otherwise, the terminology in this section is primarily informed by the textbook of Pezzè and Young [84].

Testing is a straightforward technique. One needs to specify the *input*, the *execution conditions* (e.g., the program under test, the execution environment, the initial state), and the expected *output*, or more generally, a *pass/fail criterion*, also known as a *test oracle* when it is software that evaluates the criterion. The combination of these three components is also known as a *test case* [84].

Definition 3: Test Case

A *test case* is a set of inputs, execution conditions, and a pass/fail criterion.

Definition 4: Test Oracle

A *test oracle*[3] is software that applies a pass/fail criterion to a program execution.

One common type is called *comparison-based test oracle*. For such oracles, the test case typically contains a *predicted output* accompanying the test input, which could have been precomputed or derived independently from the program under test or a specification.

Another example of this type of oracle is a *differential test oracle*, which can be established when a second implementation of the program under test is available, which in principle should return the same output for the same input. In this case, the test oracle would check whether the two implementations agree on the output for a given input. One successful application of this differential testing methodology is found in

[3]Unless specified otherwise, we use the terms "oracle" and "test oracle" interchangeably in this thesis.

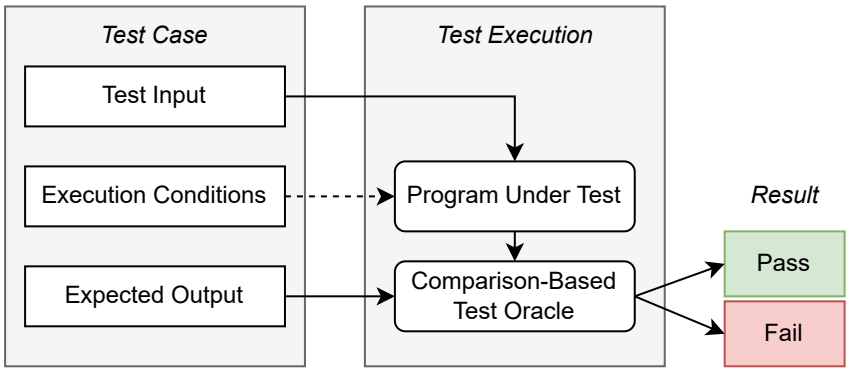


Figure 2.8: Test execution using a comparison-based oracle

compiler testing, with tools such as CSmith [120] comparing the output of a randomly generated program compiled by different compilers.

Another commonly used oracle type is known as *partial test oracle*. A textbook example for such an oracle is testing a program that aims to *find the shortest path between two nodes in a graph*. A partial oracle may, for instance, check that the path returned by the program under test connects the two nodes—without checking that the path is indeed *minimal*. Generally speaking, partial oracles only check conditions that are necessary but not sufficient for correctness [84]. Such partial oracles are also widely used in practice in the area of fuzz testing, which we discuss in detail in Section 2.2.3.2. In essence, fuzzing usually aims at uncovering program failures associated with robustness (such as crashes or memory corruptions) rather than correctness. Hence, a partial oracle in this context may simply check whether the program under test crashes for a given input.

Test execution [84] is the activity of executing a test case, which mainly involves running the program under test on the test input, and evaluating its output with regards to the test oracle (Figure 2.8). If the test was successful, the program behaved correctly with regards to the expectations expressed in the test oracle. If the test failed, however, this indicates a problem, often referred to as a *bug*.

The term *bug* is often used ambiguously in the literature [58], and may refer to either *incorrect program code*, *incorrect program state*, or *incorrect program execution*. In this dissertation, we follow the more precise definitions of Zeller [123]:

Definition 5: Defect, Infection, Failure

Defect: A defect is an incorrect *program code*.
Infection: An infection is an incorrect *program state*.
Failure: A failure is an observable incorrect *program behavior*.

Based on this vocabulary, we can now more precisely express that a failing test case uncovers a *failure*, which commonly leads to the start of a diagnostic process known as *debugging*. Although this dissertation does not focus on debugging, it is closely related to software testing and therefore warrants a brief discussion. *Automated* debugging is the focus of an established scientific field [123] and is even being adopted by tech companies [72]. However, in practice, debugging is often still a *manual* and tedious task. It typically involves a developer that tries to find out *what went wrong*, *where and when*, and ideally ends with the *repair* of the defect. The developer may start with the *failure*—which could be considered as the first observable symptom of the *defect* along a chain of *infections*—and ultimate trace back to the *defect*.

Software testing can be applied at various architectural levels. We reflect on two levels in more detail: testing at the *unit* and the *system* level, following [2]. For the sake of completeness, testing can also be applied at the *acceptance*, *integration*, and *module* level.

Unit testing aims at validating the smallest self-contained units in a software project, such as individual Java classes or C functions [84].

System testing refers to testing the system as a whole, starting from the entry point of the program under test. As we discuss later, fuzzing is often performed at this level.

In the remainder of this section, we focus on software testing techniques that are *automated* in the sense that they require little to no manual input from the tester in generating test cases. These techniques can be broadly categorized into *black-box* techniques, which are agnostic to the implementation of the program under test, generating inputs and checking outputs based on external descriptions such as specifications, and *white-box* techniques, which derive tests from source code or execution information [2]. While we present these two techniques as orthogonal, they can also be *combined*—for example, by using specification-based test input generation in conjunction with a white-box approach [49].

2.2.2 Black-Box Testing

Black-box testing treats the program under test as a *black box*, meaning the developer has no knowledge of its internals, such as code structure or execution. Instead, the developer may define test cases using input and expected output pairs, and the program behavior may be verified by checking whether the actual output matches the expected result. Due to this focus on the relation between inputs and outputs, this approach is also known as *functional testing*.

This approach offers several advantages and disadvantages. Key benefits include:

- Black-box testing is generally easier to implement than white-box testing.
- Tests are independent of the program's implementation, reducing bias related to code structure.
- Test execution has *no overhead*, unlike white-box testing, which may add overhead from instrumentation or static analysis.

However, there are also some disadvantages:

- Because programs often have vast or even infinite input space, generating purely random test cases is usually ineffective. Instead, access to a functional specification is required to derive meaningful test cases than cover a wide range of program behaviors.
- Even with such a specification, mismatches can occur between the specified input language and the actual input language accepted by the implementation. If the specification is too permissive or too strict, this can reduce the efficiency and effectiveness of the tests.

Another challenge arises when test cases are generated automatically, which is the focus of our work. For complex programs tested at the system level, *input specifications*—such as a context-free grammar—are often available and can be used to automatically derive test inputs from (see Section 2.1.1.1). However, *functional specifications* that define the expected output for a given input are rarely available at the system level. As a result, traditional comparison-based test oracles are often impractical. An exception is differential testing (see Section 2.2.1), which can be used in certain scenarios. Because of this limitation, more *generic* test oracles are typically employed—focusing on robustness rather than functional correctness. We explore this further in Section 2.2.3.2 on Fuzz Testing.

2.2.3 White-Box Testing

White-box testing derives test cases based on program code or execution feedback. A key advantage of this approach is that it does not require a functional or input specification. Instead, meaningful test inputs can be derived using information obtained from static or dynamic analysis of the code. However, a drawback is that white-box testing requires specialized tools, which often need to be implemented specifically for the programming language used in the program under test. Additionally, the static or dynamic analyses involved can introduce performance overhead.

In the following, we focus on two white-box testing techniques—fuzzing and symbolic execution—both of which automatically generate test inputs. Because functional specifications are typically not available, these techniques rely on *partial* oracles (see Section 2.2.1), commonly in the form of implicit or explicit self checks, such as assertions,

operating system-issued faults, or compiler-assisted sanitizers. We elaborate on this in Section 2.2.3.2.

Fuzzing is a relatively lightweight white-box testing technique that often relies only on runtime feedback. For this reason, it is often referred to as a *grey-box* technique. In contrast, symbolic execution is considered heavyweight, as it involves precisely tracking program behavior by means of *symbolic expressions* and solving them using *SMT solvers*.

Before discussing these two techniques in detail, we first introduce selected intermediate representations of programs that both techniques build upon. These representations are more amenable to automated analysis than source code.

2.2.3.1 Program Structure Representations

It is often useful to construct models from program source code that abstract away details that are not relevant to the current analysis, while preserving all information necessary for it. The aim is to produce an abstraction that is well-structured and suitable for analysis. In the following, we discuss two basic concepts that are at the core of compiler construction and many dynamic and static program analysis techniques, including fuzzing, and symbolic execution. We first discuss *intraprocedural* control-flow graphs, which models the control flow between atomic units (i.e., *basic blocks*) within a function. We then define and examine *call graphs*, a simple abstraction to describe *interprocedural* control flow between functions.

Control-Flow Graph

Before defining and discussing control-flow graphs, we first define their fundamental building blocks: basic blocks.

Definition 6: Basic Block

A *basic block* (BB) is a maximal-length sequence of branch-free code [36].

Elaborating on this definition, control can only be transferred to the first operation of a basic block, leading to a sequential execution of all operations in the basic block until the last operation. There may not be any branches inside the basic block, except for the last operation, which either *jumps unconditionally* to another basic block or *branches* to multiple other basic blocks.

Building on this definition of basic blocks, we define control-flow graphs in accordance with [36]:

Definition 7: Control-Flow Graph

A *control-flow graph* (CFG) is a directed graph (N, E), consisting of

1. N, the set of nodes, which are *basic blocks*.
2. E, the set of edges, where each edge $(n_i, n_j) \in N \times N$ connects two nodes, implying a possible transfer of control from n_i to n_j.

To make this definition more concrete, we present an example. Consider the function `f` and the corresponding control-flow graph shown in Figure 2.9.

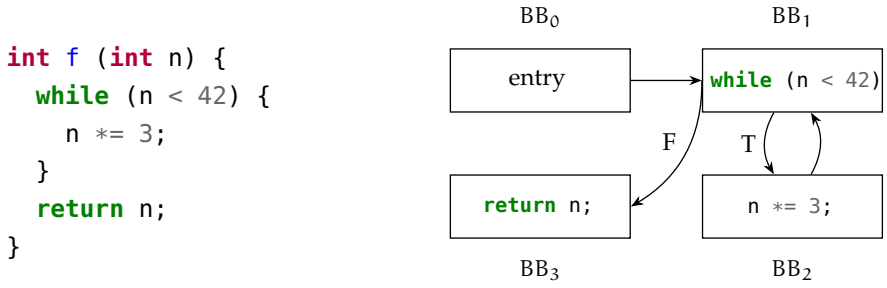


Figure 2.9: Example C function and its control-flow graph

We see that the program actually consists of four basic blocks. The *while*-loop is realized using a comparison and a conditional jump in BB_1 to either BB_2 or BB_3, and an unconditional jump from BB_2 back to the loop header in BB_1.

Following LLVM's loop terminology[4], we define the relevant terms below. This is particularly relevant for Chapter 3, which leverages loop analysis based on LLVM. In LLVM, loops are canonicalized in a way such that loops are strongly connected subgraphs, with all control-flow edges from outside this subgraph into it necessarily pointing to the same node, called the *loop header* (here: BB_1). An *entering block* is a basic block that is not part of the loop but has an edge into the loop (here: BB_0). A *latch* is a basic block, which is part of the loop, and has an edge to the loop header (here: BB_2). A *backedge* is a control-flow edge from the latch to the loop header (here: (BB_2, BB_1)). An *exiting edge* is a control-flow edge that leaves the loop (here: (BB_1, BB_3)). We call the source of this edge an *exiting block* (here: BB_1) and the destination an *exit block* (here: BB_3).

As noted earlier, control-flow graphs abstract away details that are not essential for control-flow analysis. In particular, the *contents* of basic blocks—i.e., the individual operations—may be disregarded. As a result, *data flow* can be ignored, and sequences of operations can be represented collectively within a single basic block. For some use cases, however, control-flow graphs may be augmented with data-flow information extracted from individual instructions. Therefore, while control-flow graphs provide a valuable abstraction, they may be used flexibly in practice.

[4] `https://llvm.org/docs/LoopTerminology.html`

Moreover, the control flow model of control flow graphs is not necessarily complete. On one hand, *interprocedural* control flow is typically omitted. For instance, a *function call* may appear within a basic block without introducing a control-flow edge. On the other hand, depending on the implementation, *intraprocedural* control-flow edges caused by *exceptions* may also be excluded, as including them could make the control-flow graph overly complex [84].

Call Graph

Call graphs are a model to capture interprocedural control flow, specifically calls between functions. For clarity, we define call graphs in a simple way, where each node corresponds to a function, following [84]. A more precise definition is given in [36], where each edge originates from a *call site* (i.e., the specific *call instruction* within the source function) and points to the destination function.

Definition 8: Call Graph

A *call graph* (CG) is a directed graph (N, E), consisting of

1. N, the set of nodes, which are *functions*.
2. E, the set of edges, where each edge $(n_i, n_j) \in N \times N$ connects two functions, representing a call from n_i to n_j.

As an example, consider the C program and its corresponding call graph shown in Figure 2.10. In this program, the function `f` calls itself recursively as well as another function, `g`. For simplicity, the definition of `g` is omitted, with the assumption that it does not contain any function calls. The call graph on the right models this program by representing the functions as nodes and calls between them as edges.

```
int f (int n) {
  if (n)
    return f(n-1) + g(n);
  return 0;
}
```

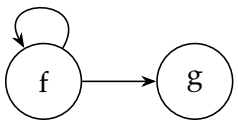


Figure 2.10: Example C program and its call graph

In general, constructing a call graph is often more complex than building an intraprocedural control-flow graph due to several factors, such as [36, 84]:

Function pointers. When functions are called indirectly via pointers, it may be difficult to statically determine the precise set of functions that can be called.

Runtime resolution of ambiguous calls in object-oriented programs. In object-oriented programming languages that support *dynamic dispatch*, it is not possible

to statically determine the precise set of functions within the class hierarchy that may be called at runtime.

Context sensitivity. As discussed above, function calls can be distinguished based on their call site. More fine-grained context-sensitive analyses may go even further by incorporating additional information—such as extending the notion of a *call site* to a *call chain*, or by tracking values passed as arguments to functions.

2.2.3.2 Fuzz Testing

Fuzzing (or fuzz testing) is an automated testing technique that primarily focuses on detecting crashes and other security-related failures.

First explored in 1990 by Barton Miller et al., fuzzing began as a primitive black-box technique that fed random inputs into programs to check their robustness. This early work showed that over 24% of 90 tested Unix utilities could crash [75]. Since the 2000s, computers have become ubiquitous and increasingly interconnected. As a consequence, ensuring the security and robustness of software has become a focus, which led to a steep rise in interest in fuzzing, both in research and practice.

A major milestone was the 2013 release of the American Fuzzy Lop (AFL) [121] fuzzer, which popularized *coverage-guided fuzzing*—a technique that uses lightweight execution feedback to guide input generation. Other notable tools that follow a similar paradigm include libFuzzer [99], honggfuzz [110], and AFL++ [44]. Coverage-guided fuzzers typically employ mutation, i.e., they modify existing inputs based on execution feedback. An alternative approach is grammar-based fuzzing, which can both generate inputs from scratch given an input grammar (see Section 2.1.1.1), as well as mutate existing inputs while preserving their syntactic validity. Fuzzers also often apply *code sanitizers*, i.e., instrumentations that detect faulty behavior such as memory corruption. These acts as *generic test oracles* that are capable of identifying certain security-related failures.

Next, we examine coverage-guided fuzzing in more detail, followed by an overview of code sanitizers and grammar-based fuzzing.

Coverage-guided fuzzing

Coverage-guided fuzzing is one of the most widely used fuzzing techniques and has proven highly effective at uncovering security-related bugs in practice. For example, as of May 2025, Google's OSS-Fuzz had found over 13,000 vulnerabilities and 50,000 bugs across 1,000 open-source projects[5].

[5]https://github.com/google/oss-fuzz

The core principle is to evolve an initial set of *seed* inputs towards maximizing code coverage, thereby increasing the likelihood of discovering bugs in the executed code. This process typically follows a genetic algorithm that continuously:

1. **Selects** an input from the set of retained inputs (*corpus*) based on its *fitness*, which is usually measured by *code coverage* or *branch coverage* in the program under test, and adjusted by a *power schedule* that accounts for how often the input has been selected previously [127],

2. **Derives** a new input from the selected one by means of *mutation* (i.e., randomly changing substrings) or *crossover* (i.e., replacing a substring with that of another input),

3. **Evaluates** the new input by executing it in the program under test with instrumentation to record code coverage at runtime. Inputs that trigger interesting behavior, such as covering previously unseen code, are added to the corpus.

Sanitizers

Fuzzers typically do not target functional correctness issues—detecting those often requires a formal specification—but instead focus on robustness issues with potential security implications. To detect such issues, they employ code sanitizers as test oracles. These are implemented as compiler instrumentation passes that insert *self-checks* into the code. A widely used sanitizer is AddressSanitizer [98]. In a nutshell, it can detect memory corruptions such as out-of-bounds reads and writes and use-after-free failures, even in cases that do not cause a program crash. Other notable sanitizers include MemorySanitizer [108] and UndefinedBehaviorSanitizer [111].

In general, sanitizers can be viewed as self-checks—similar to runtime assertions—that eliminate the need for an expected output in the test oracle. This concept is illustrated in Figure 2.11.

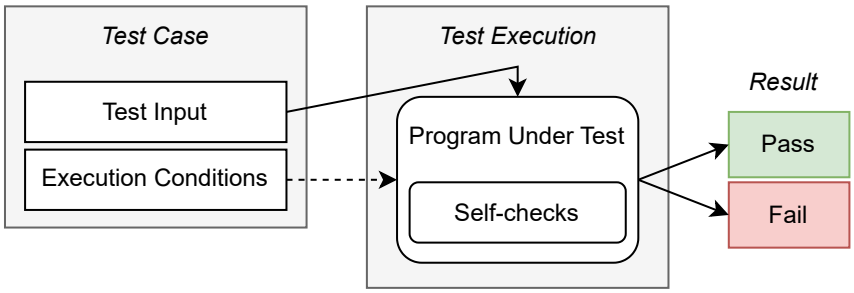


Figure 2.11: Test execution using self checks as oracle [84]

Grammar-based fuzzing

We first introduced grammar-based input generation and the underlying context-free grammars in Section 2.1.1. Here, we revisit the topic from a fuzzing perspective, expanding on key concepts and related work.

The key advantage of grammar-based fuzzing is that generated inputs are random yet syntactically valid by construction. This is particularly helpful when testing programs with an initial parsing stage, which first validates that inputs are syntactically valid before they are processed further. By ensuring syntactic validity, grammar-based fuzzing increases the likelihood of exercising and finding defects in the program logic that follows parsing.

In addition to generating inputs from scratch, grammars can be used to *mutate* existing inputs. A pioneering approach is LangFuzz [56], which uncovered over 2,600 bugs in JavaScript engines. Its key to success was to leverage inputs from bug repositories that were known to cause failures in the past. The assumption is that the underlying defects might have been only partially fixed, and thus could be retriggered with slightly modified inputs.

While grammar-based fuzzing is primarily a black-box technique, it has also been combined with white-box techniques. For instance, Godefroid, Kiezun, and Levin integrated it with symbolic execution [49]; Aschermann et al. combined it with coverage-guided fuzzing [8]; and in Chapter 5, we present an approach that integrates grammar-based fuzzing with semantic input constraints and execution feedback.

2.2.3.3 Symbolic Execution

> Unless stated otherwise, the discussion of symbolic execution in this section is primarily informed by the survey by Baldoni et al. [10].

In this section, we focus on *symbolic execution*, a systematic approach to explore programs, reason about their execution, and automatically generate test inputs based on SMT solving. Symbolic execution was first described in 1976 [63]. In the following, we give an intuitive introduction to symbolic execution by demonstrating the core idea on a simple C program shown in Figure 2.12.

This program reads in two integer inputs and checks them using three if conditionals. Notably, the program can crash if line 9 or line 11 is reached. While finding crashing inputs is not the only application of symbolic execution, for now we will focus on this use case and guide our example along the question: *How can we automatically test this program and systematically find crashing inputs?* Let us first consider the control-flow graph of this program, shown in Figure 2.13. There are four potential execution paths through this program. Symbolic execution provides a means to explore them all. In contrast to concrete execution (i.e., the *normal* way of running a program), symbolic

```
1  void program() {
2      uint32_t x = input();
3      uint32_t y = input();
4      uint32_t a = y*2;
5      if (x == a)
6          if (x == 42) {
7              y += 1;
8              if (y < 22)
9                  crash1();
10             else
11                 crash2();
12         }
13 }
```

Figure 2.12: C program serving as the running example to illustrate symbolic execution

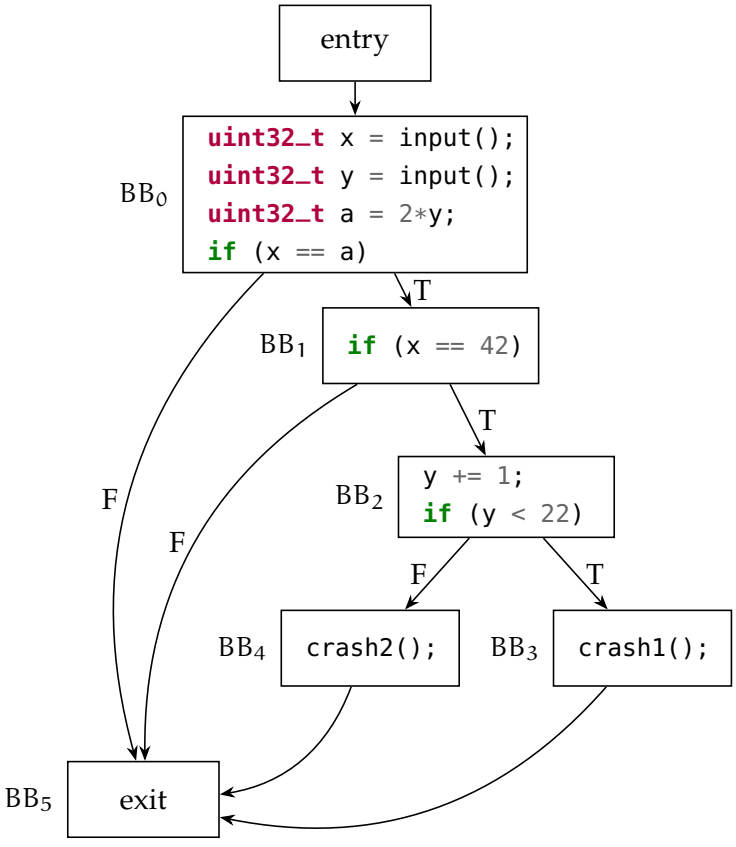


Figure 2.13: Control-flow graph of the running example C program

execution treats the input (here: x and y) as a symbolic variable that can take arbitrary values. Upon reaching a conditional branch that depends on the input, this yields a situation where *both branches* can potentially be taken. Symbolic execution handles this situation by *forking* the execution state, and storing the chosen branch decision in the respective forked execution state. More specifically, as part of the execution state, symbolic execution maintains a path constraint *pc* and a symbolic store *sym*.

Initially, *pc* holds the value *true*. When a branch is taken, *pc* is then conjoined with the corresponding branch condition, effectively encoding the constraints that must hold on the input to make the program take the same execution path (i.e., the sequence of branch decisions) as the current execution state. If the branch condition does not have a data-flow dependency on the input, it is a constant, i.e., *true* or *false*; otherwise, it is a quantifier-free boolean expression over input variables.

On the other hand, *sym* is a mapping from program variables to symbolic expressions over input variables. *sym* is updated whenever a variable assignment *lhs := rhs* occurs in the program. If *rhs* does not have a data-flow dependency on the input, *sym[lhs]* is set to a constant. However, if the input flows into *rhs*, *sym[lhs]* will be set to the symbolic expression over input variables of *rhs*. Internally, symbolic execution engines resolve memory accesses using *sym* instead of the actual memory; thus the input variable is continuously tracked throughout the execution.

To illustrate the symbolic execution of the program in Figure 2.12, the *symbolic execution tree* is presented in Figure 2.14. Each node represents a symbolic execution state and stores the current *pc* and *sym*. Consistent with Figure 2.13, each node is labeled with its basic block ID, and additionally with an if-condition, if the basic block ends with a conditional branch. Moreover, basic blocks $\texttt{BB}_3$ and $\texttt{BB}_4$ are annotated to indicate that they are *crashing*. Edges represent control-flow-induced transitions between execution states. Leaf nodes do not have successors because the program has either terminated successfully or unsuccessfully, e.g., by crashing.

One of the core features of symbolic execution is its ability to determine whether an execution state can be reached and to generate one or multiple inputs that, when run concretely in the program under test, would take the same execution path up to and including the target execution state. To exemplify this, the bottom four boxes that are connected to the leaf nodes with dashed lines show a satisfying assignment of values to the input variables, i.e., a solution for *pc*. These solutions are computed by an SMT solver (Section 2.2.3.3). In general, there may be more than one solution for a constraint. While symbolic execution engines often return only a single solution, they can easily be set up to enumerate multiple solutions by instructing the SMT solver to disallow all of the previous solutions in addition to *pc*.

It is also worth mentioning that, in general, path constraints may be unsatisfiable. For instance, the path constraints required to reach $\texttt{BB}_3$ cannot be satisfied. This is because

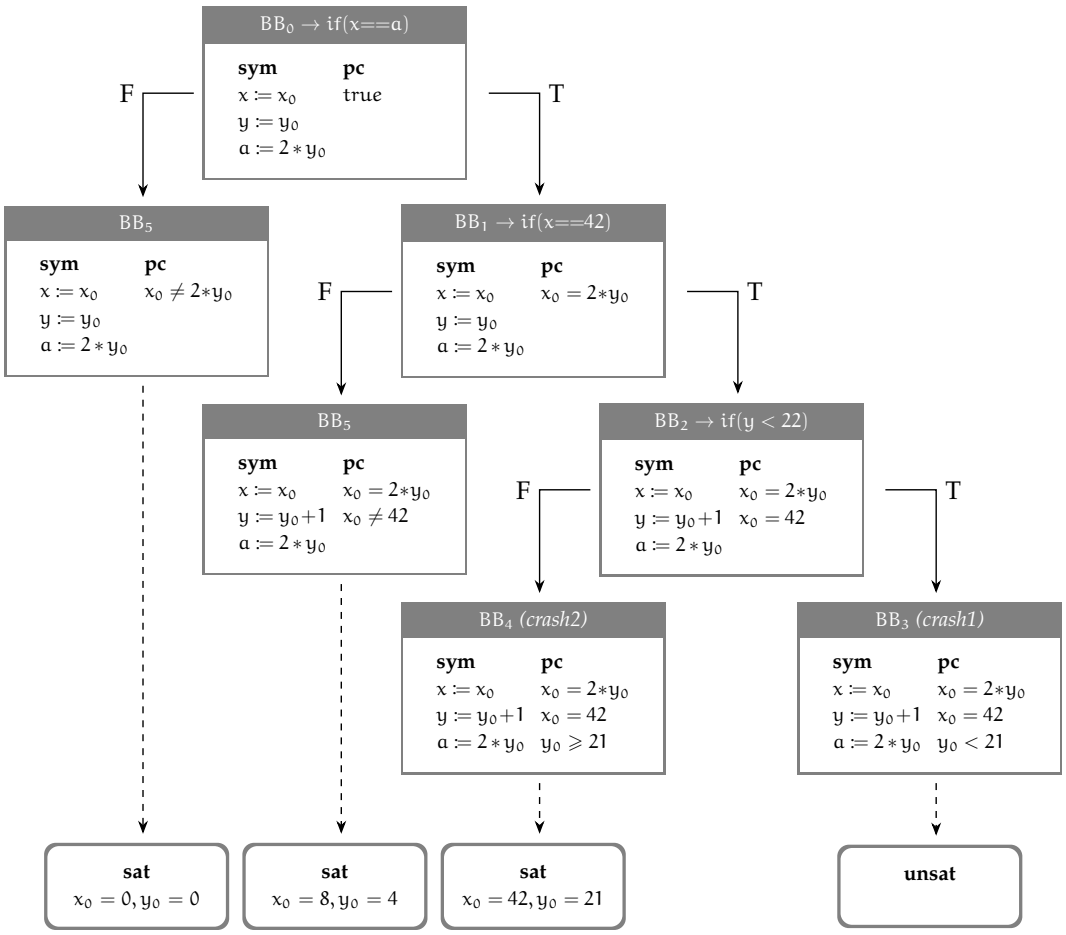


Figure 2.14: Symbolic execution tree of the running example C program

the branch condition

$$\texttt{if}(y < 22) \qquad (\textit{true}\text{-branch of } \texttt{BB}_2)$$

which, due to an increment of y in $\texttt{BB}_2$, translates to the constraint

$$y_0 < 21$$

contradicts the two previous branch constraints

$$x_0 = 2 * y_0 \wedge x_0 = 42 \qquad (\textit{true}\text{-branches of } \texttt{BB}_0, \texttt{BB}_1)$$

which can only be satisfied by the assignment:

$$x_0 \in \{42\} \wedge y_0 \in \{21, 2147483669\}$$

As a result, $\texttt{BB}_3$ and thus *crash1()* cannot be reached. In contrast, $\texttt{BB}_4$ and consequently *crash2()* can be reached by one of the two inputs $(x_0 \rightarrow 42, y_0 \rightarrow 21)$ and $(x_0 \rightarrow 42, y_0 \rightarrow 2147483669)$. Note how symbolic execution also keeps track of *type information* and leverages the reasoning capabilities of an SMT solver to find out that, due to unsigned integer overflow semantics, $y_0 \rightarrow 2147483669$ is equivalent to $y_0 \rightarrow 21$.

To summarize, symbolic execution can find out which of the four execution paths in the example are feasible, and generate a concrete input for each of the feasible execution paths that would follow the same execution path under concrete execution.

In this example, the symbolic execution tree is *finite*. However, in general, it may be *infinite* in presence of unbounded loops, and too large to explore exhaustively in most realistic use cases. This circumstance gives raise to a number of practical challenges, which we briefly discuss at the end of this section.

SMT Solving

This section gives a gentle introduction to *SMT solving*. As this thesis primarily utilizes SMT solvers within the context of symbolic execution, we do not pursue mathematical rigor; instead, we adopt a user-oriented perspective, providing an overview of SMT solvers that is suitable for black-box users. For more in-depth information, we refer to the summaries of Barrett et al. [12, 14] and the SMT-LIB standard [103].

We start by describing the SAT problem, and subsequently lead over to the related SMT problem, which motivates the need for SMT solvers. Next, we discuss the SMT-LIB initiative [13], which aims to standardize interfaces of SMT solvers, followed by a survey of popular SMT solvers, and end with a concrete example that showcases an SMT constraint modeling a symbolic path constraint from the running example.

Introduction and relation to SAT solving. The *boolean satisfiability problem* (SAT) is a classical problem in computer science. This problem centers on determining

whether there exists an *interpretation* of the variables in a given boolean expression (i.e., a propositional logic formula) that results in this expression evaluating to *true*. SAT is the first problem that was proved to be NP-complete [62]. Moreover, deciding the satisfiability of a boolean expression is as hard as finding a satisfying assignment of variables. While no efficient algorithm is known to solve the SAT problem with polynomial worst-case time complexity, *SAT solvers* are commonly based on solving procedures such as the Davis–Putnam–Logemann–Loveland (DPLL) [38] and conflict driven clause learning (CDCL) [100] algorithms.

SMT generalizes the boolean SAT problem to expressions of a specific (background) theory, i.e., problem domain. Typically, these theories are suitable to express constraints that are capable of modeling complex software and hardware systems. Some of the common theories include [104]:

- *The theory of integers* may be used to express both linear and non-linear arithmetic constraints over integers.
- *The theory of fixed-size bit-vectors* allows to express constraints involving fixed-size sequences of bits (bit-vectors) combined with arithmetic operations, bit-wise operations, concatenation, and more.
- *The theory of strings* facilitates the precise formulation of constraints involving string operations and relationships.

One of the strengths of SMT solvers is their modularity, allowing them to combine different theories and, consequently, their solving procedures. As an example, by combining the *theory of strings* with the *theory of integers*, a solver could reason about both the string representation of integers, as well as the integer representation of numerical strings, even in the context of a larger constraint.

While solving SMT constraints remains a challenging and fundamentally hard task, improving the performance of SMT solvers is a research focus and subject to competitions such as SMT-COMP [102].

Language standardization. The SMT-LIB initivate [13] supports SMT research by standardizing languages and interfaces, which enables the comparison of SMT solvers and promotes advancements in the field.

Specifically, the three main goals of the initiative are [13]:

1. providing rigorous descriptions of background theories,
2. developing and standardizing common input and output languages for SMT solvers,

3. establishing and making available a large library of more than 100.000 benchmarks for SMT solvers.

Moreover, SMT-LIB also allows to specify *logics*, which are subsets of *background theories* and combinations thereof, tailored to specific types of problems. This enables solvers to implement specialized solving procedures for certain use cases. For instance, the theory of integers can be divided into quantifier-free or linear subsets, that may model the complexity of the problem at hand more adequately.

Most SMT solvers support the SMT-LIB standard, enabling competitive benchmarking and allowing solvers to be interchanged seamlessly within user programs, such as symbolic execution engines.

Implementations. The following list summarizes some of the most widely used SMT solvers, all of which support the SMT-LIB language, are available under open-source licenses, and are actively maintained at the time of writing.

Z3 Developed by Microsoft Research, Z3 [77] is one of the most popular SMT solvers. It implements a wide array of theories, enabling it to reason about constraints involving arithmetic, fixed-size bit-vectors, uninterpreted functions, quantifiers, arrays, and more.

CVC The Cooperating Validity Checker (CVC) family of SMT solvers [16, 17, 11] is developed at Stanford University and the University of Iowa. CVC5 supports all of the theories specified in the SMT-LIB standard, as well as non-standard theories.

Yices2 Built at the Computer Science Laboratory of SRI International, Yices2 [41] is another SMT solver that supports an extensive collection of theories, including linear and non-linear arithmetic, bit-vectors, and uninterpreted functions.

STP The Simple Theorem Prover (STP) [48] focuses on the theory of quantifier-free bitvectors (QF_BV), which is appropriate for many tasks in program analysis and testing. While it only supports this one theory, it regularly performs well on benchmarks such as SMT-COMP [109, 15].

Back to the Running Example. We now revisit the running example and showcase a concrete SMT-LIB constraint (Figure 2.15) that a symbolic execution engine might generate to encode the input constraints necessary to reach $\texttt{BB}_4$ in Figure 2.13. In the following, we discuss this constraint in detail. The first line instructs the SMT solver to use the *QF_BV* theory, which supports *quantifier-free formulas over the theory of fixed-size bitvectors* [103]. Lines 3–4 then declare the input variables *x_0* and *y_0* as bit-vectors of size 32 to model the `uint32_t` C data type. A bit-vector is a fixed-size sequence of bits. Note that bit-vector declarations do not contain information about signedness. Instead,

the interpretation of bit-vector signedness is encoded in operations [94]. Specifically, this is reflected by the "u" in the `bvult` operation in Line 11. The *assert*-statements in Lines 7, 9, and 11 encode the branch constraints, and Lines 13–14 instruct the solver to check satisfiability and, if possible, return a satisfying assignment for the input.

```
1  (set-logic QF_BV)
2
3  (declare-const x_0 (_ BitVec 32))
4  (declare-const y_0 (_ BitVec 32))
5
6  ; BB0 -> BB1: assert (x == 2 * y);
7  (assert (= x_0 (bvmul #x00000002 y_0)))
8  ; BB1 -> BB2 (0x2a=42d): assert (x == 42);
9  (assert (= x_0 #x0000002A))
10 ; BB2 -> BB3 (0x15=21d): assert (!((y + 1) < 21));
11 (assert (not (bvult (bvadd y_0 #x00000001) #x00000015)))
12
13 (check-sat)
14 (get-model)
```

Figure 2.15: SMT-LIB constraint modeling the input constraint to reach BB_4 in Figure 2.13

When run on this constraint, Z3 returns one of the two possible solutions, such as:

$$x\_0 \coloneqq 0x2a \quad y\_0 \coloneqq 0x15$$

To query the solver for another solution, it suffices to add an additional constraint that disallows the first solution:

```
1  (assert (not (and (= x_0 #x0000002a) (= y_0 #x00000015))))
2  (check-sat)
3  (get-model)
```

Figure 2.16: SMT-LIB constraint to disallow the first solution

Z3 now happily returns the second solution:

$$x\_0 \coloneqq 0x2a \quad y\_0 \coloneqq 0x80000015$$

Finally, we disallow also the second solution to exhaustively enumerate all possible solutions. The `check-sat` command now returns **unsat** as no further solution is possible.

```
(assert (not (and (= x_0 #x0000002a) (= y_0 #x80000015))))
(check-sat)
(get-model)
```

Figure 2.17: SMT-LIB constraint to disallow the second solution

Challenges

Symbolic execution offers a systematic approach to exploring programs and generating test inputs. However, its practical application is hindered by several well-known challenges, which are outlined below.

Path Explosion A central challenge in symbolic execution is the *path explosion problem*. Each time the symbolic execution encounters a conditional statement (e.g., an `IF-THEN-ELSE` construct), the engine *forks* the execution to explore both branches, producing two execution states that capture the possible outcomes. This forking causes the number of execution states to grow exponentially with program size. Loops and recursion amplify the issue, as they repeatedly trigger forking through re-execution of code segments.

This exponential increase in execution states leads to two key difficulties:

1. The available *memory* may be exhausted by storing all execution states.
2. The *execution time* also grows exponentially with the number of states.

A common mitigation strategy is to bound the exploration depth of loops and recursive calls. Although this risks incomplete program exploration, it can be an acceptable tradeoff in some scenarios. We adopt this approach in our work (Chapter 3). Another approach relies on *search heuristics*, which prioritize which execution states should be explored next.

Constraint Complexity Another major challenge is the complexity of constraint solving. In principle, SMT solvers allow for formulating a wide range of constraints that arise in programs, and many constraints can be solved in practice. However, some constraints can take prohibitively long to solve, and for others no decision procedure exists. Recall that the SAT problem is already *NP-complete*, and that certain classes of constraints—such as non-linear integer arithmetic—are *undecidable* [10]. As a consequence, symbolic execution may fail to explore certain program paths, even when a solution exists.

Symbolic Memory Model A memory model maps program variables to *symbolic expressions over input variables*, which represent their content. In low-level languages such as C, however, variables are accessed through *memory addresses*, which themselves may be symbolic. Hence, it is challenging to build a comprehensive model. Some

symbolic execution engines support symbolic memory addresses to a limited extent—though in general this is infeasible, since such addresses may reference arbitrary locations and overwhelm the analysis. Other engines *concretize* symbolic addresses, restricting each to a single possible concrete value. Concretization reduces complexity at the expense of possibly missing valid execution paths.

Environment Model Modeling the environment presents another challenge, since programs rarely operate in isolation. Programs commonly interact with external components, such as the operating system (e.g., through files or system calls), or other programs, which may be implemented in different programming languages or run remotely. When symbolic execution is applied only to the main program, memory tracking can become incomplete whenever data crosses these environment boundaries.

Common strategies to address this challenge include: (1) concretizing memory at environment interfaces, (2) symbolically executing the entire system via virtualization [33], and (3) providing specialized models for common interfaces [28].

3 Symbolic Input Grammar Mining

This chapter is taken, either directly or with minor modifications, from our 2025 TOSEM paper *Inferring Input Grammars from Code with Symbolic Parsing*.
My contribution in this work is as follows:
① contributions to original idea,
② implementation, and
③ evaluation.

Chapter Overview

Generating effective test inputs for a software system requires that these inputs be *valid*, as they will otherwise be rejected without reaching actual functionality. In the absence of a specification for the input language, common test generation techniques rely on *sample inputs*, which are abstracted into matching grammars and/or evolved guided by test coverage. However, if sample inputs *miss* features of the input language, the chances of generating these features randomly are slim.

In this work, we present the first technique for *symbolically* and *automatically* mining input grammars from the code of recursive descent parsers. So far, the complexity of parsers has made such a symbolic analysis challenging to impossible. Our realization of the *symbolic parsing* technique overcomes these challenges by (1) associating each parser function `parse_ELEM()` with a nonterminal `<ELEM>`; (2) limiting recursive calls and loop iterations, such that a symbolic analysis of `parse_ELEM()` needs to consider only a finite number of paths; and (3) for each path, create an expansion alternative for `<ELEM>`. Being purely static, symbolic parsing does not require seed inputs; as it mitigates path explosion, it scales to complex parsers.

Our evaluation promises symbolic parsing to be highly accurate. Applied on parsers for complex languages such as *Tiny-C* or *json*, our STALAGMITE implementation extracts grammars with an accuracy of 99–100%, widely improving over the state of the art despite requiring only the program code and no input samples. The resulting grammars cover the entire input space, allowing for comprehensive and effective test generation, reverse engineering, and documentation.

```
<start> ::= <json_parse_value>
<json_parse_value> ::=
      <opt_sign> <number>
    | <opt_sign> <nan>
    | <opt_sign> <inf>
    | "[" <json_parse_array>
    | "{" <json_parse_object>
    | "\"" <opt_any_str> "\""
    | "null" | "false" | "true"
<opt_any_str> ::= "" | <any_char> "+"
<opt_sign> ::= "" | "+" | "-"
<inf> ::= /[iI][nN][fF]/ | /[iI][nN][fF][iI][nN][iI][tT][yY]/
<nan> ::= /[nN][aA][nN]/
<number> ::= <main> <opt_exponent>
<json_parse_array> ::= <json_parse_array'> | "]"
<json_parse_object> ::= <json_parse_object'> | "}"
<any_char> ::= /[\x01-\xff]/
<main> ::= <digits> | <opt_digits> "." <digits>
<opt_exponent> ::= "" | /[eE]/ <opt_sign> <digits>
<digits> ::= <digit>+
<opt_digits> ::= "" | <digits>
<digit> ::= /[0-9]/
<json_parse_array'> ::= <json_parse_value> "," <json_parse_array'>
    | <json_parse_value> "]"
<json_parse_object'> ::=
      <json_parse_value> ":" <json_parse_value> "," <json_parse_object'>
    | <json_parse_value> ":" <json_parse_value> "}"
    | "}"
```

Figure 3.1: *harrydc-json* grammar mined by STALAGMITE

3.1 Introduction

Generating test inputs for computer programs continues to be one of the big challenges of software testing. The biggest challenge is that one needs inputs that are *valid* (such that they are not rejected by the parser) and at the same time *uncommon* (such that one can test corner cases not found in common inputs). Totally *random* inputs, for instance, would be uncommon, but also very likely invalid. *Sample* inputs collected or condensed from public sources (say, via an LLM), on the other hand, tend to be *valid*, but also *common*.

In order to obtain inputs that are valid *and* uncommon, the textbook solution is to *specify* the input language. Using a formal *grammar*, we can use the grammar as a *producer* and thus generate inputs that are syntactically *valid* in the first place. Figure 3.1, for instance, shows a grammar for the *harrydc-json* parser. Having this grammar has several benefits:

- Every input produced by this grammar is *valid*, i.e., accepted by *harrydc-json*. Using this grammar and a fast producer [53], we can easily produce millions of valid test inputs within minutes.
- The grammar covers the entire input language of *harrydc-json*, including every feature (including *uncommon* ones); using it as a producer will test every *harrydc-json* feature.
- We can use the grammar to detect *deviations* from standards. The string `{true: infinity,}` is accepted by the grammar in Figure 3.1 (and *harrydc-json*), but in multiple ways invalid according to the *json* standard.[1]
- We can use it to *parse* existing inputs, check their validity or extract input fragments.
- We can *mutate* existing inputs while preserving validity—all this with much higher efficiency than with random inputs or random lexical mutations, and with much higher language coverage than sample inputs.

All these benefits of grammars are well recognized in the testing community. The catch, however, is the widespread absence of formal specifications, including grammars. It thus is no surprise that recent research has focused on successfully *mining grammars* from code and/or input samples:

- *Autogram* [57] and *Mimid* [52] introduced the concept of *dynamic grammar mining*. Given a program `P` and a set of sample inputs, they dynamically track how `P` processes input characters, and form tokens and grammar structures from characters processed similarly.
- *Arvada* [64] learns grammars from a set of sample inputs, using a *parse oracle* to check which input parts can be replaced by others (and thus form a grammar abstraction).

The downside of these approaches, however, is that they are *biased* towards the set of sample inputs; if a language feature is not present in the samples already (like `NaN` in *json* samples), they cannot observe it and thus not incorporate it into the mined grammar. And of course, if no sample inputs are available, these approaches become entirely inapplicable, as there are neither executions to observe nor samples to mine grammars from.

How can we avoid this reliance on sample inputs? So far, the problem has been that *symbolic* approaches (which need no inputs) are effective in the small, but always have *failed* to analyze nontrivial parsers—the high number of branches, loops, and recursive

[1] In the *json* standard, (1) object keys (`true`) must be strings; (2) `infinity` is not a legal value; and (3) trailing commas are not allowed.

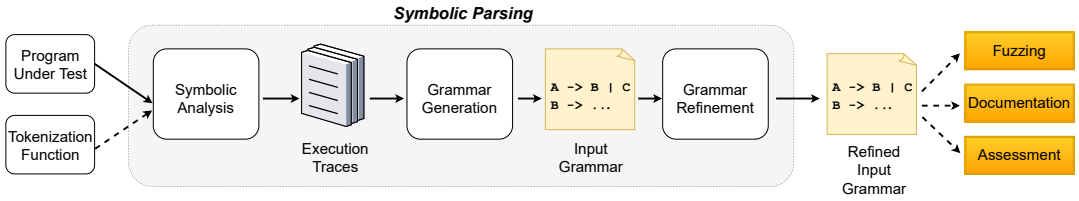


Figure 3.2: How STALAGMITE works. STALAGMITE infers context-free input grammars from recursive descent parsers. It starts by symbolically executing and tracing the program under test, comprehensively exploring execution paths by limiting loop iterations and recursive calls, while tracking *input consumptions* by the parser. Subsequently, these symbolic execution traces are converted into an input grammar by leveraging execution context information. Finally, this input grammar is refined to reduce overapproximation.

calls, all intertwined, results in a combinatorial explosion of paths that was impossible to manage.

In this chapter, we introduce STALAGMITE[2], the first scalable *fully automatic static grammar miner.* From a program under test with an input parser (Figure 3.2), STALAGMITE automatically extracts the input language as a grammar. The *json* input grammar in Figure 3.1, for instance, is automatically mined from *harrydc-json* by STALAGMITE.

This grammar

- is extracted from code alone, i.e. not requiring any input samples;
- has a *recall* of 100% (i.e., it covers the entire input language);
- has a *precision* of 100% (i.e., all strings produced are valid);
- can easily serve as *producer, parser,* and *mutator* for efficient fuzzing; and
- is sufficiently structured and readable for reverse engineering and documentation.

STALAGMITE takes as input a C program with a *recursive descent* input parser and produces its input grammar.[3] If the program under test uses a separate *lexer* to process characters into tokens, an additional *harness* of 5–10 lines is required to specify the information flow between lexer and parser. Other than that, STALAGMITE operates fully automatically and extracts the complete grammar within minutes to hours.

[2]STALAGMITE stands for static language and grammar mining for testing.

[3]The alternative *table-driven parsers* are generated from a grammar specification; for these, a formal grammar is there in the first place and does not need to be extracted. Recently, the CLANG and GCC compilers switched from table-driven parsers to recursive descent parsers to gain larger control over the parsing process.

The core idea behind STALAGMITE— symbolic parsing — was first presented in a short paper [24]. Our contributions extend the original proposal as follows:

1. We describe how we realized symbolic parsing, evolving the instrumentation-based function-level sketch from the short paper into a full system-level implementation built on top of the KLEE symbolic execution engine.

2. We detail our techniques for input consumption tracking and grammar refinement.

3. We conduct a comprehensive evaluation on nine real-world parsers, including the four previously analyzed.

To realize symbolic parsing (and hence static grammar mining), we tackle the path explosion problem in the context of recursive descent parsers by limiting the exploration of loops and recursion, facilitating a comprehensive symbolic exploration. During symbolic execution, we track *consumptions of input characters* and associate them to the current execution context (i.e., call path and loop iterations). This context information allows constructing *parse trees* of explored inputs. Subsequently, we generalize these parse trees to input grammars:

- Inner nodes of the parse trees (i.e., functions and loops) are converted to non-terminal symbols in the grammar. This is consistent with the implementation style of recursive descent parsers, which define a function for each non-terminal symbol.

- Bounded control structures (i.e., loop iterations and recursion) are generalized and again instantiated, allowing for unbounded repeated elements (loops) and nested elements (recursion) again.

- Lexical elements (i.e., tokens) are generalized by matching observations with predefined string classes (such as lower case strings or digits).

Note, though, that while this chapter demonstrates the feasibility of symbolic parsing in practice, it would be overly optimistic at this point to expect that one can place an arbitrary program into a black box and have the input grammar pop out. Complex input processors with multiple stages, as found in compilers, are very much out of reach for our implementation, as are binary input formats with features such as length encodings, offsets, or checksums that cannot be represented adequately in a context-free grammar. Hence, there are still years of research and engineering ahead until one can claim a one-stop solution—even if this work shows a new and very promising path towards such a solution.

In summary, we make the following contributions:

1. We introduce the *first approach to statically mine grammars from code* without requiring input samples.
2. We *detail* how we realized *symbolic parsing* in our STALAGMITE prototype.
3. We *evaluate* STALAGMITE and show that it produces highly accurate grammars with precision and recall of 99–100%.

The remainder of this chapter is organized as follows. **Section 3.2** introduces our symbolic execution-based technique for exploring parsers and producing execution traces. **Section 3.3** explains how we infer inputs grammars from these execution traces. **Section 3.4** provides implementation details of STALAGMITE. In **Section 3.5**, we evaluate STALAGMITE on a set of parsers, assessing the accuracy and utility of the mined grammars. Related work is discussed in **Section 3.6**, followed by limitations and future work in **Section 3.7**. We close with our conclusion and key takeaways in **Section 3.8**. The STALAGMITE tool and all experimental data are available as open source (**Section 3.9**).

3.2 Symbolically Exploring Parsers and Generating Execution Traces

In this section, we present the first step of symbolic parsing: tracing and bounding the symbolic execution of parsers. The objective of this step is to collect an execution trace for each symbolically executed control-flow path. Each trace maps input positions to execution contexts, consisting of the call path and loop iterations that were active during consumption of this input position, as well as all possible solutions. A sample execution trace is shown in Figure 3.3. To briefly anticipate the next step, which we detail in Section 3.3, these execution traces are later converted into parse trees (Figure 3.4), which are then aggregated and generalized to infer inputs grammars.

To ensure meaningful results, we only consider execution paths that lead to a successful parse. Determining whether an input was successfully parsed is straightforward, as parsers typically return a non-zero value or throw an exception to indicate an error. Throughout this section, we refer to Algorithm 1, showcasing how the individual components of STALAGMITE may be implemented in a symbolic execution engine.

3.2.1 Assumptions

We start by defining recursive descent parsers, as STALAGMITE focuses on mining grammars from such parsers.

#	Execution Context	Solutions
0	parse:value	`[`
1	parse:value:array:L1I1:value	`/[0-9]/`
2	parse:value:array:L1I1	`,`
3	parse:value:array:L1I2:value	`"`
4	parse:value:array:L1I2:value	`"`
5	parse:value:array:L1I2	`]`

Figure 3.3: Simplified execution trace for *harrydc-json*

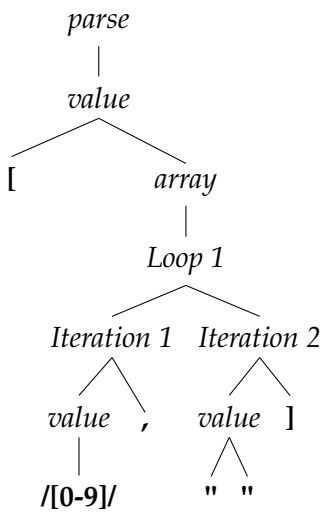


Figure 3.4: Parse tree derived from Figure 3.3

Definition 9: Recursive Descent Parser

A recursive descent parser consists of a set of *mutually recursive parse functions*. Each parse function corresponds to one non-terminal symbol in the grammar and recognizes the portion of the language that this symbol defines. To parse a non-terminal symbol `<B>` in a production rule of `<A>`, the parse function `A` calls parse function `B`. To recognize a terminal symbol `"b"` in a production rule of `<A>`, function `A` compares characters from the input stream [36].

We make the following assumptions about the program under test:

- The program under test is a *recursive descent parser*.
- The program under test incrementally processes and accepts characters from the input stream.
- The program under test only checks syntactic input features that can be expressed in a context-free grammar.
- The program under test can be compiled to LLVM IR (e.g., C and C++ programs).

3.2.2 Tracking Input Consumption

One of the key challenges in symbolic grammar inference is identifying the exact point in execution when the parser consumes each input character. This is essential because we utilize the execution context, which is active during the consumption of each input

character, to construct the input grammar. Specifically, we leverage the hierarchical information provided by the currently active call path and loop iterations.

Definition 10: Input Consumption

We define the *input consumption* of an input character as the point in the execution at which the parser recognizes the character and no longer needs to process it again for parsing purposes.

Our method for tracking input consumptions is similar to the one used by Mimid [52], but it differs in two significant ways. Before outlining these differences, we first provide an overview of the technique used by Mimid.

3.2.2.1 The Mimid Approach

Mimid is a dynamic input grammar mining technique that uses lightweight instrumentation to track how a parser consumes input characters of a given concrete input. Specifically, it monitors accesses only to the original input buffer, assuming that the parser neither modifies nor duplicates it. Since a parser may access a character multiple times before ultimately consuming it, Mimid attributes each character to the execution context of its last recorded access.

3.2.2.2 The STALAGMITE Approach

STALAGMITE differs from Mimid in two key ways when tracking input consumption. First, instead of monitoring only the original input buffer, we track accesses to all buffers that contain input characters using a lightweight data-flow tracking approach. Second, we do not only record the *last* access but *all* accesses to input characters. Subsequently, as we discuss in Section 3.3.1, we apply a heuristic to find the input accesses that actually correspond to input consumptions. In the following section, we detail these two aspects of our approach.

Lightweight Data-Flow Tracking

We observed that some parsers may copy portions of the input buffer into an other buffer during parsing. As a result, tracking accesses only to the original input buffer can lead to inaccuracies. For instance, consider the code snippet in Figure 3.5 from one of our evaluation subjects, *cjson*, where a substring of the input buffer is copied into an other buffer ① and subsequently processed further ②.

We address this issue using a lightweight data-flow tracking approach. Specifically, we *track accesses into direct copies of the input buffer*, but *we do not track accesses into buffers that contain* transformed *input characters*, such as `(input[0] - 0x30)` or `(input[0] + input[2])`. The latter would indicate an evaluation performed by parsers that do not

```
1  static cJSON_bool parse_number(cJSON * const item,
2                  parse_buffer * const input_buffer) {
3      /* ... */
4      for (i = 0; /* ... */; i++) {
5          switch (buffer_at_offset(input_buffer)[i])
6          {
7              case '0': case '1': case '2': case '3': case '4':
8              case '5': case '6': case '7': case '8': case '9':
9              case '+': case '-': case 'e': case 'E':
10                 number_c_string[i] = buffer_at_offset(input_buffer)[i]; ①
11                 break;
12             /* ... */
13             default: goto loop_end;
14         }
15     }
16 loop_end:
17     number_c_string[i] = '\0';
18     number = strtod((const char*)number_c_string, (char**)&after_end); ②
19     /* ... */
20 }
```

Figure 3.5: Snippet from *cjson* showing how it copies part of the input buffer to *number_c_string* during parsing

strictly separate parsing from interpretation, which we deliberately avoid tracking. In contrast, parsing code is expected to process unmodified input characters (but may very well process copies of them). For example, we treat the buffer `number_c_string` in Figure 3.5 as a *copy* of the input buffer, and track accesses to it. This is straightforward to implement in a symbolic execution based technique like STALAGMITE, as the symbolic store already keeps track of the symbolic input. More precisely, this is implemented in Lines 18—19 of Algorithm 1.

Recording All Input Accesses

We record all accesses to the input buffer, labeling each access with an incremental number that we call the *access order*, and store this along with the execution context in the trace (Lines 20-21 of Algorithm 1). Once the parser under test has terminated successfully, we query the solver of the symbolic execution engine for all possible solutions for each input character, and finally output the trace. This is implemented in Lines 23—25 of Algorithm 1.

3.2.3 Limiting Loops

Next, we describe our technique to bound loops in parsers. The purpose is to enable comprehensive symbolic exploration by mitigating path explosion. Before symbolic execution starts, we retrieve the header, exiting, and exit basic blocks of each loop in

Algorithm 1 STALAGMITE core modifications to interpretation-based symbolic execution engines

```
1:  function symbolicInterpreter(ExecutionState es)
2:    while running do
3:      switch event
4:        ...
5:        case Call from call site cs to function f:          ▷ Limiting Recursion
6:          es.stackFrames.push((cs, f, loopIterationStack=stack()))
7:          // Analogously, on return: pop from es.stackFrames
8:          if es.stackFrames.count((cs, f)) > 3 then
9:            terminateState(es)
10:         end if
11:         ...
12:       case Enter loop header h:                           ▷ Limiting Loops
13:         es.stackFrames.last().loopIterationStack.push(h)
14:         // Analogously, on loop exit: pop from loopIterationStack
15:         if es.stackFrames.last().loopIterationStack.count(h) > 4 then
16:           terminateState(es)
17:         end if
18:         ...
19:       case Load from address a:                           ▷ Tracking Input Consumption
20:         symExpr ← load(a)
21:         if symExpr == (Read symbolicInputBuffer inpPos) then
22:           es.trace[inpPos]['accessOrders'].push(es.accessOrder++)
23:           es.trace[inpPos]['executionContext'].push(es.stackFrames.serialize())
24:         end if
25:         ...
26:   end while
27:   for inpPos in es.trace.keys() do
28:     es.trace[inpPos]['solutions'] ← solve(es, (Read symbolicInputBuffer inpPos))
29:   end for
30:   serializeTrace(es.trace)
31: end function
```

the program. During symbolic execution, we first mark loops as *input-processing* if they access multiple input characters. Loops are assigned the *input-processing* attribute only for the current execution context, allowing loops to take different roles, e.g., for different calling contexts. We effectively limit the exploration of *input-processing* loops to at most four iterations. This number was chosen based on our observations of the parsers we evaluated. We do so by counting control flow transitions to the header basic block of each loop, and terminate the current execution state when a loop would be entered for the fourth time. This is implemented in Lines 11—15 of Algorithm 1.

3.2.4 Limiting Recursion

Similar to how we limit loops, we also bound recursive calls to a recursion depth of three. We track the call path during symbolic execution as a vector of *(caller, callee)* pairs within the symbolic execution state. When a function is called, we check if the current *(caller, callee)* pair appears more than three time in the call path. If it does, we terminate the current execution state, effectively bounding recursive calls. Specifically, this is implemented in Lines 5—9 of Algorithm 1. The number *three* was chosen based on our observations of the parsers we evaluated. In principle, this number could be increased if necessary, which would also increase the number of paths explored and thus the complexity.

3.2.5 Handling Parsers With a Lexing Stage

In ad-hoc tokenization, which is assumed in Section 3.2.2, parse functions only check for one specific token at a time. However, some parser implementations follow a different methodology and contain a single dedicated tokenization function, which is called by all parse functions and which may read *any* of the possible tokens. Symbolic execution of such parsers is challenging [81] because the tokenization function is called multiple times during symbolic execution and often contains many branches and loops. We found that this circumstance routinely overburdens symbolic execution.

To address this problem, STALAGMITE analyzes the tokenization function only once in isolation, storing pairs of *token instance* (e.g., "123") and associated *token identifier* (e.g., a numerical value representing an `INT` token) that was returned by the tokenization function, in a token grammar. When this tokenization function is called during the symbolic execution of the parser, we detour this call to a *proxy function* that simply returns a *symbolic token identifier* without processing the input. This drastically reduces the burden on the symbolic execution engine.

Some manual work is required to implement (1) the *harness* to symbolically execute the tokenization function *in isolation*; and (2) the *proxy function*, which serves as *glue code* when analyzing the whole parser. The harness needs to mark the input variable symbolic, call the tokenization function, check parsing success, and pass the token

identifier to the symbolic execution engine; this work is easy to do for a developer, requiring less than 10 lines of code. Automating this task is difficult, however, as the interface between the parser and the tokenization function is not standardized. In fact, tokenization functions may return the token identifier in many different ways, such as via a global variable, a return value, or an output parameter. Similarly, the tokenization function may read the input in different ways.

Figure 3.6 shows the harness for the tokenization function *next_sym()* of *Tiny-C*, one of our evaluation subjects. It sets up *inp* as a symbolic input, and writes it to the global *cursor* variable used by *next_sym()*, which is then called. Finally, it passes the token identifier *sym* to the symbolic execution engine, which logs it as part of the trace. Figure 3.7 shows the proxy function for *next_sym()*, which is used instead during parser analysis. It simply assigns a symbolic token identifier to the global variable *sym* holding the current token identifier.

For parsers with a lexing stage, we use the same technique to track input consumptions as we do for ad-hoc tokenization, except that we record accesses to symbolic token identifiers instead of input characters.

```
void wrapper_next_sym() {
  char* inp = stalagmite_sym_input();
  cursor = &inp;
  next_sym();
  stalagmite_solve_and_outp_token(sym);
}
```

Figure 3.6: Harness for *Tiny-C*'s tokenizer *next_sym()*

```
void proxy_next_sym() {
  sym = stalagmite_next_sym_token();
}
```

Figure 3.7: Proxy tokenization function for *Tiny-C*

3.3 Inferring the Input Grammar From Execution Traces

At this point, symbolic execution has completed, producing multiple execution traces (e.g., Figure 3.3), each representing a distinct control-flow path. Additionally, for parsers with a lexing stage, the tokenization function analysis has generated pairs of *token identifiers* and *token instances*. Next, to infer the input grammar, we proceed with the two steps briefly outlined below, which we discuss in more detail in the following sections.

Converting Traces into a Grammar. We first convert execution traces into parse trees, leveraging the execution context of input consumptions to label tree nodes. Subsequently, we merge these parse trees and convert them into a single grammar. Finally, we reinstantiate and generalize loops in this grammar.

Generalizing Tokens in the Grammar. We process each token identifier and associated token instances. If all token instances are identical, the token is likely a *keyword* or *operator*. Hence, we do not process it any further. However, if many different token instances are associated to the same token identifier, we generalize them into one of our predefined string sets that fits best.

3.3.1 Converting Traces to an Input Grammar

Constructing the input grammar requires the execution contexts of input consumptions. Up to this point, we have recorded all accesses to input characters, and we now identify which of these are input consumptions.

Identifying Input Consumptions in Input Accesses

In most cases, it is reasonable to assume that the last access to an input character corresponds to its point of consumption, following the approach of Mimid. However, we observed instances where the last access does not align with the actual point of consumption. For example, consider the function in Figure 3.8 from our *calc* evaluation subject.

This function handles the addition and subtraction of two expressions. From an input consumption tracking perspective, the key detail is that the current input character is first copied into the buffer `op` ①. It is then consumed ②, assuming it is either `"+"` or `"-"` (otherwise, the parse would be invalid). The actual consumption is evident when the input pointer is incremented ③. However, the same input character is accessed again ④ because *calc* evaluates the arithmetic expression as part of parsing. As a result, relying on the last access as the point of consumption would produce incorrect results.

To address this issue, we propose a general heuristic to identify input consumptions, aiming to map each input character to its correct point of consumption. First, as detailed in Section 3.2.2.2, we record *all accesses to each input character* during symbolic execution, including their execution context, and assign incremental labels to track order. Next, we apply our heuristic to the recorded accesses to determine the point of consumption for each input character.

As an example, Figure 3.9 shows an execution trace from the *calc* subject on input `0+0`. Notably, the character at position 1 was subsequently accessed ten times (labeled 6 to 15). However, before its final access (labeled 26), characters at position 2 and 3 were accessed. Simply using the last accesses as the point of consumption (i.e., at labels 5, 26, 21, 28) would lead to an incorrect result. Specifically, the access labeled 26 would cause the "+" character to be mapped to ④ instead of ② in Figure 3.8.

To resolve this issue, we apply Algorithm 2 to identify input consumptions. The goal of this algorithm is to select and output one *access order* for each input character such

```c
static int parse_sum(const char **pexp, char expect, int *value) {
  int rh, ret;
  ret = parse_mult(pexp, expect, value);
  if (!ret)
    while (**pexp != expect) {
      char op = **pexp; ①
      ret = ERR_TOK;
      if (op != '+' && op != '-') ②
        break;
      (*pexp)++;  ③
      ret = parse_mult(pexp, expect, &rh);
      if (ret)
        break;
      switch (op) { ④
        case '+':
          *value += rh;
          break;
        case '-':
          *value -= rh;
          break;
      }
    }
  return ret;
}
```

Figure 3.8: Function of *calc* that combines parsing and evaluation

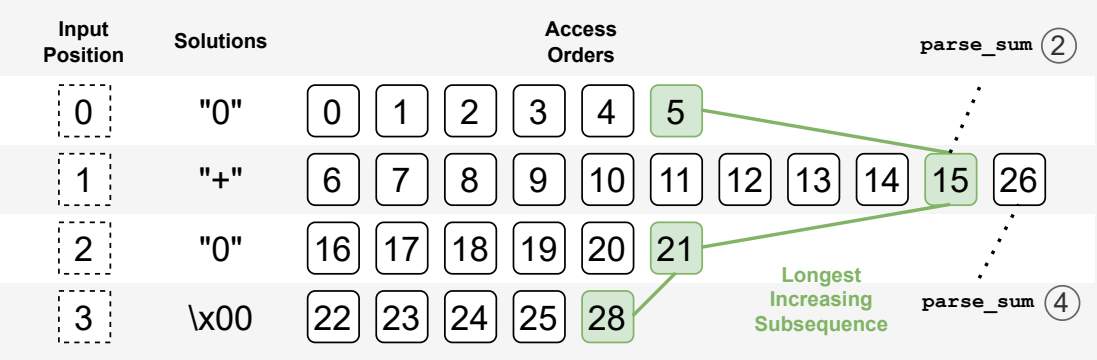


Figure 3.9: Sample execution trace illustrating how input consumptions are identified. Execution contexts are omitted for brevity.

that the selected access orders are non-decreasing. This algorithm starts by initializing the `resultAccessOrders` list with the last access order of each input character (Line 10), which, for the running example, is `[5,26,21,28]`. It then calculates the longest increasing subsequence (LIS) of these access orders, which, in the running example, is `[5,21,28]`. Since the LIS is shorter than the number of input characters (Line 12), the algorithm identifies the first outlier — the access order corresponding to the first input character not part of the LIS — and backtracks to the previous access of that character (Line 15). In the example, the algorithm identifies `26` as the outlier and pops it from the `traces[1]['accessOrders']` list of the second input character. In the next iteration, the algorithm computes the LIS of `[5,15,21,28]`, finds that its length matches the number of input characters, and thus returns *resultAccessOrders*.

Until now, we have not discussed Line 5—9 of Algorithm 2. This part of the algorithm serves as a fallback to handle situations where backtracking on individual character accesses does not yield a maximum increasing subsequence. In such cases, the algorithm assigns the access orders of the previous input character to the current input character. This approach guarantees an increase in the length of the LIS (since we allow duplicates), facilitating convergence towards a solution.

Algorithm 2 Identifying Input Consumptions

```
1:  function identifyInputConsumptions(traces)
2:    while True do
3:      resultAccessOrders ← []
4:      for inpPos ← 0 to length(traces) − 1 do
5:        if traces[inpPos]['accessOrders'] = [] then     ▷ Backtrack was not successful
6:          if inpPos = 0 then traces[inpPos]['accessOrders'] ← [0]
7:          else traces[inpPos]['accessOrders'] ← traces[inpPos − 1]['accessOrders']
8:          end if
9:        end if
10:       resultAccessOrders.append(traces[inpPos]['accessOrders'][−1])
11:     end for
12:     lis ← longestIncreasingSubsequence(resultAccessOrders)
13:     if length(lis) = length(resultAccessOrders) then return resultAccessOrders
14:     end if
15:     outlier ← {o ∈ resultAccessOrders | o ∉ lis}[0]
16:     outlierPos ← accessOrderToInpPos(outlier)
17:     traces[outlierPos]['accessOrders'].pop()          ▷ Backtrack to previous access
18:   end while
19: end function
```

Deriving the Grammar

Algorithm 3 shows our approach for deriving the input grammar from execution traces. The first step is converting each execution trace into a parse tree. Each execution context in the execution trace corresponds to a path in the parse tree. The parse tree is constructed by inserting paths in the order of input indices. The leaf node of each path are the terminal solutions. To illustrate this, Figure 3.4 presents the parse tree for the execution trace shown in Figure 3.3.

Next, all parse trees are converted into a grammar by performing the following steps:

1. Each node is converted into a non-terminal symbol that encodes the calling context. For instance, calls from two different call sites to the function `value` will result in the non-terminal symbols **<value>** and **<value'>**. All children of the node are added as rule alternatives to the definition of that node in the grammar.

2. Loops are generalized by introducing *continuing* and *exiting* iterations. The last iteration is considered *exiting* (because it leaves the loop), whereas other iterations are *continuing*, i.e., they jump back to the loop header.

3. Terminal symbols are generalized as described in Section 3.3.2.

```
<start>   ::= <parse>
<parse>   ::= <value>
<value>   ::= "[" <array>
<array>   ::= <L1>
<L1>      ::= <L1_cont> <L1>
            | <L1_exit>
<L1_cont> ::= /[0-9]/ ","
<L1_exit> ::= "\"\"]"
```

Figure 3.10: Input grammar extracted from the execution trace in Figure 3.3 (simplified)

Figure 3.10 shows the grammar for the parse tree in Figure 3.4. This grammar, which was derived from a single execution trace, can already produce inputs such as `[1,5,8,""]` and `[2,""]`. The grammar is completed step by step by processing all execution traces, and hence, parse trees, finally resulting in the grammar shown in Figure 3.1.

3.3.2 Generalizing Tokens in the Input Grammar

The purpose of this step is to generalize tokens in the input grammar. For parsers with a lexing stage, we mined a separate *token grammar*, which maps token identifiers to associated token instances, as discussed in Section 3.2.5. In this case, we generalize token instances for each token identifier. For parsers with ad-hoc lexing, the token grammar is embedded into the grammar mined in the previous step. In this case,

Algorithm 3 Converting traces to an input grammar. Calling contexts omitted for brevity.

```
 1: function treePath(executionContext)
 2:     // E.g., this function converts "parse:value:array:L1I1" to a path of four nodes:
 3:     // InnerNode("parse")→InnerNode("value")→InnerNode("array")→LoopNode("L1I1")
 4: end function
 5: function leafNode(solutions)
 6:     // Returns a LeafNode containing all solutions
 7: end function
 8: function treeToGrammar(parseTree)
 9:     grammar ← Grammar()
10:     for Node N in parseTree do
11:         switch N.type
12:             case LeafNode:
13:                 grammar[N.name].addRule([s] for s in N.solutions)
14:             case InnerNode:
15:                 rule ← [C.name for C in N.children]
16:                 grammar[N.name].addRule(rule)
17:             case LoopNode:                                  ▷ Loop Generalization
18:                 grammar["L" + N.name].addRule(["L" + N.name + "_cont", "L" + N.name])
19:                 grammar["L" + N.name].addRule(["L" + N.name + "_exit"])
20:                 rule ← [C.name for C in N.children]
21:                 if isLastIteration(N) then
22:                     grammar["L" + N.name + "_exit"].addRule(rule)
23:                 else
24:                     grammar["L" + N.name + "_cont"].addRule(rule)
25:                 end if
26:     end for
27:     return grammar
28: end function
29: function traceToTree(trace)
30:     parseTree ← Tree()
31:     consumingAccessOrders ← identifyInputConsumptions(trace)        ▷ Section 3.3.1
32:     for inpPos ← 0 to length(traces) − 1 do
33:         consumingAccessOrder ← consumingAccessOrders[inpPos]
34:         consumingExecutionContext ← accessOrderToExecCtx(traces, consumingAccessOrder)
35:         path ← treePath(consumingExecutionContext)
36:         leaf ← leafNode(traces[inpPos]['solutions'])
37:         parseTree.insert(path + leaf)
38:     end for
39: end function
40: function tracesToGrammar(traces)
41:     grammar ← Grammar()
42:     for trace in traces do
43:         grammar.union(treeToGrammar(traceToTree(trace)))            ▷ Section 3.3.1
44:     end for
45:     return generalizeTerminals(grammar)                             ▷ Section 3.3.2
46: end function
```

we only generalize token instances associated with non-terminal symbols of external functions (such as `strtod`). The reason is that external functions may not adhere to our assumptions listed in Section 3.2.1. For other non-terminal symbols, the token structure is already clear from the code structure.

```
<INT>  ::= "9017" | "2" | "118" | ...
<ID>   ::= "m" | "s" | "u" | "a" | ...
<LPAR> ::= "("
<WHILE_SYM> ::= "while"
```

Figure 3.11: Token instances aggregated by token ID

```
<INT>       ::= <digit>+
<ID>        ::= <lower_char>
<LPAR>      ::= "("
<WHILE_SYM> ::= "while"
```

Figure 3.12: Generalized token instances of Figure 3.11

In the following, we detail our token generalization technique, which is identical for both types of lexers. Figure 3.11 shows an excerpt of the token identifiers and their associated token instances for the *Tiny-C* subject. We generalize these token instances to sets of characters by matching them with predefined patterns of string classes such as digits, hexadecimal digits, floating point numbers, lowercase strings, and so on. Table 3.13 shows an excerpt of these predefined string classes. The patterns we try to match get increasingly more permissive, and we choose the least permissive set that contains all observed token instances. Moreover, we check if tokens may start with strings of whitespaces and generalize this accordingly. If at most three token instances are observed per token identifier, we consider this a *keyword or operator token*, which we do not generalize but instead include in the grammar as-is. For instance, in the example above, we do not generalize **<WHILE_SYM>** (as it is a keyword) but we do generalize **<INT>**. The result is a *generalized token grammar*, illustrated in Figure 3.12.

Table 3.13: String classes (excerpt)

Regular Expression	Non-Terminal
`\d`	**<digit>**
`[a-z]`	**<lower_char>**
`[0-9a-fA-F]`	**<hexdigit>**
`[-+]?((\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?)`	**<float_simple>**

3.3.3 Simplification

At this point, the grammar is operational. To improve its readability, we iteratively apply two transformations until reaching a fixed point:

Inlining. We inline production rules with a single rule alternative in the referencing context, resulting in a smaller and more readable grammar.

Optional generalization. If two rule alternatives become identical after adding a single non-terminal symbol at an arbitrary position in one of them, we merge them. As an example, consider the initial and resulting grammar in Figure 3.14, concisely modeling optional whitespaces.

Initial grammar

```
<statement> ::= <expr> <WS> ";"
              | <expr>      ";"
```

Resulting grammar

```
<statement> ::= <expr> <WS>? ";"
```

Figure 3.14: Optional non-terminal generalization

3.3.4 Reducing Overapproximation

When converting traces into a grammar, we duplicate non-terminal symbols of functions based on their call site. That is, if a function F is called from two call sites across all execution traces, the grammar would contain two definitions `<F>` and `<F′>`. However, this context is somewhat coarse; a function may parse different strings based on interprocedural constraints that have been passed along a long chain of calls. We differentiate functions only by their immediate call site, prioritizing completeness of grammars by accepting potential overapproximations.

We address this problem in a general way after mining the initial input grammar by refining overapproximating grammar rules using a dynamic technique. Note that this step is optional and can be omitted if the overapproximation is acceptable. We start by generating a set of inputs (e.g., 1000) from the mined input grammar using a grammar-based fuzzer. Next, we run these inputs in the parser under test and classify them based on the return value as passing (syntactically valid) or failing (syntactically invalid). For each of the failing inputs, we try to make it pass by applying a minimal syntactic change. Consider the failing input `5 = 1;` of *Tiny-C*. The input fails because the left-hand side of the assignment is a number. This assignment is derived from the grammar rule:

```
<expr> ::= <test> "=" <expr>
```

The derivation tree of the left-hand side of the assignment is:

```
<test>-<sum>-<term>-<INT>-"5"
```

To fix this input — and the grammar — automatically, we perform a post-order traversal of the derivation tree of the failing input (with regards to the initially mined grammar), and repeatedly replace each inner node with a randomly generated subtree. If replacing this subtree makes the input syntactically valid for the parser under test, we have a refinement candidate. Let us assume the refinement candidate `a = 1;` was generated. The derivation of the left-hand side is:

```
<test>-<sum>-<term>-<ID>-"a"
```

Next, we try to fix the grammar in the most general way, capturing the essence of what differentiates the valid from the invalid input. We do so by restricting the rule alternatives of the non-terminal symbol corresponding to the root of the replaced subtree with the rule chosen in it. In this example, the fix would be to update the definition of `<term>` in the grammar:

```
<term> ::= <ID>
```

To validate this fix, we check whether it causes *underapproximation*. We parse a set of pre-generated syntactically valid inputs with the updated grammar. If all these inputs can be parsed, we update the grammar permanently and move on to the next syntactically invalid input, if any.

Otherwise (which is the case here, as no `<INT>` could be generated anymore regardless of the context) we discard the grammar update. Next, we propagate the rule of the root of the current subtree to its parent node, in order to check if applying the update in the parent context results in a non-underapproximating grammar. In the example, the next two steps are to refine the rule alternatives of `<sum>`, and `<test>`. However, both refinements also lead to underapproximating grammars. Finally, after going up one more level, refining the rule alternatives of `<expr>` does not cause underapproximation:

```
<expr> ::= <ID> "=" <expr> | <test>
```

In general, in case no non-underapproximating grammar update can be found, we continue with the next failing input. Given a sufficiently large and diverse set of samples generated during grammar refinement, the result is a refined grammar, which is at least as precise as the initially mined grammar, and has the same recall.

3.4 Implementation

Our implementation is based on the symbolic execution engine KLEE [28] which operates on LLVM IR [65]. We extended KLEE both with static analyses as well as dynamic monitoring code. In total, we added 1190 lines of C++ code to KLEE. On the one hand,

we use compiler passes provided by the LLVM compiler infrastructure before starting symbolic execution:

- We run the *UnifyLoopExits* [115] transformation pass of LLVM, which ensures that loops have a single exit basic block, facilitating our subsequent analysis.
- We use the *LoopInfoWrapperPass* [71] to analyze loops in the program and retrieve their header, exiting, and exit basic blocks.
- We specify the *–switch-type=simple* option to KLEE in order to lower switch instructions to branch instructions.

On the other hand, we modified KLEE to monitor the symbolic execution in order to (1) log input accesses; (2) limit loop iterations and recursive calls; (3) output all possible solutions for each input byte; and (4) enable composite analysis of parsers with a dedicated tokenization function. In addition, we implemented 1210 lines of Python code to infer the input grammar from execution traces output by KLEE.

3.5 Evaluation

STALAGMITE is the first method capable of inferring input grammars exclusively from the code of parsers. In contrast, all existing grammar mining techniques [52, 64, 5, 67, 57] require an initial set of sample inputs to guide the learning process. Consequently, if no sample inputs are available, these prior approaches are unable to learn an expressive grammar. However, in practice, it is often possible to generate at least some sample inputs — even if they do not cover all language features. Therefore, to enable a fair comparison with state-of-the-art grammar mining techniques, we generate initial sets of sample inputs using symbolic execution, as discussed below.

We conduct a comprehensive evaluation of STALAGMITE, addressing the research questions:

RQ1: Grammar Accuracy. How accurate are the mined grammars, and how do they compare to those mined by state-of-the-art techniques?

RQ2: Grammar Conciseness. How concise are the mined grammars?

RQ3: Scaffolding Effort. How much manual effort is needed to set up parsers with a lexing stage?

RQ4: Time Consumption. How long does it take to mine the grammars?

RQ5: Bugs Found. Do discrepancies exist between parser implementations and their corresponding language standards?

We evaluate STALAGMITE on a set of parsers that cover a wide range of languages, summarized in Table 3.15.

Table 3.15: Evaluation subjects

Subject	Source	Description	Chomsky Hierarchy	Programming Language
`TINY-C`	[112]	*Parser for a subset of C.*	context-free	C
`HARRYDC-JSON`	[59]	*Parser for JSON.*	context-free	C
`CJSON`	[60]	*Parser for JSON.*	context-free	C
`PARSON`	[82]	*Parser for JSON.*	context-free	C
`MJS`	[76]	*Parser for a subset of JavaScript.*	context-free	C
`LISP`	[70]	*Parser for LISP.*	context-free	C
`CALC`	[6]	*Parser for arithmetic expressions.*	context-free	C
`CALCCPP`	[29]	*Parser for arithmetic expressions.*	context-free	C++
`CGI-DECODE`	[30]	*Decoder for CGI.*	regular	C

3.5.1 RQ1: Grammar Accuracy

Precision and recall are the core metrics to assess the quality of an inferred grammar. Intuitively, a precision of 100% implies that all inputs that were generated from the mined grammar are accepted by the parser under test, i.e., there are no false positives. In contrast, a recall of 100% means that all inputs generated from the language we want to learn can be parsed by the mined grammar, i.e., there are no false negatives. Consequently, computing recall requires a *golden grammar* that serves as a ground truth for the language we want to learn, as we cannot directly generate inputs from the parser under test. Both metrics are combined in the F1 score, which is the harmonic mean between precision and recall. These are all established metrics in the field of input grammar mining [52, 64, 5, 67]. Next, we provide a detailed explanation of how we compute these evaluation metrics, followed by the presentation of the results and a discussion.

To calculate precision, we proceed as follows:

1. We generate 1000 unique inputs from the inferred grammar using a grammar fuzzer [124].
2. We check how many of these inputs are accepted by the parser under test as valid inputs.
3. We output the precision as the percentage of valid inputs among the 1000 total inputs.

As a prerequisite for calculating recall, we define a *golden grammar* for each parser under test, which serves as the ground truth for the language we aim to learn. Specifically, as we share the evaluation subjects *harrydc-json*, *mjs*, *Tiny-C* and *cgi_decode* with

Mimid [52], we reuse the golden grammars provided by the reproducibility package of Mimid. Likewise, for *cjson* and *parson*, we reuse the golden grammar of the *harrydc-json* subject. We carefully constructed the golden grammar used for *calc* and *calccpp* by hand. For *lisp*, we took the golden grammar from the ANTLRv4 git repository [50]. To compute recall, we apply the following steps:

1. We generate 1000 unique inputs from the golden grammar using a grammar fuzzer [124], retaining only the inputs that are accepted by the parser under test.
2. We check how many of these inputs can be parsed by the inferred grammar.
3. We output the recall as the percentage of parsable inputs among the 1000 total inputs.

We complement this research question by also investigating how accurate the grammars mined by state-of-the-art techniques are. We consider Kedavra to be the state-of-the-art technique in black-box grammar mining, and Mimid to be the state-of-the-art white-box technique. Both were recently released, provide reference implementations and rely on sample inputs. In their original evaluations, Mimid used sample inputs generated from the golden grammars, whereas Kedavra reused a set of carefully constructed, short sample inputs from Arvada, covering all language features. In practice, these requirements may be hard to attain, as the golden grammar is typically not available in the first place to generate sample inputs from.

To enable a fair comparison with these techniques, we generate sample inputs using KLEE as follows:

1. We run KLEE with a 24-hour time budget on the program under test, and output all test cases.
2. We preprocess test cases to inputs amenable for input grammar miners by extracting the raw data using KLEE's `ktest-tool` and truncating inputs at the first NULL byte.
3. We then filter all valid inputs that can be parsed successfully by the program under test.
4. We randomly select ten sets of $100, 200, ..., 1000$ inputs from the set of all valid inputs, such that larger sets always include all inputs from smaller sets.

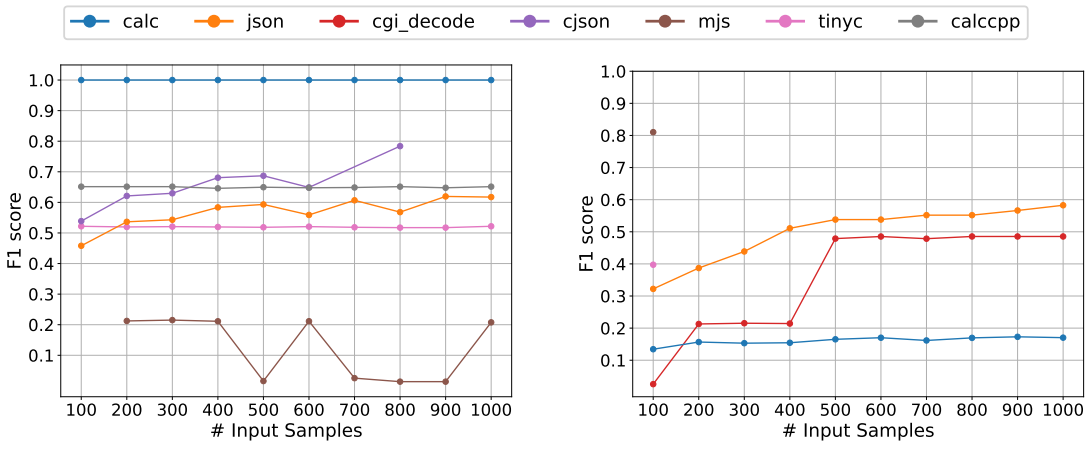
Subsequently, we run the respective grammar miner on each set of sample inputs to produce input grammars. For Kedavra, we found that its lexer does not support newline characters, and hence we additionally removed those characters during the preprocessing step. Moreover, Kedavra checks the program under test for whitespace insensitivity. If it is insensitive to whitespaces, Kedavra will not include them in the grammar. As this may put Kedavra at a disadvantage during recall assessment,

we filter whitespaces from inputs generated from the golden grammar during recall computation, which is favorable for Kedavra.

As ours is a white-box technique, one might wonder how much code of the program under test can be covered using inputs generated from the grammar, and whether this coverage could serve as a proxy for grammar accuracy. However, we believe that code coverage measurements are not meaningful in this context. This is because the target programs in this domain are typically isolated parsing components. Parsers often contain extensive syntax error-handling code, which equally contributes to code coverage, and which is specifically covered when using an incorrect grammar. As a result, input grammar miners producing incorrect grammars can appear to achieve high code coverage simply by exercising this error-handling code. To avoid drawing misleading conclusions about the effectiveness of an approach, we therefore opted not to include this experiment in our study.

Table 3.16: Grammar accuracy

	Stalagmite (Initial)			Stalagmite (Refined)			Kedavra			Mimid		
Subject	**precision**	**recall**	**F1**	**precision**	**recall**	**F1**	**precision**	**recall**	**F1**	**precision**	**recall**	**F1**
TINY-C	15.7%	100.0%	27.1%	99.6%	100.0%	99.8%	100.0%	35.3%	52.2%	100.0%	24.8%	39.7%
LISP	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	N/A	N/A	N/A	N/A	N/A	N/A
MJS	52.2%	100.0%	68.6%	65.0%	84.4%	73.4%	66.6%	12.3%	20.8%	75.0%	88.1%	81.0%
HARRYDC-JSON	88.4%	100.0%	93.8%	100.0%	100.0%	100.0%	97.8%	45.1%	61.7%	100.0%	41.1%	58.3%
CJSON	87.9%	100.0%	93.6%	99.2%	100.0%	99.6%	96.7%	65.9%	78.4%	N/A	N/A	N/A
PARSON	76.8%	78.5%	77.6%	87.1%	81.2%	84.0%	N/A	N/A	N/A	N/A	N/A	N/A
CALC	23.3%	100.0%	37.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	68.3%	9.9%	17.3%
CALCCPP	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	48.3%	65.1%	N/A	N/A	N/A
CGI-DECODE	99.9%	100.0%	99.9%	99.9%	100.0%	99.9%	N/A	N/A	N/A	87.4%	33.6%	48.5%



(a) Kedavra (b) Mimid

Figure 3.17: Grammar accuracy vs. number of input samples

Discussion

The results of this measurement are shown in Table 3.16. We first discuss the grammars mined by STALAGMITE. The initially mined grammars consistently have a recall of 100% with the exception of *parson*, which reaches a recall of 78.5%. The precision of initially mined grammars ranges from 15.7% (*Tiny-C*) to 100% (*calccpp*,*lisp*). After the dynamic grammar refinement step, the precision of six out of nine subjects rises, while three subjects remain at a precision of over 99.9%. For seven out of nine subjects, precision and recall are over 99% after refinement, indicating that the mined grammars are highly accurate.

For two subjects, *mjs* and *parson*, we observe non-optimal results. *mjs* is the most complex parser in our evaluation, consisting of more than 30 parse functions and over 16.000 lines of code. Because of this large code size, we cannot rule out that *mjs* does not violate some of our assumptions (Section 3.2.1). Moreover, the drop in recall to 84.4% for *mjs* indicates that the grammar refinement step incorrectly accepted an *underapproximating* grammar update. This could have been avoided by generating a larger set of valid inputs from the initially mined grammar during this step. For *parson*, we investigated the root cause and identified that the code violates our assumption of incremental processing. Specifically, in function `is_decimal`, a loop iterates over a substring of the input in reverse direction, which is not supported by our approach.

Although the results for *mjs* and *parson* are not perfect, we argue that they are still valuable and effective for fuzzing applications. Our results demonstrate that STALAGMITE can learn input languages even from highly complex parsers without requiring any sample inputs.

For Kedavra and Mimid, we also analyze how grammar accuracy depends on the number of sample inputs in Figure 3.17. Neither tool was able to produce input grammars for all nine subjects. Mimid imposes strict requirements on the source code of the program under test [43], such as no gotos, macros, or enumerations, specific formatting for case statements, and that the entire code must be contained in a single source code file. We modified the *calc* source code to meet the requirements of Mimid but were unable to do so for *lisp*, *cjson*, and *parson*, which prevented us from running Mimid successfully on these subjects. Additionally, Mimid only supports C and Python programs, so *calccpp* is not compatible. As a black-box tool, Kedavra can in principle be applied to any program. However, within our 24-hour time budget for this experiment, it failed to produce grammars for *cgi_decode*, *lisp*, and *parson*. We suspect this is because these subjects do not fully comply with Kedavra's tokenization rules, causing the inference process to not terminate within a reasonable amount of time. Surprisingly, Kedavra did not generate a grammar in time for *mjs* with 100 sample inputs, but did so for larger sizes. Moreover, it did not terminate on *cjson* for sample sizes larger than 800. Similarly, within the 24-hour budget, Mimid was only able to infer grammars for *mjs* and *Tiny-C* for 100 sample inputs.

We observe that larger sets of sample inputs moderately improve grammar accuracy for some subjects, but never reach STALAGMITE's level—except for Kedavra on the simple *calc* subject. As expected, both sample-based approaches achieve high precision (as they were given valid sample inputs), but perform poorly in recall. In summary, STALAGMITE is the only approach that consistently mines highly accurate grammars from source code alone.

STALAGMITE grammars are highly accurate,
with seven out of nine subjects reaching a precision and recall of over 99%.

3.5.2 RQ2: Grammar Conciseness

In addition to being useful for testing, the ability of STALAGMITE to infer an input grammar may also serve as a reverse engineering aid. As such, conciseness and readability are crucial for engineers working with the inferred grammar. Consistent with recent related work [5, 67], we measure conciseness by means of quantitative grammar metrics, shown in Table 3.18. The measured quantities are the number of unique non-terminal symbols (*NT*), the number of rule alternatives (*RA*), the average rule length *l(RA)*, and the sum of all rule lengths *S*. Readability, on the other hand, is not merely a quantitative metric. Evaluating it would require a user study, which is beyond the scope of this study. Nevertheless, STALAGMITE improves readability by extracting function names from the parser to label grammar non-terminals. This information is not available to black-box approaches, which label non-terminals using only sequential integers.

Table 3.18: Grammar conciseness statistics

	Initial				Refined				Golden			
Subject	**NT[1]**	**RA[2]**	**l(RA)[3]**	**S[4]**	**NT[1]**	**RA[2]**	**l(RA)[3]**	**S[4]**	**NT[1]**	**RA[2]**	**l(RA)[3]**	**S[4]**
TINY-C	32	119	3.39	403	43	136	3.28	446	12	59	1.46	86
LISP	14	287	1.07	306	14	287	1.07	306	33	80	1.25	100
MJS	74	856	1.9	1627	74	473	1.8	852	608	1856	1.28	2375
HARRYDC-JSON	51	663	1.15	765	41	389	1.26	491	30	162	1.19	192
CJSON	45	687	1.28	880	50	691	1.28	887	30	162	1.19	192
PARSON	48	2212	1.15	2540	58	2459	1.29	3182	30	162	1.19	192
CALC	22	54	1.65	89	21	50	1.6	80	8	27	1.63	44
CALCCPP	32	82	2.01	165	32	82	2.01	165	8	27	1.63	44
CGI-DECODE	5	281	1.01	283	5	281	1.01	283	5	99	1.02	101

[1]NT = unique non-terminals, [2]RA = rule alternatives, [3]l(RA) = average rule length, [4]S = sum of rule lengths

Discussion

In general, we observe that the grammar size may increase or decrease after the dynamic grammar refinement step. The grammar may grow if the definition of a non-terminal symbol is refined only in a specific context, leaving the original definition intact in a

Table 3.19: Manual effort (unique lines of code)

Subject	#L Tokenization Harness	#L Tokenization Proxy
TINY-C	3	2
LISP	5	5
MJS	5	10

different context. In contrast, the grammar size may also decline if the grammar is refined by deleting rule alternatives.

Note that the high numbers of rule alternatives is mostly due to lexical definitions. For instance, the character set `[a-z]` would add 26 rule alternatives, whereas `[\x01-\xff]` would even add 255 rule alternatives. Moreover, parsers may restrict character sets in certain contexts. As an example, a parser that parses quoted strings as part of its input language (e.g., *json*), may not allow the character `"` inside quoted strings, again adding a non-terminal definition with 254 rule alternatives. We leave it to future work to post-process mined grammars in order to reduce the number of rule alternatives in such cases. For *parson*, we found that the excessive number of rule alternatives is due to the violation of our assumption of incremental processing, as discussed in the previous research question, which leads to a large enumeration of possible characters in a specific context. Table 3.18 also shows that golden grammars are typically smaller than mined grammars, which is expected since they are carefully handcrafted. Moreover, as discussed in RQ5, the program under test can be more permissive than the golden grammars, justifying the larger size of some mined grammars.

Grammars mined by STALAGMITE *are reasonably sized. The use of function names to label non-terminal symbols enhances readability.*

3.5.3 RQ3: Scaffolding Effort

The effort required to prepare a parser for analysis with STALAGMITE is roughly equivalent to that of conventional software testing, i.e., setting up the program such that it accepts input from a test generator.

For subjects with a lexing stage (*mjs*, *Tiny-C*, *lisp*), STALAGMITE requires *glue code* that specifies the information flow between lexer and parser; see Figure 3.6 and Figure 3.7 for *Tiny-C*. To quantify the manual effort to prepare these subjects, Table 3.19 summarizes the number of unique lines of code we implemented. We expect the manual effort for this stage to take less than ten minutes.

In principle, an industrial implementation could eliminate this manual effort by automatically detecting information flow between lexer and parser and matching it against a number of common patterns. However, this is beyond the scope of this work.

Scaffolding code for the evaluation subjects with a lexing stage consists of between 5 and 15 unique lines of code.

3.5.4 RQ4: Time Consumption

In this research question, we investigate the time requirements of STALAGMITE and analyze its distribution over the different analysis steps. We ran STALAGMITE on each evaluation subject with a memory limit of 16 GB and a time limit of 24 hours for parser analysis, and two hours for tokenization analysis for parsers with a lexing stage.

Table 3.20: Time consumption

Subject	Lexer Analysis	Parser Analysis	Grammar Refinement	Total Time
TINY-C	00h:09m	05h:06m	00h:03m	05h:18m
LISP	00h:02m	24h:15m	<00h:01m	24h:18m
MJS	00h:43m	09h:19m	01h:16m	11h:19m
HARRYDC-JSON	N/A	11h:32m	<00h:01m	11h:32m
CJSON	N/A	24h:27m	00h:01m	24h:28m
PARSON	N/A	12h:36m	00h:28m	13h:04m
CALC	N/A	25h:18m	00h:01m	25h:20m
CALCCPP	N/A	24h:08m	<00h:01m	24h:08m
CGI-DECODE	N/A	<00h:01m	<00h:01m	<00h:01m

Table 3.20 summarizes the time consumption. Column *Lexer Analysis* shows the time spent on analyzing the lexer of parsers with a lexing stage. Column *Parser Analysis* indicates the time required by the parser analysis (including converting traces to the grammar), whereas column *Grammar Refinement* lists the times spent on refining the grammars.

STALAGMITE takes between one minute and 25 hours to infer grammars, requiring less than 16 GB of memory.

3.5.5 RQ5: Bugs Found

From a developer perspective, it is crucial to ensure that the parser implementation matches the language specification. In this research question, we evaluate if the implemented parsers are *too permissive*, accepting some inputs that are not accepted by the specification. We assume the specification is provided by the golden grammar used in RQ1. As an example, we compare the grammar inferred from the *harrydc-json* parser with the golden *json* grammar to see if it accepts more than the golden grammar allows.

We address RQ5 by

(1) generating inputs from the mined grammar,

(2) retaining those inputs that can be parsed by the parser implementation under test, and

(3) checking whether the golden grammar can also parse these inputs.

In case the golden grammar cannot parse an input that is accepted by the implementation, we say that the implementation is too permissive. We applied this methodology on the evaluation subjects, and present some of the mismatches we discovered.

JSON. For the *harrydc-json* parser, we detected a syntactic mismatch in the STALAGMITE-extracted grammar. In Figure 3.1, consider the definition of `<json_parse_object'>`. We see that keys of objects may be any `<json_parse_value>`, including strings, numbers, arrays, and objects—say, `{1: "foo"}`. However, the *json* specification[4] and our golden grammar only permit strings as keys of objects. The issue was validated with concrete inputs using the *harrydc-json* parser as oracle. It is easy to pinpoint such issues in STALAGMITE grammars.

***mjs*.** For *mjs*, we detected two syntactic mismatches, which are not allowed in JavaScript. Figure 3.21 shows an excerpt of the mined *mjs* grammar, defining variable declarations. 1. The `<opt_comma>` definition shows that in *mjs* `let` constructs, commas are *optional* between variable names. As a consequence, it is possible to declare multiple variables by separating them with a whitespace only (e.g., `"let a b"`). 2. Moreover, variable declarations may contain a trailing comma. For instance, `"let a, b,"` is valid in *mjs*.

```
<statement> ::= "let" <let_loop> | ...
<let_loop>  ::= <ascii_str> <opt_comma> <let_loop> | ...
<opt_comma> ::= "" | ","
```

Figure 3.21: Simplified *mjs* `let` declarations mined by STALAGMITE

These examples highlight one of the advantages of symbolic input grammar mining. Sample-based grammar miners typically learn from *well-formed* inputs that do not include faulty input features. As a result, they are less likely to find bugs associated with such borderline-valid inputs. In principle, one could use KLEE-generated (and thus potentially semi-valid) inputs with sample-based grammar miners, as we did in RQ1, and expect to find similar bugs. To assess this possibility, we manually inspected the *mjs* and *json* grammars mined in RQ1 by Kedavra and Mimid. However, we found that the mismatches discussed in this research question were not captured in their mined grammars—most likely because the relevant language features were not

[4] `https://ecma-international.org/publications-and-standards/standards/ecma-404/`

present in the sample inputs. In summary, developers can use STALAGMITE to catch mismatches in their implementation easily. The alternative would be to manually assess the parser implementation for syntactic mismatches, which is time-consuming and error-prone.

STALAGMITE can learn grammars accurately, capturing even subtle details.

3.6 Related Work

Automated inference of input grammars has been studied extensively, with the vast majority of existing work focusing on learning input grammars from examples.

3.6.1 Language Inference

In her seminal paper [3], Angluin presented the L* algorithm, which can learn regular languages by means of active learning. This algorithm requires a *teacher* that serves as a parse oracle and can decide whether the *learned language* is equivalent to the target language, providing a counterexample in case of a negative answer. In the same paper, Angluin also presented the L^{cf} algorithm to address learning of context-free languages in a similar way. However, this algorithm is not practical due to its strong assumptions on the grammar and information required by the learner. In general, black-box inference of context-free grammars was shown to be computationally infeasible by Angluin et al. [4].

Therefore, recent black-box approaches rely on the fact that context-free languages might not be as complicated in practice [64], giving raise to heuristics-infused black-box learning. Moreover, white-box approaches [52, 57, 96] leverage knowledge of the program code to draw conclusions about the input language accepted by the parser.

3.6.2 Black-Box Grammar Inference

Arvada [64] is a black-box approach to learn context-free grammars based on a set of positive sample inputs and a parse oracle. It starts by generating *flat* parse trees (i.e., the root node has one child for each input character) for each input. Next, it gradually adds *structure* to these trees by combining sibling nodes under a new intermediate parent node (*bubbling* operation), retaining only those bubbles that can be merged later. The validity of a merge operation is checked using the parse oracle. As a black-box approach, Arvada is agnostic of the implementation. However, it is unlikely to synthesize features not found in sample inputs.

Treevada [5] finds that the Arvada approach is non-deterministic, and requires very short sample inputs to be available. Treevada introduces a pre-structuring heuristic, which avoids any generalization attempts that would result in unbalanced brackets.

Kedavra [67] is an improvement over TreeVada and follows a similar technique. It first decomposes sample inputs on token boundaries and then infers the grammar incrementally, resulting in a more efficient and effective inference process.

By construction, all these approaches require sample inputs on top of the program under test—in contrast to STALAGMITE, which only needs the input-processing program, but also the ability to analyze it. Given a sufficiently large and sufficiently diverse of inputs to learn from, Arvada, Treevada, and Kedavra can all produce useful grammars. However, if such a set of inputs is already available, one may also use it directly for tasks such as testing.

3.6.3 White-Box Grammar Inference

Autogram [57] is the seminal work on white-box mining of context-free input grammars. This approach is based on dynamically tracking data flow in a parser when it processes the input. Hence, Autogram depends on sample inputs and requires that data flow is present in the first place.

Being the approach most closely related to ours, Mimid [52] is a white-box approach to learn context-free grammars from recursive descent parsers. However, in contrast to STALAGMITE, Mimid requires sample inputs. In essence, Mimid tracks how the parser implementation processes inputs, labeling an input character with the execution context (based on the call stack and control-flow structures) at the point of its acceptance. This labeling allows to derive grammar-inducing parse trees from the seed inputs, which are further generalized using experiments.

All these approaches are *dynamic* and require sample inputs—in contrast to STALAGMITE, which only needs the input-processing program. And again, if a feature is not present in the set of sample inputs, it is unlikely to be learned by these approaches.

3.6.4 Static Grammar Inference

To our knowledge, the only other work that statically mines grammars from code is by Schröder et al. [96, 97]. Like STALAGMITE, it requires no sample inputs but targets *short ad-hoc Python parsers*, e.g., `xs = map(int, s.split(','))`, extracting input grammars using summaries of commonly used string functions. In contrast, STALAGMITE uses symbolic execution and can handle much larger recursive descent parsers implemented in C.

The idea of STALAGMITE, including first results, were first published in a short paper [24].

3.6.5 Generating Valid Inputs for Parser-Driven Programs

In this section, we review approaches for testing program code that lies beyond an initial input parser. While mature tools like AFL are highly effective at uncovering robustness issues such as crashes in parsers themselves, testing downstream program logic requires generating valid inputs that are not rejected by the parser.

Early work in this area includes *Grammar-based Whitebox Fuzzing* by Godefroid et al. [49], which leverages context-free grammars to enhance symbolic execution of programs that process structured inputs, such as interpreters. LANGFUZZ [56] is a fuzzer that builds on context-free grammars to parse and recombine inputs from bug repositories, which were known to trigger faults in previous versions of the program under test. It achieved notable success by uncovering over 2,600 bugs in widely used JavaScript engines.

More recent approaches have explored extracting information from parsers using dynamic tainting. lFuzzer [73] uses dynamic tainting to learn valid tokens from input parsers. These tokens can then be included in the dictionary of fuzzing tools like AFL, enabling more efficient token-level mutations. Similarly, pFuzzer [74] extends this concept by not only learning tokens incrementally but also combining them to synthesize valid inputs.

In contrast to these techniques, STALAGMITE is a *static* technique to infer input grammars from the code of parsers.

3.7 Limitations and Future Work

While STALAGMITE makes significant contributions, acknowledging its limitations opens exciting opportunities for future work.

Accurate token mining. In STALAGMITE, we learn tokens by generating token instances using symbolic execution and matching them with predefined patterns. While we expect most tokens to belong to common categories such as uppercase strings or digits, in general, tokens can represent any regular language, which may not always align with one of the predefined patterns, leading to overapproximation. Future work may learn tokens more accurately—e.g., using active learning of regular expressions [3].

Automatic harness construction. While STALAGMITE currently only requires a small amount of manual work to handle parsers with a lexing stage, we will investigate ways to further automate our analysis for such parsers.

Semantic constraints. Our STALAGMITE approach is entirely focused on learning the syntax of inputs. Input formats may additionally have non-syntactic constraints,

such as *a variable must be defined before it used*. As part of our ongoing research, we explore ways to *extract* such semantic constraints from a given program in order to *lift* it to the input specification, thereby enhancing its precision even further. The input specification language ISLa [106] allows to specify such semantic constraints in conjunction with context-free input grammars.

Other domains. In this work, we have investigated the inference of context-free input grammars from recursive descent parsers. Such parsers typically process recursive text-based languages. In the future, we would like to extend our approach to other domains, such as text-based and binary protocols, as well as parsers processing binary data.

Assumptions. Finally, the assumptions in Section 3.2.1 can all be read as limitations, too.

3.8 Conclusion

Accurate input specifications allow fuzzers to efficiently and comprehensively test programs. Up until now, all approaches to automatically infer input grammars have required a set of *sample inputs* as a starting point. Consequently, these approaches could only learn input features that were present in the provided sample inputs, making them inherently dependent on the quality and diversity of the sample inputs. In this chapter, we present STALAGMITE, the first static input grammar mining technique, which produces accurate input grammars entirely without the need for sample inputs. Our evaluation shows that grammars produced by STALAGMITE can have near-perfect precision and recall, and are very accurate at capturing the input format accepted by the program under test.

3.9 Data Availability

All of STALAGMITE and experimental data is available at:

`https://github.com/leonbett/stalagmite`

4 Input Grammar Evaluation

This chapter is taken, either directly or with minor modifications, from our 2026 ICSE paper *How Good are Input Grammar Miners? An Empirical Study*.
My contribution in this work is as follows:
① original idea,
② implementation, and
③ evaluation.

Chapter Overview

To generate valid test inputs for a system, one needs a *specification* of its input language—typically a *context-free grammar* that describes input syntax. But where can one get such a grammar from? In the past years, the field of *input grammar mining* has emerged, with creative approaches to extract input grammars from inputs, code, or both. But how good are these approaches? In particular; How *accurate* are the grammars they mine?

In this study, we systematically *evaluate* grammar miners for these questions. Notably, we find that the previous evaluations conducted by the respective authors—producing a set of inputs from a golden grammar and having them checked by the mined grammar, or vice versa—are insufficient, as they have a strong bias towards short, possibly unrealistic inputs. We therefore also measure the *diversity* of the mined grammars using k-*path coverage* with varying depths k to find how many *combinations* of grammar elements are actually represented.

Ideally, a mined grammar should have perfect precision and recall regardless of the depth k. However, our results show that for all approaches presented so far, precision and recall can drop significantly compared to reported results when increasing k and thus checking for "deeper" diversity, especially for complex input languages such as Lisp, JSON, or Tiny-C. For instance, the Tiny-C grammar mined by Arvada achieves a precision of 75% when considering k-paths with $k = 1$ (the originally reported precision was 73%), but this drops to 46% for $k = 5$. White-box approaches based on program analysis, such as Mimid and Stalagmite, are more stable with varying depth k, but can be challenged by complex parsers such as mjs. Raising the bars for evaluation, our study shows that there is still room for improvement in grammar mining.

4.1 Introduction

The field of input grammar mining [64, 5, 67, 57, 52, 23] has gained significant attention in recent years due to its potential to improve software testing and reverse engineering. The core challenge it tackles is to extract a context-free grammar from a given program that accurately describes the accepted input syntax. This grammar can then be used to generate test inputs, passing initial validation checks and potentially revealing deep bugs in the program under test. Tools such as *Grammarinator* [55], *ISLa* [106], *FANDANGO* [122] and the fuzzers included in *The Fuzzing Book* [126] are able to directly transform grammars into highly efficient input producers. LANGFUZZ [56] has detected more than 2,600 bugs in JavaScript engines using grammar-based fuzzing to date. Beyond testing, extracted grammars facilitate reverse engineering by helping analysts to reconstruct the input format of a program. More broadly, grammars play an important role in many areas. Grammars are the base for parsing inputs, allowing for the reuse of data and code formats. They are also the method of choice to document interactions and messages—21% of all RFCs contain the grammar operator "::=" or the string "BNF" for Backus-Naur-Form, a standard syntax for grammars.

The general problem of inferring context-free grammars is hard. For instance, one cannot infer a context-free grammar from (black-box) membership queries only [4]. However, this restriction may not necessarily hold for context-free grammars and programs used in practice. The question is: *How close can an extracted grammar approximate a real input grammar?* This resulted in the development of various *grammar mining techniques,* which can be broadly categorized into (1) *black-box* approaches [64, 5, 67], learning from input samples and membership queries; and (2) *white-box* [57, 52, 23] approaches, analyzing program code and its executions. To compare techniques, the core metric used in the literature is *grammar accuracy*, which measures how *similar* the language induced by the mined grammar is to that of a *golden grammar* (i.e., a reference grammar serving as ground truth). It is crucial for the grammar accuracy metric to reflect faithfully how well the mined grammar approximates the golden grammar.

However, we identify a critical issue in the existing literature. The commonly used grammar accuracy metric is insufficient for effectively evaluating the quality of mined grammars. Specifically, the current metric is based on randomly generating a set of 1,000 random strings from both the mined and the golden grammar. Each set is then parsed using the other grammar (if possible) to determine precision and recall. This can be problematic for several reasons: (1) the sampling process is *unsystematic,* favoring short inputs, and may not cover the grammar space sufficiently, (2) the *number of samples* is arbitrary and not justified, (3) the metric does not provide *fine-grained insights* into which language features were learned.

To investigate this potential issue, we conduct an empirical study, reassessing the accuracy of the five most recent state-of-the-art input grammar mining approaches by means of a systematic evaluation based on k-*path coverage* [54] that addresses the

aforementioned limitations. Intuitively, k-path coverage measures how many different depth combinations of length k are covered by the mined grammar; it thus does not only measure coverage of individual grammar elements, but also the number of contexts in which these elements appear. The greater the value of k, the deeper the context.

To evaluate an input grammar, we start by generating the set of all k-paths (i.e., sequences of k connected non-terminals in the grammar graph) for $k = 1 \ldots n$. Next, we proceed as follows to construct a derivation tree for each k-path, ensuring that the tree contains the given k-path:

(1) the root node of the derivation tree is set to the start non-terminal of the grammar,

(2) which is then connected via the shortest path in the grammar to the root node of the k-path;

(3) at last, all unexpanded nodes in the derivation tree are expanded with random but cheap productions.

We then use the inputs represented by these derivation trees to calculate grammar accuracy. This approach has several advantages over the existing metric: (1) it systematically covers the grammar space by considering all k-paths, (2) it provides fine-grained insights into the language features covered by the mined grammar, (3) it allows to cover all k-paths, even when they are only reachable via a long chain of derivations. By re-evaluating the input grammars mined by five recent input grammar mining approaches, our empirical study finds that their accuracy is often lower than reported, with 22% of grammars showing a significant drop in precision, and 30% exhibiting a significant decrease in recall.

In summary, our contributions are:

- We introduce a systematic metric to evaluate the accuracy of mined input grammars based on k-path coverage.
- We re-evaluate input grammars from existing grammar mining papers using our metric, providing new insights into their effectiveness.
- We provide an open-source implementation of our metric to facilitate its adoption in future research.

The remainder of this chapter is organized as follows. **Section 4.2** provides background information on input grammar mining, the challenges of comparing grammars, and k-path coverage. **Section 4.3** introduces our technique, which systematically generates k-path covering inputs from input grammars. **Section 4.4** briefly describes the implementation of our technique. **Section 4.5** presents a systematic evaluation of the input grammars mined by five existing grammar mining approaches. **Section 4.6** discusses related work. **Section 4.7** summarizes possible threats to validity of our study. **Sec-**

tion 4.8 closes the chapter with our conclusion and implications for grammar mining research.

4.2 Background

4.2.1 Input Grammar Mining

The goal of input grammar mining is to automatically learn a context-free grammar that accurately describes the input syntax accepted by a given program. When such an input grammar is available, it can greatly enhance software testing and reverse engineering. In test input generation, grammar-based fuzzers leverage input grammars to efficiently produce inputs that cover all language features—and hence, program behaviors [54]. Additionally, input grammars can act as documentation for the input language understood by the program, supporting both development and reverse engineering efforts.

Existing grammar mining approaches can be broadly categorized into black-box and white-box techniques. Black-box techniques [64, 5, 67] typically require valid sample inputs and black-box access to the program under test. In contrast, white-box techniques depend on program analysis to extract the grammar from program code. Some white-box techniques [52, 57] also require an initial set of valid sample inputs, while others [24, 23] do not.

A major challenge for black-box techniques is dealing with incomplete sets of sample inputs. These techniques often struggle to infer language features that do not appear in at least one sample. In contrast, the main challenge for white-box techniques is to analyze the program code to determine the language it recognizes. To date, all existing white-box approaches rely on the input parser of the program under test being implemented in a recursive descent style.

Input grammar mining has gained significant attention in recent years, with the research frontiers being learning grammars more accurately, scaling to more complex input languages, and reducing requirements. For example, Stalagmite [23] infers grammars directly from the program code, without depending on sample inputs like other techniques do.

4.2.2 On the Difficulty of Comparing Grammars

Why is comparing context-free grammars so difficult? At its core, the difficulty arises from the well-established fact that determining the equivalence of two context-free grammars is undecidable [101]. In other words, no algorithm can universally determine whether two given grammars generate the exact same language. However, in many practical scenarios, a strict binary determination of equivalence is neither necessary

nor desirable. Instead, it is often more informative to assess the degree of similarity between grammars rather than merely confirming or refuting their equivalence.

Mined grammar. A context-free grammar describing the syntax of inputs accepted by the program under test. Such grammars are automatically inferred by grammar miners.

Golden grammar. A grammar written or extracted from documentation by hand. Such grammars serve as ground truth to evaluate grammar miners, i.e., by comparing how well a mined grammar approximates a golden grammar.

In grammar mining research, the prevailing approach for measuring grammar accuracy is as follows:

1. Generate 1,000 strings from the **golden grammar** by randomly expanding a derivation tree, initially consisting of only the start non-terminal, until all nodes are expanded. In order to avoid unbounded recursion, the depth of the derivation tree is limited by a small (typically 10) constant [52, 64].

2. For each of these strings, check if the **mined grammar** can parse it. Then compute *recall* as $\frac{\#\,\text{parsed strings}}{1{,}000}$.

3. To compute precision, repeat the two steps above, but swap golden grammar and mined grammar.

4. Finally, compute accuracy as the harmonic mean of precision and recall.

In the following, we will show an artificial example that illustrates the limitations of this prevailing approach and motivates the need for a more systematic metric. We start with a definition of grammar ambiguity, one of the key challenges in comparing grammars, and then construct a simple ambiguous grammar that is able to fool the existing metric.

Definition 11: Grammar Ambiguity

A string s is derived ambiguously in a context-free grammar G if there are at least two different leftmost derivations for s in G. A grammar G is **ambiguous** if it generates some string ambiguously [101].

Some languages are inherently ambiguous, requiring an ambiguous grammar to describe them; however, in most practically relevant cases, grammar ambiguity is not required, but may occur.

"Million Dollar Grammar"

Consider the following regular expression, which matches arbitrary non-empty strings:

```
[\s\S]+
```

We can easily translate it to a context-free grammar:

```
<start> ::= <any>
<any>   ::= <byte> <any> | <byte>
<byte>  ::= "\x00" | "\x01" | ... | "\xFF"
```

We can also express the language with a more verbose, equivalent but ambiguous regular expression `[\s\S]+ | [0-9]+`, which we can again translate to a grammar:

```
<start>  ::= <any> | <digits>
<any>    ::= <byte> <any> | <byte>
<byte>   ::= "\x00" | "\x01" | ... | "\xFF"
<digits> ::= <digit> <digits> | <digit>
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"
```

This grammar is equivalent to the initial grammar as it also accepts arbitrary non-empty strings via `<any>`. The ambiguity stems from the fact that `<digits>` is a subset of `<any>`. Similarly, we can construct an even larger, much more ambiguous grammar (Figure 4.1) that is still equivalent to the initial grammar.

```
<start>  ::= <a1>
<a1>     ::= <digits> | <a2>
<a2>     ::= <digits> | <a3>
...      ::= ...
<a9>     ::= <digits> | <a10>
<a10>    ::= "0" | <any>
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"
<digits> ::= <digit> <digits> | <digit>
<byte>   ::= "\x00" | "\x01" | ... | "\xFF"
<any>    ::= <byte> <any> | <byte>
```

Figure 4.1: A nested and ambiguous grammar that is equivalent to the initial grammar

When compared with a golden grammar for *JSON*, unsurprisingly, this grammar would score a recall of 100%. This is because the grammar can parse any string, including any

JSON string. Surprisingly, however, this grammar also scores a precision of 100% on the existing metric when compared with the golden grammar for *JSON*. As discussed above, the existing metric bounds the depth of the derivation tree when generating inputs. In this case, the `<any>` non-terminal will never be generated, because it is only reachable with a depth of over 10, and the existing metric will "close off" the production with the cheapest alternative beyond that depth (here: `<a10> ::= "0"`). As a consequence, only digits are ever generated. Since numbers are valid JSON strings[1], all generated inputs will be accepted by the golden grammar, resulting in perfect precision.

While we deliberately constructed this example to illustrate the limitations of the existing metric, it highlights the need for a more systematic approach to evaluating grammar accuracy.

4.2.3 k-Paths

Havrikov and Zeller introduced the concept of *k-path coverage* [54] as a metric for evaluating the extent to which inputs cover the syntactic features of an underlying grammar. In this section, we provide a concise overview of k-path coverage and graph-based representation of grammars.

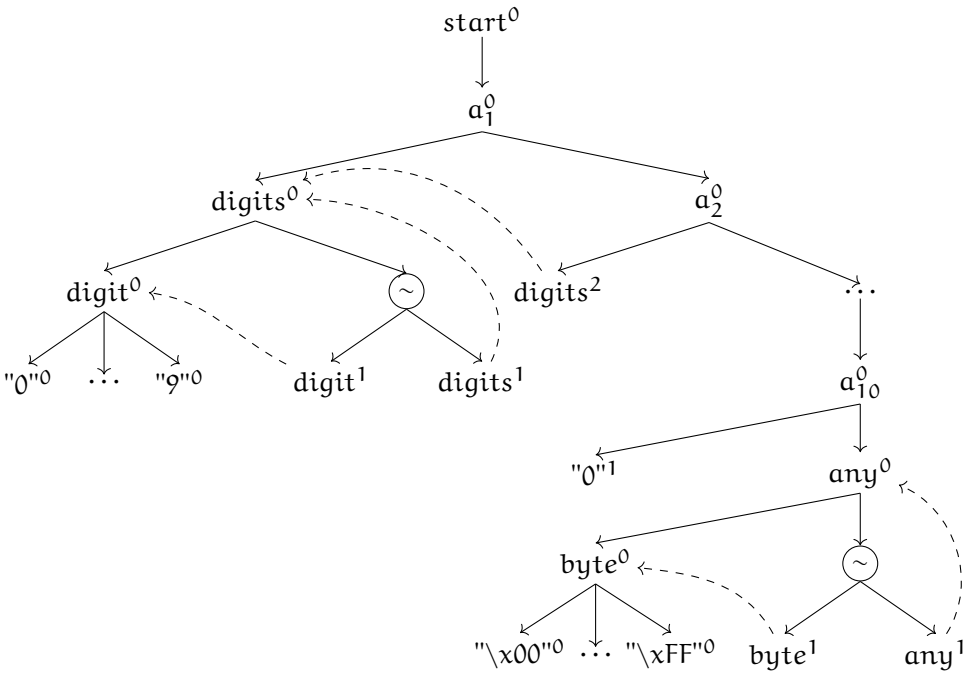


Figure 4.2: Graph representation of the grammar in Figure 4.1

[1] `https://www.json.org/json-en.html`

Consider the grammar graph in Figure 4.2, which relates to the grammar in Figure 4.1 as follows:

- Each node with an outgoing edge is a non-terminal symbol.
- Each node without an outgoing edge is a terminal symbol.
- Each directed edge from node A to node B implies that B is a production rule of A.
- The special node $\sim$ implies that its children are to be concatenated as one production rule.
- Each node is labeled with a unique superscript number to distinguish equal nodes in different contexts (referred to as a symbolic node).
- Dashed lines have the same semantics as solid lines, but help to maintain the visibility of the underlying tree-like structure.

Based on this grammar graph representation, which is further detailed in [54], one can define the concept of k-paths:

Definition 12: k-path

A k-path is a sequence of nodes connected in the direction of the edges with exactly k symbolic nodes [54].

For the special case of $k = 1$, the k-paths correspond to all nodes in the grammar graph, with duplicates (due to different contexts) removed. That is, superscripts are not taken into account. For Figure 4.2, for instance, the **1-paths** are `start`, `digits`, `digit`, `byte`, `any`, a_1, ..., a_{10}, `"0"`, ..., `"9"`, `"\x00"`, ..., `"\xFF"`. For $k > 1$, we take contexts (i.e., superscripts) into account. For example, the **2-paths** are $(start^0 \rightarrow a_1^0)$, $(a_1^0 \rightarrow digits^0)$, $(a_1^0 \rightarrow a_2^0)$, $(digits^0 \rightarrow digit^0)$, $(digit^0 \rightarrow "0"^0)$, and so on.

How does this relate to grammar accuracy? Clearly, if the inputs used to compute grammar precision covered all 1-paths in Figure 4.2, the precision would be well below 100%. Specifically, if an input includes the 1-path `any`, `byte`, or any of the production rules of `byte`, then the input is not valid JSON with a high probability—it is simply a random string. As we consider more and deeper k-paths, our confidence in assessing grammar accuracy increases.

4.3 Measuring Grammar Accuracy

As discussed in Section 4.2.2, the existing grammar accuracy metric may not adequately cover the grammar space. In contrast, k-paths offer a structured way to evaluate how well inputs capture the syntactic features of a given grammar. The following section

introduces our GRAMACCU technique, which systematically generates inputs using *k-paths* and uses them to measure grammar accuracy.

Note that the original k-path work [54] already proposed an algorithm that produces k-path covering inputs. However, that algorithm aims to include multiple k-paths within a single tree to keep the total number of generated input trees small. In contrast, our algorithm constructs a new tree for each k-path that specifically contains that k-path, but otherwise tries to avoid adding more unrelated k-paths. The rationale is that we do not want to influence the measurement with unrelated k-paths included in the tree. We do not consider our adapted input generation algorithm to be our main contribution. Instead, we see it as a vehicle to enable our empirical study. We consider the empirical study of existing input grammar miners, and the insight that k-path based input generation is useful for assessing grammar similarity, to be our main contributions.

4.3.1 Generating Inputs

The high-level algorithm used by GRAMACCU for generating inputs is shown in Algorithm 4. Next, we demonstrate each step of the algorithm on a simple example. Let us suppose we enumerated all 3-paths for the grammar graph in Figure 4.2, and we currently process one of them, say $(\texttt{byte}^1 \rightarrow \texttt{byte}^0 \rightarrow \texttt{"\textbackslash x42"}^0)$.

Algorithm 4 Generating k-Path Inputs from the Grammar

```
1:  procedure GenerateInputs(grammar, k)
2:      inputs ← []
3:      kPaths ← GenerateKPaths(grammar, k)
4:      for each kPath in kPaths do
5:          ① kPathTree ← KPathToTree(kPath)
6:          ② rootedTree ← RootTree(grammar, kPathTree)
7:          ③ finalTree ← ExpandTree(rootedTree)
8:          inputs ← inputs + TreeToStr(finalTree)
9:      end for
10:     return inputs
11: end procedure
```

The steps of the algorithm are as follows:

1. Convert this 3-path into a subtree, consistent with the grammar graph ①. The resulting subtree is shown in Figure 4.3, highlighted in blue.

2. Root this k-path subtree at the start non-terminal of the grammar ②. In our case, the start non-terminal is **<start>**. To root the tree, we compute the shortest path from $\texttt{start}^0$ to the root node of the 3-path, i.e., $\texttt{byte}^1$. In case there are multiple shortest paths, we choose the first one we encounter during breadth-first search.

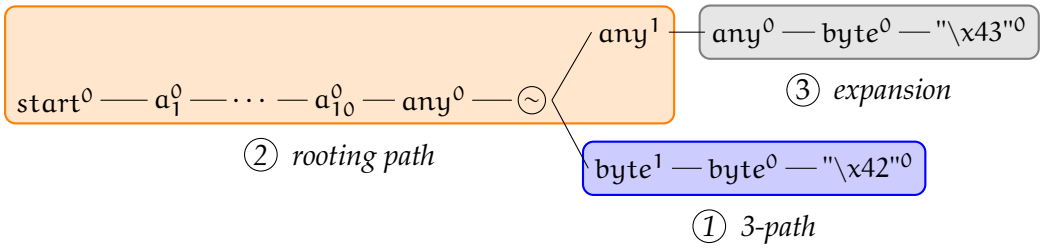


Figure 4.3: Rooted and expanded derivation tree for the 3-path $(\texttt{byte}^1 \rightarrow \texttt{byte}^0 \rightarrow \texttt{"\textbackslash x42"}^0)$

At this step, we only expand non-terminals that are on the shortest path from the start node to the root node of the 3-path. The rooted tree is shown in Figure 4.3, highlighted in orange and blue.

3. Finally, expand all unexpanded non-terminals in the tree with cheap productions ③. The result of this step is highlighted in gray in Figure 4.3. Note that the expansion $(\texttt{byte}^0 \rightarrow \texttt{"\textbackslash x43"}^0)$ was chosen randomly.

This derivation tree is then converted into a string by concatenating all terminal symbols, i.e., `"\x42\x43"`.

4.3.2 Computing Grammar Accuracy

Next, we explain how GRAMACCU computes grammar accuracy using the GENERATEINPUTS function. The high-level algorithm is shown in Algorithm 5. This algorithm takes as input two grammars, *minedGram* and *goldenGram*, along with a positive integer *kLimit* that defines the maximum k-path length. For each of the grammars, the algorithm generates inputs using the GENERATEINPUTS function for $k = 1, \ldots,$ *kLimit* and tries to parse them with the other grammar. More precisely, precision is computed by generating inputs from *minedGram* and checking if `goldenGram` can parse them. Recall is computed by generating inputs from *goldenGram* and checking if *minedGram* can parse them.

This algorithm is designed to evaluate black-box grammar mining techniques. For white-box approaches, it needs to be slightly adapted, as the program under test (PUT) serves as a parse oracle. In this case, precision is measured by generating inputs from the mined grammar and checking whether the PUT can successfully parse them. As with black-box approaches, recall is calculated by generating inputs from the golden grammar and checking if the mined grammar can parse them. However, since the golden grammar may be more permissive than the PUT, the generated inputs are first filtered using the PUT to ensure they are valid.

In our experiments, we set *kLimit* $= 5$, which proved to be reliable number for all grammars, as it covers a significant portion of the grammar space, and terminates in

Algorithm 5 Evaluating Grammar Accuracy

```
1:  procedure Eval(g_Gen, g_Parse, kLimit)
2:      inputs ← []
3:      for k ← 1 to kLimit do
4:          inputs ← inputs + GenerateInputs(g_Gen, k)
5:      end for
6:      parsed ← 0
7:      for each input in inputs do
8:          if CanParse(g_Parse, input) then
9:              parsed ← parsed +1
10:         end if
11:     end for
12:     return parsed / Length(inputs)
13: end procedure
14: procedure Accuracy(minedGram, goldenGram, kLimit=5)
15:     precision ← Eval(minedGram, goldenGram, kLimit)
16:     recall ← Eval(goldenGram, minedGram, kLimit)
17:     return precision, recall
18: end procedure
```

a reasonable amount of time. However, note that k-paths grow exponentially. Hence, for larger grammars, it may be necessary to reduce *kLimit* to avoid excessive time consumption.

4.4 Implementation

We implemented our GRAMACCU technique in Python and provide an easy-to-use command-line interface. Our code is based on the existing GrammarGraph[2] implementation, and expects grammars to be represented in the JSON-based FuzzingBook [126] format. As we discuss in Section 4.5, we provide conversion scripts to translate grammars from existing grammar mining papers to this format, should they not already use it.

4.5 Evaluation

We systematically evaluate five existing input grammar mining approaches (detailed in Section 4.5.1) using GRAMACCU. Our evaluation aims to assess grammar accuracy more comprehensively and gain additional insights into the quality of the inferred grammars.

[2] `https://github.com/rindPHI/GrammarGraph`

To enable this comparison, we first reproduce the input grammars using the respective reproducibility packages. Next, we convert the grammars into the FuzzingBook grammar format, which serves as input format for our GRAMACCU implementation.

We address the following research questions:

RQ1: Revisiting Grammar Accuracy. How does the grammar accuracy computed by GRAMACCU compare to the reported accuracy in five existing grammar mining papers?

RQ2: Golden Grammar Coverage. How well do inputs generated using GRAMACCU from the mined grammars cover k-paths in the golden grammars?

RQ3: Qualitative Analysis of Black-Box Approaches. Which language features are captured by the mined grammars? Do they extend beyond the training inputs?

RQ4: Growth of k-Paths. How does the number of k-paths scale as k increases, and what are the practical implications?

RQ5: Added Value of k > 1. How do larger k values improve grammar evaluation?

4.5.1 Compared Approaches

In this section, we provide information on the grammar mining approaches compared in our evaluation. We focus on their requirements, and detail the original methods used to compute precision and recall by the respective papers. Table 4.4 lists the approaches we compare in this study. We selected these grammar miners because they represent the state of the art in grammar inference—covering the most recent black-box (Arvada, Treevada, Kedavra) and white-box (Mimid, Stalagmite) techniques—with all providing publicly available reference implementations.

Arvada [64]

Being the first black-box approach in this line of research, Arvada was evaluated on eight ANTLR4 parsers (*arith*, *fol*, *json*, *lisp*, *mathexpr*, *turtle*, *while*, and *xml*), as well as on three standalone parsers (*curl*, *Tiny-C*, and *nodejs*). For each of these parsers except *nodejs*, Arvada provides a golden grammar. For the ANTLR4 parsers, the golden grammar is simply the ANTLR4 grammar. For *Tiny-C*, the golden grammar of Mimid was used. For *curl*, the golden grammar was derived from RFC 1738. For *nodejs*, no golden grammar was used. Instead, inputs were generated using the JavaScript generator of Zest [79]. Based on these golden grammars, the test and training inputs, which are provided as part of the reproducibility package of Arvada, were produced as follows [64]:

Approach	WB/ BB	PL	Train Inputs	# Test Inputs	Grammar Format
Arvada	BB	N/A	Golden	Golden (1K)	Custom-1
Treevada	BB	N/A	Arvada	Arvada	Custom-1
Kedavra	BB	N/A	Arvada	Arvada	Custom-2
Mimid	WB	Py, C	Golden	Golden (1K)	FuzzingBook
Stalagmite	WB	LLVM	No	Golden (1K)	FuzzingBook

Table 4.4: Comparison of grammar mining approaches in our evaluation. The *WB/BB* column specifies whether each approach is black-box or white-box. *PL* indicates the programming languages supported by white-box approaches. *Train Inputs* and # *Test Inputs* detail the source and number of training and test inputs used in the original studies. *Grammar Format* presents the original grammar format.

- **Training inputs** were generated by ensuring all production rules of the golden grammar are covered, while keeping the individual training inputs small.
- **Test inputs** were generated by randomly sampling 1,000 inputs from the golden grammar. Details on how this was done were not reported.

To compute recall, Arvada checks how many of the 1,000 inputs in the test set are accepted by the mined grammar. To compute precision, Arvada randomly samples 1,000 inputs from the mined grammar and checks how many are accepted by the parse oracle.

Data Collection

We reproduced the evaluation of the Arvada paper by running the Docker container [89] from the reproducibility package and extracted the mined input grammars, golden grammars, training inputs, and test inputs. In our evaluation, we excluded the *nodejs* subject for all compared approaches because it lacks a golden grammar, which is essential for our evaluation. For the *curl* subject, a golden grammar was not provided, but we rebuilt it from RFC 1738. To prepare the grammars for our evaluation with GRAMACCU, we converted them to the FuzzingBook grammar format. Specifically, for mined grammars, we implemented a converter in Python that translates Arvada grammars to FuzzingBook format. For the golden grammars, we used an existing tool that converts ANTLR4 grammars to the FuzzingBook format [51]. As Arvada is a non-deterministic approach, the reproducibility package runs it ten times, producing ten grammars for each subject. To enable a fair comparison in our evaluation, we consider all ten grammars for each subject and average precision and recall.

Treevada [5]

Except for the subjects *arith*, *fol*, and *mathexpr*, Treevada was evaluated on the same parsers as Arvada. These subjects were excluded because their parsers took extremely long to parse some inputs. While Arvada handled this by setting a timeout, treating inputs that hit the timeout as valid, Treevada excluded these subjects to preserve the integrity of precision calculations.

Treevada and Kedavra are closely related incremental successors of Arvada, and both reuse the same test set provided by the reproducibility package of Arvada to compute recall. Similarly, precision is computed analogously to Arvada by these two approaches.

Data Collection

We reproduced the evaluation of the TreeVada paper by running the provided Docker container [93] and extracted the mined input grammars, golden grammars, training inputs, and test inputs. Although Treevada was also evaluated on the Arvada training set "R0", its reproducibility package only generates results for the training set "R1" (which was introduced by Treevada). Hence, in our evaluation, the grammars mined by Treevada are based on the "R1" set. As Treevada uses Arvada's grammar format, we reused that converter to translate grammars to the FuzzingBook format.

Kedavra [67]

Except for *nodejs*, for which the authors of Kedavra have found the parse oracle to be implemented incorrectly, Kedavra was evaluated on the same parsers as Arvada. Additionally, Kedavra was evaluated on a new subject called *minic*, which is a C-like language implemented as an ANTLR4 grammar.

Kedavra found the sampling algorithm of Arvada to be insufficient, as it rejects samples that are longer than 300 characters, and limits expansions to a recursion depth of five. To address this, Kedavra introduced a new sampling strategy that limits the usage of each production rule to no more than ten times, trying to explore the grammar more deeply. However, this strategy is also insufficient, as it does not guarantee that all k-paths are covered.

Data Collection

We reproduced the evaluation of the Kedavra paper by running the provided Python code [90] and extracted the mined input grammars, golden grammars, training input samples, and test input samples. We implemented a converter in Python that translates Kedavra grammars to the FuzzingBook format.

Mimid [52]

The dynamic white-box approach Mimid was evaluated on six Python subjects; *calc.py*, *cgidecode.py*, *mathexpr.py*, *microjson.py*, *parseclisp.py*, and *urlparse.py*; as well as on three C subjects; *Tiny-C*, *mjs*, and *json*— all of which are implemented as recursive descent parsers. For each of these subjects, Mimid provides a golden grammar. The training inputs were constructed in different ways. For some subjects, they were generated using the golden grammar. For others, they were collected from the web or generated by hand.

To compute precision, Mimid randomly generates 1,000 inputs from the mined grammar with a recursion depth limit of ten and checks how many are accepted by the parse oracle. To compute recall, Mimid generates 1,000 inputs, which are accepted by the parse oracle, from the golden grammar, and checks how many of them are accepted by the mined grammar.

Data Collection

For Python subjects, we collected the mined input grammars and golden grammars from the Jupyter Notebook linked in the GitHub repository [91] of the paper. For C subjects, we ran the C reproducibility package of Mimid and extracted the mined input grammars and golden grammars. The set of test inputs is generated on-the-fly in the reproducibility package and not readily available. As Mimid already outputs grammars in the FuzzingBook format, no conversion was necessary.

Stalagmite [23]

The static white-box approach Stalagmite was evaluated on eight C subjects; *Tiny-C*, *harrydc-json*, *mjs*, *lisp*, *cjson*, *parson*, *calc*, *cgi_decode*; as well as one C++ subject, *calccpp*. The first three C subjects were also evaluated by Mimid. For each of these subjects, as well as for *cgi_decode*, Stalagmite reuses the golden grammars of Mimid. Moreover, as *cjson* and *parson* are also JSON parsers, the Mimid golden grammar for JSON is used again. For *calc* and *calccpp*, the golden grammar was written by hand. For *lisp*, the golden grammar was taken from the ANTLR4 GitHub repository. Precision and recall are computed similarly to Mimid, except that duplicate inputs are pruned to ensure more diversity.

Data Collection

We collected the mined input grammars and golden grammars from the GitHub repository [92] of the paper. Stalagmite does not require training inputs, as it statically analyzes the code of the program under test. Moreover, Stalagmite already uses the FuzzingBook grammar format.

Table 4.5: Grammar accuracy recomputed using GRAMACCU

Subject	P_{orig}	P_{k1}	P_{k2}	P_{k3}	P_{k4}	P_{k5}	$Diff_P$	R_{orig}	R_{k1}	R_{k2}	R_{k3}	R_{k4}	R_{k5}	$Diff_R$
Arvada														
arith	1	1	1	1	1	1	0	1	1	1	1	1	1	0
curl	0.55	0.85	0.78	0.71	0.65	0.6	0.05	0.92	0.33	0.34	0.37	0.44	0.45	−0.47
fol	1	1	1	1	1	1	0	0.87	0.98	0.97	0.95	0.94	0.93	0.06
json	0.95	0.99	0.98	0.92	0.87	0.84	−0.11	1	1	1	1	1	1	0
lisp	0.9	0.99	0.98	0.98	0.95	0.94	0.04	0.52	0.79	0.74	0.54	0.38	0.35	−0.17
mathexpr	0.97	0.97	0.98	0.96	0.93	0.91	−0.06	0.84	0.94	0.94	0.91	0.88	0.85	0.01
tinyc	0.73	0.75	0.63	0.58	0.51	0.46	−0.27	0.92	0.95	0.93	0.94	0.96	0.96	0.04
turtle	1	1	1	1	1	1	0	1	1	1	1	1	1	0
while	1	1	1	1	1	1	0	0.7	0.87	0.86	0.77	0.69	0.62	−0.08
xml	0.98	1	1	1	1	1	0.02	0.96	0.94	0.93	0.94	0.94	0.94	−0.02
Treevada														
curl	0.89	0.94	0.87	0.8	0.75	0.7	−0.19	0.6	0.23	0.25	0.26	0.37	0.36	−0.24
json	0.96	1	1	1	0.98	0.97	0.01	0.9	0.97	0.97	0.93	0.88	0.88	−0.02
lisp	0.98	0.97	0.99	0.99	0.99	0.99	0.01	1	1	1	1	1	1	0
tinyc	0.86	0.92	0.92	0.88	0.86	0.85	−0.01	0.97	1	0.99	0.99	0.99	0.99	0.02
turtle	0.94	0.98	0.94	0.86	0.87	0.83	−0.11	1	1	1	1	1	1	0
while	1	1	1	1	1	1	0	1	1	1	1	1	1	0
xml	1	1	1	1	1	1	0	1	1	1	1	1	1	0
Kedavra														
arith	1	1	1	1	1	1	0	1	1	1	1	1	1	0
curl	1	1	1	0.99	1	0.99	−0.01	0.11	0	0	0	0	0	−0.11
fol	0.99	0.64	0.88	0.82	0.82	0.83	−0.16	0.55	0.94	0.9	0.89	0.82	0.75	0.2
json	1	1	0.99	0.99	1	1	0	0.97	0.99	0.98	0.98	0.99	0.99	0.02
lisp	1	1	1	1	1	1	0	1	1	1	1	1	1	0
mathexpr	0.94	0.79	0.8	0.71	0.57	0.35	−0.59	0.92	0.98	0.96	0.96	0.96	0.94	0.02
minic	1	0.97	0.98	0.95	0.97	0.98	−0.02	0.48	0.13	0.12	0.05	0.02	0	−0.48
tinyc	1	0.96	0.9	0.65	0.46	0.31	−0.69	1	0.19	0.16	0.08	0.03	0.01	−0.99
turtle	1	0.98	0.95	0.9	0.85	0.82	−0.18	1	1	1	1	1	1	0
while	1	1	1	1	1	1	0	1	1	1	1	1	1	0
xml	1	1	1	1	1	1	0	1	1	1	1	1	1	0
Mimid														
json	1	1	1	1	1	1	0	0.84	0.63	0.59	0.55	0.49	0.47	−0.37
mjs	0.95	0.96	0.91	0.9	0.9	0.9	−0.05	0.96	0.6	0.55	0.52	0.31	0.13	−0.83
calc.py	1	1	1	1	1	1	0	1	1	1	1	1	1	0
cgidecode.py	1	1	1	1	1	1	0	1	1	1	1	1	1	0
mathexpr.py	0.88	0.98	0.96	0.95	0.96	0.97	0.09	0.93	0.49	0.51	0.66	0.82	0.8	−0.13
microjson.py	0.99	0.99	1	0.99	0.99	0.98	0	0.93	0.41	0.44	0.36	0.43	0.44	−0.49
sexpr.py	0.99	0.98	0.98	0.99	0.98	0.97	−0.02	0.81	0.67	0.7	0.67	0.67	0.65	−0.15
urlparse.py	1	1	1	1	1	1	0	0.96	0.97	1	0.95	0.83	0.81	−0.16
tinyc	1	1	1	1	1	1	0	0.93	1	1	0.95	0.94	0.92	−0.01
Stalagmite														
calc	1	1	1	1	1	1	0	1	1	1	1	1	1	0
cgi_decode	1	0.99	0.99	1	1	1	0	1	1	1	1	1	1	0
cjson	0.99	0.99	0.99	0.99	0.99	0.99	0	1	1	1	1	1	1	0
json	1	1	1	1	1	1	0	1	0.98	1	0.99	0.99	0.98	−0.02
lisp	1	1	1	1	1	1	0	1	1	1	1	1	1	0
mjs	0.65	0.56	0.41	0.33	0.33	0.29	−0.36	0.84	0.9	0.36	0.42	0.35	0.21	−0.64
parson	0.87	0.94	0.91	0.65	0.54	0.55	−0.32	0.81	0.78	0.8	0.7	0.71	0.7	−0.11
calccpp	1	1	1	1	1	1	0	1	1	1	1	1	1	0
tinyc	1	0.99	0.98	0.99	0.99	0.99	−0.01	1	1	1	1	1	1	0

4.5.2 RQ1: Revisiting Grammar Accuracy

In this research question, we re-evaluate the precision and recall values of the five approaches using GRAMACCU. Specifically, for each subject previously assessed by the respective approaches, we compute precision and recall as described in Algorithm 5 with $kLimit = 5$. We also report the original precision and recall values as reported in the respective papers. The results are shown in Table 4.5. The $Diff_P$ ($Diff_R$) column shows the difference between the reported precision (recall) and the GRAMACCU-computed precision (recall) with $k = 5$. Differences are highlighted in blue if the GRAMACCU value with $k = 5$ is either better than the reported value or if the reported value is at most 10% worse. Differences are marked in orange, indicating that the GRAMACCU precision or recall values are significantly lower (i.e., more than 10%) than reported values.

For Arvada, we see that precision and recall are significantly lower for 2 out of 10 subjects. For Treevada, precision and recall are also significantly lower than originally reported, specifically on 2 and 1 out of 7 subjects, respectively. Arvada's accuracy drops for *curl*, *json*, *lisp* and *Tiny-C*, whereas Treevada's accuracy drops for *curl* and *turtle*. The most significant drops are observed for *curl* and *Tiny-C*, which are both rather large and complicated grammars. For Kedavra, in most cases, we find that the reported precision and recall values withstand our evaluation, with 5 out of 11 scoring (near-)perfect accuracy levels. We note that Kedavra identified only the two subjects, *Tiny-C* and *minic*, as being *non-space sensitive*. Space sensitivity is a specific configuration parameter of the grammar output format of Kedavra, implying that the grammars generated for these two subjects do not include any whitespaces. Upon manual inspection, we found these grammars to be of good quality. Since it is unclear how to re-introduce whitespaces into these grammars in an unbiased manner, we decided to leave them unchanged for the evaluation. It is important to note that the actual accuracy for these two subjects may be higher than reported in Table 4.5, depending on how non-space sensitivity is handled. For the white-box approach Mimid, we see that precision remains stable for all subjects, while recall drops significantly for 6 out of 9 subjects. Finally, for Stalagmite, precision and recall only drops significantly for the *mjs* and *parson* subjects.

All compared approaches are challenged by our evaluation, with the latest approaches, Kedavra (black-box) and Stalagmite (white-box), performing best.

In addition to comparing reported and reproduced accuracy, we also investigate how well current evaluation practices cover the *deeper* parts of grammars. We begin by empirically validating that the deeper parts of the "million-dollar grammar" (introduced in Section 4.2.2) are not explored by prevalent input generation techniques. To do so, we translated this grammar into the respective formats required by the five studied approaches, and used their original input sampling implementations to generate 1,000,000 inputs. The results show that, without exceptions, all inputs sampled by all approaches consist solely of digits—none include non-digit characters allowed by the `<any>` non-terminal. This demonstrates that prevalent input generation techniques are unable to reach deep parts of the "million-dollar grammar".

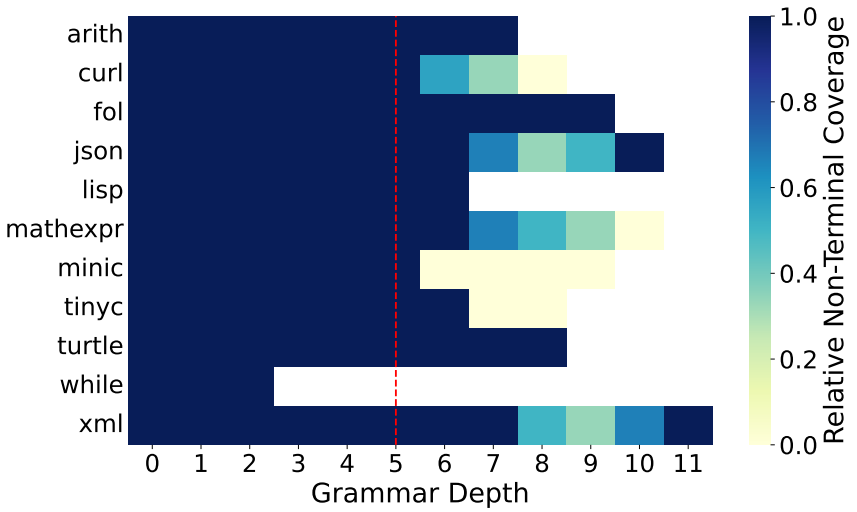


Figure 4.6: Heat map showing the depth reach of the prevalent sampling technique for the golden grammars of Kedavra. We aggregate all non-terminal symbols that can be reached with a minimum path length of "Grammar Depth" from the start non-terminal, and count how many of these non-terminals were covered at least once by 1,000 generated inputs.

However, since this grammar was artificially constructed, we further investigate to what extent this phenomenon occurs in grammars employed by the grammar mining approaches under study. For this, we compute the shortest path from the start symbol to each non-terminal in a given grammar. We then generate 1,000 inputs from the grammar using the prevalent sampling algorithm. Because implementations of this algorithm vary only slightly across approaches, we consistently use the version from Mimid (and Stalagmite), modifying only the `max_depth` parameter—set to 5 for black-box and 10 for white-box approaches, in line with their original configurations. Next, we assess which non-terminals are actually covered by at least one generated input. We illustrate this for the golden grammars of Kedavra (which includes those used by Arvada and Treevada) in Figure 4.6. Our findings show that all non-terminals reachable within a depth of 5 are covered. Beyond this depth, coverage drops significantly. Some deeper non-terminals are occasionally reached due to the need for the sampling algorithm to "close off" an expansion after reaching the depth limit, which sometimes requires additional productions to reach a terminal.

Extending the analysis beyond Kedavra, we observe that in every approach—across both mined and golden grammars—at least one subject grammar exhibits insufficient coverage beyond the configured `max_depth` when using the prevalent sampling technique.

4.5.3 RQ2: Golden Grammar Coverage

In this research question, we investigate how many k-paths in the golden grammar can be covered by inputs generated from the mined grammars using GRAMACCU. This metric is important as it goes beyond binary recall results ("the golden grammar can (or cannot) parse an input generated by the mined grammar") and shows how well the mined grammar captures the syntactic features of the golden grammar. To realize this measurement,

1. we generate inputs using GRAMACCU from the mined grammars for $k = 1, 2, 3$;
2. we try to parse each input with the golden grammar;
3. we count how many distinct k-paths are contained in the corresponding parse trees;
4. we also measure the number of total k-paths in the golden grammar, and compute the ratio of covered to total k-paths.

During our analysis, we identified some minor issues with the golden grammars of *Tiny-C*, *lisp*, and *mjs*, which we addressed for this research question. For *Tiny-C*, the golden grammar incorrectly included an obligatory space between tokens, while the program under test accepts an arbitrary number of whitespaces, including none. For *lisp*, the golden grammar contained an epsilon production rule under the whitespace non-terminal at a problematic position, causing our Earley parser-based implementation to hang. We made a slight modification to this golden grammar to avoid this issue while maintaining its equivalence. For *mjs*, the very large golden grammar also contained several epsilon productions, which proved challenging to rewrite and difficult for an Earley parser to handle. As a result, we introduced a parse timeout of one second and excluded any inputs that could not be parsed within this time frame.

The results of this measurement are shown in Table 4.7. We highlight k-path coverages in bold if the corresponding approach achieved the highest value among all approaches evaluated for this subject.

Note that for black-box approaches, the golden grammar typically acts as the parse oracle, exactly matching the language that these approaches aim to learn. In contrast, for white-box approaches, the oracle is the program under test, which is expected to accept a language similar to the golden grammar. However, it is possible that there are differences in certain details, such as the program allowing trailing commas in a list, while the golden grammar does not [23]. As a result, achieving 100% k-path coverage may not be feasible for white-box approaches, as some inputs accepted by the golden grammar might not be accepted by the program under test. For black-box approaches, on the other hand, the goal should be to achieve 100% k-path coverage. In both cases, it is important to compare approaches that were evaluated on the same subjects. In the following, we discuss our findings in more detail.

Table 4.7: Golden 1-,2-,3-paths covered by mined grammars

Subject	1-paths			2-paths			3-paths		
	Tot	Cov	Rel	Tot	Cov	Rel	Tot	Cov	Rel
Arvada									
arith	31	**31**	**1**	55	**55**	**1**	78	**75**	**.96**
curl	185	**122**	**.66**	422	**171**	**.41**	859	**241**	**.28**
fol	124	**124**	**1**	167	**167**	**1**	494	**432**	**.87**
json	116	**116**	**1**	161	158	.98	209	198	.95
lisp	57	**57**	**1**	71	**71**	**1**	127	95	.75
mathexpr	136	**134**	**.99**	268	263	.98	386	360	.93
tinyc	65	64	.98	113	112	.99	532	516	.97
turtle	95	**95**	**1**	115	**115**	**1**	166	**166**	**1**
while	24	**24**	**1**	52	**52**	**1**	295	265	.9
xml	126	**126**	**1**	154	**154**	**1**	214	**214**	**1**
Treevada									
curl	185	118	.64	422	161	.38	859	225	.26
json	116	**116**	**1**	161	**161**	**1**	209	**208**	**1**
lisp	57	**57**	**1**	71	**71**	**1**	127	**127**	**1**
tinyc	65	64	.98	113	112	.99	532	**524**	**.98**
turtle	95	**95**	**1**	115	**115**	**1**	166	**166**	**1**
while	24	**24**	**1**	52	**52**	**1**	295	**270**	**.92**
xml	126	**126**	**1**	154	**154**	**1**	214	212	.99
Kedavra									
arith	31	**31**	**1**	55	**55**	**1**	78	**75**	**.96**
curl	185	98	.53	422	109	.26	859	132	.15
fol	124	**124**	**1**	167	**167**	**1**	494	313	.63
json	116	**116**	**1**	161	**161**	**1**	209	**208**	**1**
lisp	57	**57**	**1**	71	**71**	**1**	127	**127**	**1**
mathexpr	136	**134**	**.99**	268	**266**	**.99**	386	**362**	**.94**
minic	102	**74**	**.73**	282	**212**	**.75**	2040	**1097**	**.54**
tinyc	65	62	.95	113	109	.96	532	475	.89
turtle	95	**95**	**1**	115	**115**	**1**	166	**166**	**1**
while	24	**24**	**1**	52	**52**	**1**	295	**270**	**.92**
xml	126	**126**	**1**	154	**154**	**1**	214	**214**	**1**
Mimid									
json	131	112	.85	199	164	.82	271	227	.84
mjs	787	130	.17	2473	167	.07	13631	201	.01
calc.py	24	**23**	**.96**	36	**33**	**.92**	111	**102**	**.92**
cgidecode.py	99	**99**	**1**	100	**100**	**1**	104	**104**	**1**
mathexpr.py	81	**32**	**.4**	102	**46**	**.45**	273	**88**	**.32**
microjson.py	127	**121**	**.95**	210	**179**	**.85**	341	**250**	**.73**
sexpr.py	83	**65**	**.78**	101	**83**	**.82**	261	**194**	**.74**
urlparse.py	36	**36**	**1**	42	**42**	**1**	56	**55**	**.98**
tinyc	65	63	.97	113	108	.96	532	418	.79
Stalagmite									
calc	29	**29**	**1**	45	**45**	**1**	144	**142**	**.99**
cgi_decode	100	**67**	**.67**	101	**68**	**.67**	106	**69**	**.65**
cjson	131	**121**	**.92**	199	**177**	**.89**	271	**214**	**.79**
json	131	**125**	**.95**	199	**183**	**.92**	271	**236**	**.87**
lisp	78	**51**	**.65**	104	**59**	**.57**	195	**68**	**.35**
mjs	787	**206**	**.26**	2473	**295**	**.12**	13631	**451**	**.03**
parson	131	**131**	**1**	199	**198**	**.99**	271	**265**	**.98**
calccpp	29	**29**	**1**	45	**45**	**1**	144	**142**	**.99**
tinyc	65	**65**	**1**	113	**113**	**1**	532	522	.98

On *lisp*, both Treevada and Kedavra achieve perfect 3-path coverage, whereas Arvada only reaches 75%. For *curl*, none of the black-box approaches achieve perfect k-path coverage, but both Arvada and Treevada consistently outperform Kedavra. On *fol*, Arvada outperforms Kedavra with regards to 3-path coverage.

The two white-box approaches share the three subjects *Tiny-C*, *mjs*, and *json*. On all of these subjects, Stalagmite consistently outperforms Mimid. However, both approaches yield poor results on *mjs*, suggesting that this language could not be learned effectively.

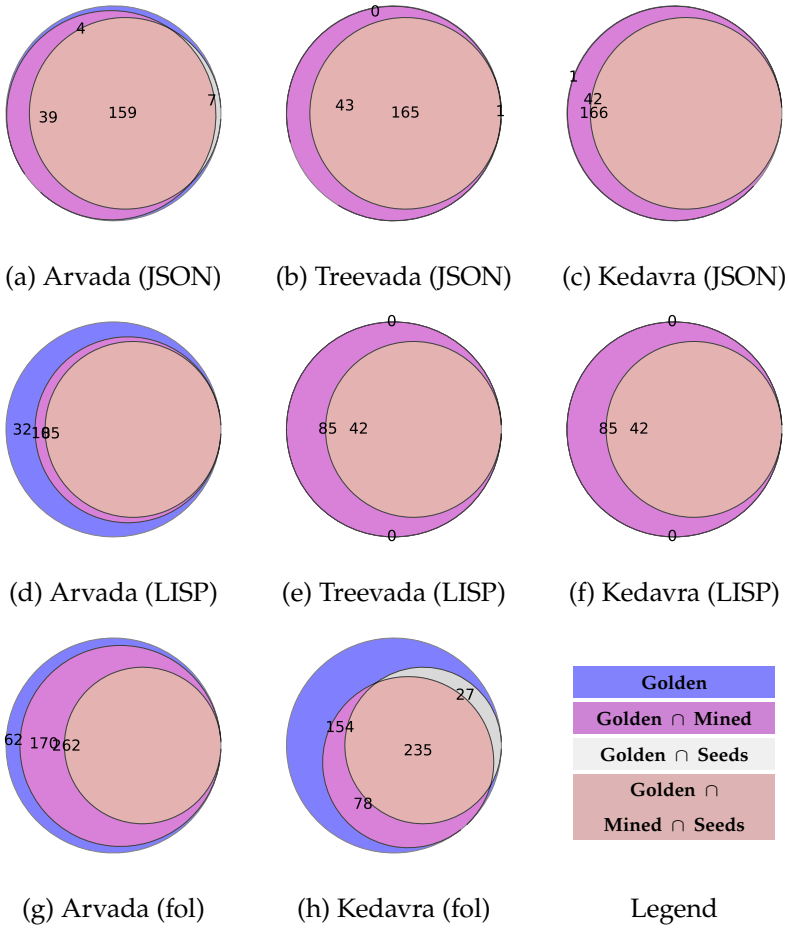
Black-box approaches generally achieve high k-path coverage in the golden grammar, although some subjects remain a challenge. Stalagmite consistently outperforms Mimid on the shared subjects.

4.5.4 RQ3: Qualitative Analysis of Black-Box Approaches

In this research question, we extend RQ2 by not only measuring the coverage of k-paths in the golden grammar achievable by inputs generated from the mined grammar, but also examining which k-paths are already present in the training samples. This enables us to assess how effectively the language features from the training inputs have been learned and generalized by the mined grammar. Additionally, by analyzing *which* k-paths were not learned, we can identify the language features that were not learned. We focus solely on black-box approaches because they include training samples as part of their reproducibility package.

We discuss our findings on three selected subjects, *json*, *lisp* and *fol*, shown in the Venn diagrams of Figure 4.8. Since Treevada was not evaluated on *fol*, we show data only for Arvada and Kedavra. For *json*, all approaches are able to generalize well beyond the training inputs, with mined grammars covering 39 (Arvada) to 43 (Treevada) additional 3-paths. However, Arvada fails to learn seven 3-paths already present in the sample inputs. Moreover, Arvada misses four 3-paths in the golden-grammar, whereas Kedavra only misses one and Treevada misses none. For *lisp*, both Treevada and Kedavra incorporate all 3-paths of the sample inputs into their mined grammar and generalize well, covering all 3-paths in the golden grammar. While Arvada also incorporates all 3-paths of the sample inputs into the mined grammar, it generalizes poorly and misses 32 3-paths in the golden grammar. An outlier is the *fol* subject, where Arvada covers all 3-paths present in the training samples and generalizes beyond them, while Kedavra fails to learn all the training sample 3-paths and generalizes worse than Arvada. However, neither approach learns all 3-paths of the golden grammar, with Arvada still performing better.

This set representation of k-paths also lets us identify language features not learned by the mined grammars. For instance, consider the following excerpt from the golden grammar of *lisp*:



(a) Arvada (JSON) (b) Treevada (JSON) (c) Kedavra (JSON)

(d) Arvada (LISP) (e) Treevada (LISP) (f) Kedavra (LISP)

(g) Arvada (fol) (h) Kedavra (fol) Legend

Figure 4.8: Venn diagrams showing covered 3-paths

```
<start> ::= <sexpr>
<sexpr> ::= "(" <sexpr> <E> "." <E> <sexpr> ")" | <atomicsymbol>
```

In this case, the grammar mined by Arvada fails to cover the 3-path $(sexpr^0 \rightarrow sexpr^1 \rightarrow sexpr^1)$ in the golden grammar. In practical terms, this means that it cannot generate nested S-expressions like `"((a . b) . c)"`, which are required to cover that 3-path. However, it can generate simpler, non-nested S-expressions such as `"(a . b)"`. Our k-path based metric allows to identify such issues at a much more granular level than precision and recall, and hence enables researchers to improve their grammar mining techniques.

K-path-based grammar assessment enables detailed analyses of the language features learned by grammar mining approaches, as well as their ability to generalize beyond the training inputs.

4.5.5 RQ4: Growth of k-Paths

In this research question, we examine how the number of k-paths grows as k increases. This analysis is essential for evaluating the feasibility of assessing input grammar accuracy using GRAMACCU, which produces a varying number of inputs based on both the k parameter and the assessed grammar.

We exemplarily show the growth of k-paths, which is proportional to the runtime of GRAMACCU, for the grammars mined by Kedavra. In principle, this analysis could be performed for all approaches, with similar exponential results, but we limit our evaluation to Kedavra for brevity. To better understand the runtime effort of GRAMACCU, we also measure the time it takes to generate inputs for all k-paths, as well as the time it takes to parse these inputs with the golden grammar. The results are shown in Figure 4.9.

We observe that the number of k-paths grows exponentially with k as stated in the original paper [54], and may quickly surpass the constant number of 1,000 inputs used by the previous evaluations.

Although we restrict our evaluation to $k \leqslant 5$ to maintain a manageable number of inputs, we believe that significantly higher k values could still be utilized for assessing grammar accuracy, for instance, by sampling a fixed number of inputs for each k. We note that this exponential growth is by no means unexpected or problematic; it is inherent to context-free grammars. Limiting k to a small number allows for measuring grammar accuracy more comprehensively than with an unsystematic metric.

Despite the exponential growth of k-paths, GRAMACCU enables a systematic assessment of grammar accuracy by allowing k to be set to a manageable value.

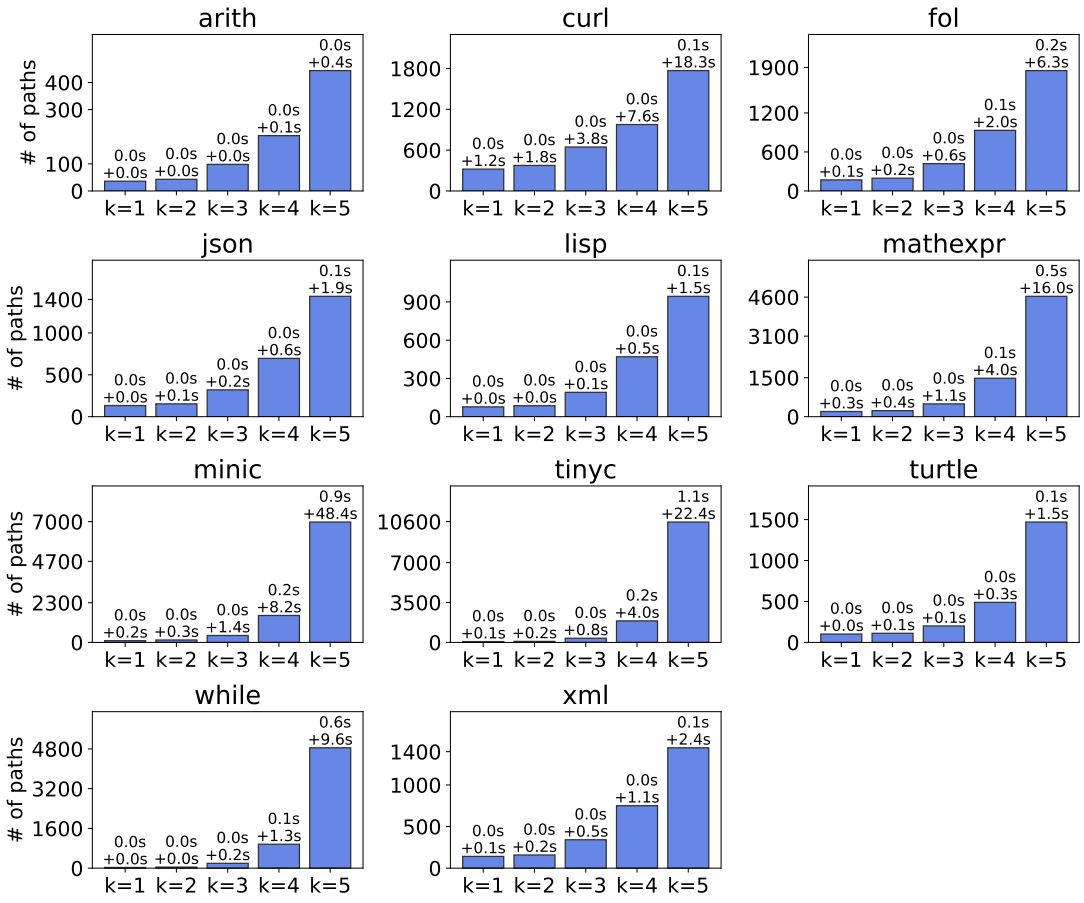


Figure 4.9: Growth of k-paths in Kedavra-mined grammars. Top numbers above bars indicate time (seconds) to generate all k-path inputs; bottom numbers show time to parse them with a Python Earley parser using the golden grammar.

4.5.6 RQ5: Added Value of k > 1

In this research question, we analyze whether considering k-paths with k > 1 provides additional value. Notably, enumerating all (k+1)-paths inherently includes all k-paths—possibly multiple times. This raises a question: does using k-paths with k > 1 simply reinforce the same "faulty" 1-paths when estimating grammar accuracy, or does it uncover new issues?

To investigate, we conduct the following experiment:

1. Enumerate all k-paths (k = 1, 2, 3) from the current grammar.
2. Generate 10 random inputs per k-path using GRAMACCU.
3. Parse-check each input with the antagonist grammar (i.e., the golden grammar when generating from the mined grammar, and vice versa).
4. Classify a k-path as "faulty" if none of its 10 derived inputs are accepted. (If only some inputs fail, the fault is due to another unrelated k-path in these inputs.)
5. Filter "faulty" k-paths for k > 1 by excluding those that contain a previously identified "faulty" j-path with j < k. This ensures that we count only genuinely new "faulty" k-paths that depend on the current length k.

We illustrate this in Table 4.10 for the recall estimation of Treevada. For *json*, only one faulty 2-path appears: $(\texttt{list}^0 \rightarrow \texttt{value}^2)$, as seen in inputs like `"[null]"` or `"[{}]"`. While both 1-paths $(\texttt{list})$ and $(\texttt{value})$ are accepted individually (for some contexts), *a list containing a single value* is never accepted by the mined grammar. In *Tiny-C*, the single faulty 2-path is $(\texttt{term}^0 \rightarrow \texttt{paren\_expr}^4)$, observed in inputs like `"( h < p );"`. This issue does not appear in 1-paths, as parenthesized expressions may also occur in the context of *if or while conditions*, such as `"if ( h < p ) a = 1 ;"`, which were learned correctly. However, Treevada fails to recognize them in the context of standalone *terms*.

This shows that certain faults only emerge in inputs containing k-paths with $k > 1$. While inputs generated from 1-paths may include longer k-paths by chance, systematically exploring k-paths with $k > 1$ is more reliable for uncovering these faults.

Considering k-paths with $k > 1$ *provides additional value as some faults inherently depend on longer k-paths.*

4.6 Related Work

To the best of our knowledge, no prior work has independently evaluated input grammar mining approaches using a systematic metric. Our study fills this gap and provides a baseline for future research. There exists one prior work that compares context-free

Table 4.10: Benefit of $k > 1$ for Treevada (Recall). Columns *T* list the total number of k-paths, *F* the number of faulty ones, and *P* the number after pruning based on shorter faulty k-paths.

Subject	1-paths			2-paths			3-paths		
	T	F	P	T	F	P	T	F	P
curl	310	195	195	422	250	0	859	481	3
json	149	0	0	161	1	1	209	8	0
lisp	59	0	0	71	0	0	127	0	0
tinyc	64	0	0	86	1	1	334	4	0
turtle	100	0	0	115	0	0	166	0	0
while	25	0	0	52	0	0	295	0	0
xml	138	0	0	154	0	0	214	0	0

grammars to detect (non-)equivalence by generating inputs that satisfy coverage criteria such as non-terminal or production coverage [46]. In contrast to our work, that study does not take deeper context into account as we do with k-paths, and the evaluation is not focused on input grammar miners.

While we re-evaluated the grammars inferred by the five state-of-the-art approaches [64, 5, 67, 52, 23] in context-free input grammar mining, we briefly discuss other related work in the field. GLADE [18] was one of the first approaches to mine input grammars in a black-box setting, although its results were later challenged [19]. Building on GLADE, REINAM [119] uses reinforcement learning to infer probabilistic context-free input grammars. Early contributions to white-box input grammar mining include the work of Lin et al. [68, 69]. Recently, alternative directions in grammar mining have emerged, including attribute grammar mining [86] and static inference of regular expressions [96, 97]. Beyond text-based languages, BIEBER [37] targets binary parsers, inferring parsers for formats such as WAV and BMP from instrumented program executions.

4.7 Threats to Validity

Internal Validity. The validity of an empirical study as ours may be threatened by errors in the implementation of the evaluation. We mitigate the risk by building on GrammarGraph[3], an existing implementation for computing k-paths, which was used by prior work [106]. Additionally, our grammar converters may have introduced errors. We minimized this risk by using existing tools for ANTLR4 grammars[4] and by manually verifying the grammars for black-box approaches.

[3] `https://github.com/rindPHI/GrammarGraph`
[4] `https://github.com/vrthra/antlr2json`

We also ensured that the original parsers of the respective approaches could parse all inputs generated from our converted grammars.

Generalizability. Our empirical study re-evaluates five existing grammar mining approaches on the same benchmarks as their original evaluations but with a new, more systematic metric. While we believe that our selection of grammar mining approaches is representative of the state-of-the-art, our findings do not necessarily generalize to all other grammar mining approaches and benchmarks.

Construct Validity. Like all grammar mining evaluations, ours relies on a golden grammar as ground truth. For the black-box approaches we evaluate, this holds true, as the golden grammar is the parse oracle. For the white-box approaches, however, it only approximates the language accepted by the program under test and may not be entirely accurate, potentially impacting the results. This limitation is shared by all white-box grammar mining evaluations.

4.8 Conclusion

Input grammars are highly useful in contexts such as test input generation and reverse engineering. The field of input grammar mining aims to automatically infer input grammars from programs. The primary metric for assessing the quality of mined input grammars is grammar accuracy, measured in terms of precision and recall.

In this study, we show that the prevalent technique for measuring grammar accuracy is biased and propose a systematic technique, GRAMACCU, based on k-path coverage [54]. By re-evaluating input grammars produced by five recent approaches, we identify significant drops in precision and recall compared to the originally reported results. We advocate adopting metrics like ours to improve the reliability of grammar accuracy assessments and better judge the usefulness of input grammar miners. To foster future research, we release our GRAMACCU implementation as open source.

4.9 Data Availability

Our reproducible evaluation is available at:

`https://doi.org/10.5281/zenodo.17395623`

5 Input Constraints and Execution Goals

This chapter is taken, either directly or with minor modifications, from our 2026 ICST paper *Combining Input Constraints with Execution Goals*.
My contribution in this work is as follows:
① contributions to original idea,
② implementation, and
③ evaluation.
The contribution of my co-author, Marius Smytzek, is as follows:
① contributions to original idea.

Chapter Overview

Recent advances in black-box test generation (fuzzing) allow combining syntax specifications (grammars) with *semantic constraints* over grammar elements. These constraints not only help in producing valid inputs ("⟨*checksum*⟩ should be a Luhn checksum over ⟨*number*⟩") but also allow specifying *testing goals* ("⟨*payload*⟩ should be as short as possible"). Such constraints give testers unprecedented control over the generated inputs. However, constraint-based fuzzers like ISLa or Fandango have only been limited to *input features* and black-box fuzzing.

This chapter shows how to extend this concept with *execution constraints* relating to the program under test. We introduce a set of *primitives* that check for specific execution features such as coverage of specific locations, memory usage, execution time, and more. Testers can then use these to specify additional testing goals: "I want an input that is (1) valid, (2) as short as possible, and (3) results in a maximum of code coverage."

We implement our approach on top of the Fandango fuzzer, finding that its evolutionary algorithm efficiently finds solutions that satisfy input and execution constraints. We demonstrate that the combination of input and execution constraints allows us to specify and solve approaches such as directed fuzzing, performance fuzzing, and coverage-guided fuzzing evolving individuals and populations alike, effectively covering the entire range of testing goals from white-box to black-box fuzzing in a unified approach.

5.1 Introduction

Traditionally, the world of software testing has been split into two camps. *Black-box tests* are derived from a *specification*, which exists independently of the program under test; the aim is to ensure that all aspects of the specification are covered. *White-box tests* are derived from the *implementation*, which is known to the tester; here, the aim is to cover the *implemented* features. Both paradigms are needed, particularly as an implementation may not fully satisfy its specification, and an abstract specification may not cover all concrete details.

As it comes to *generating* test inputs automatically, though, the vast majority of approaches focus on the white-box paradigm. That is because the program under test is (1) always available by definition, and (2) comes in a form that (at least in principle) allows for analyzing the program's code and executions. This concept contrasts with black-box test generation, which needs a formal specification of the program's input language. While *context-free grammars* are a common and proven way to specify input languages, they are not expressive enough to capture all aspects of a program's input—notably, *semantic* aspects such as checksums, length constraints, references, and other properties that cannot be expressed in a context-free grammar.

In recent years, a new generation of black-box test generation tools has emerged that allows users to specify *semantic constraints* over the grammar. These constraints come as logical formulas over grammar elements. In the grammar

$$\langle \textit{start} \rangle ::= \langle \textit{credit-card} \rangle \tag{5.1}$$

$$\langle \textit{credit-card} \rangle ::= \langle \textit{number} \rangle \langle \textit{checksum} \rangle \tag{5.2}$$

an input constraint such as

$$\langle \textit{checksum} \rangle = \text{luhn}\big(\langle \textit{number} \rangle\big) \wedge \big|\langle \textit{number} \rangle\big| < 20 \tag{5.3}$$

expresses that the $\langle \textit{checksum} \rangle$ digit should be a Luhn checksum over $\langle \textit{number} \rangle$, and the remaining $\langle \textit{number} \rangle$ should have fewer than 20 characters. Test generators such as ISLa [106] or Fandango [122] allow users to specify such constraints, and then generate inputs that satisfy them.

Such constraints are useful for generating valid inputs and specifying *testing goals* over the generated inputs. If a tester wants to test *Visa* cards, a constraint such as $\langle \textit{credit-card} \rangle$.startswith(`'4'`) ensures the number starts with a 4; likewise, a constraint such as $\big|\langle \textit{credit-card} \rangle\big| = 2$ will put some length checks to the test. In Fandango, testers can use any Python function to specify such constraints, giving them considerable flexibility in specifying the input language. To satisfy the constraints and produce inputs, Fandango uses robust constraint-aware evolutionary algorithms.

While most white-box test generators uniquely focus on *increasing coverage* in the program under test, some tools also provide flexibility in specifying testing goals. Specialized fuzzers such as SlowFuzz [83] or PerfFuzz [66] focus on performance-related goals

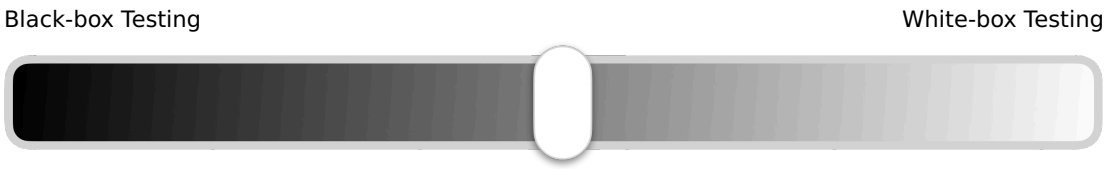


Figure 5.1: FANDANGOGREY flexibly combines (black-box) input constraints and (white-box) execution goals.

rather than code coverage, such as maximizing the execution time of a program. FuzzFactory [78] provides a framework for composing multiple domain-specific fuzzing strategies on top of AFL. **However, these tools do not allow control of the input language; they do not allow specifying *valid* inputs, and they cannot use the tester's *domain knowledge* on where the bugs might be.**

In this chapter, we propose a new approach combining the best of the black-box and white-box worlds, empowering testers to *specify both input constraints and execution goals in a single framework.* Specifically, we extend Fandango with *additional constraint primitives* that refer to the execution domain, thus allowing an arbitrary mix of input constraints and execution goals. This integration enables the use of grey-box fuzzing techniques—including coverage-guided fuzzing, domain-specific feedback-driven fuzzing, and combinations thereof—covering the entire range of approaches from black-box to white-box fuzzing in a single framework (Fig. 5.1).

Our approach reuses the evolutionary algorithm of Fandango in a two-phase process: it first generates inputs that satisfy user-defined input constraints, and then *evolves* those valid inputs to optimize execution goals based on feedback from the program under test. Users can thus declare and arbitrarily combine input constraints and execution goals using the complete flexibility of Python.

Let us illustrate this with a simple example. Let us assume we want to determine which credit card number takes the longest to process for validity. With our approach, we can express this using the grammar in Eqs. (5.1) and (5.2), the input constraint in Eq. (5.3), and the execution goal:

$$\textbf{maximizing}\ \mathrm{DynamicAnalysis}(\langle \mathit{start} \rangle).\mathrm{ExecutionPathLength}() \tag{5.4}$$

Here, the keyword **maximizing** indicates a *soft constraint*—a value we want to maximize. *DynamicAnalysis* is a primitive that collects execution feedback from the program under test; Overall, this goal states that we want an input that satisfies the checksum constraint and maximizes the execution path length when executed. Since long executions tend to be trivially produced by long inputs, we could also add another execution goal that we want the input to be as short as possible:

$$\textbf{minimizing}\ |\langle \mathit{start} \rangle| \tag{5.5}$$

We pose that such specifications are accessible even for non-expert users; they are far more expressive, readable, maintainable, and combinable than the specialized code of existing domain-specific fuzzers. We demonstrate the practicality and versatility of our approach through a series of experiments on real-world programs implemented in C and C++. Specifically, we implement five domain-specific fuzzing scenarios, including directed fuzzing, performance fuzzing, and coverage-guided fuzzing. These goals can be expressed in our unified framework with one to two lines of specification and efficiently solved by the Fandango evolutionary algorithm.

In summary, our main contributions are:

1. We present a novel unified approach that allows for *specifying execution goals in conjunction with input constraints,* enabling rich grey-box fuzzing applications within the Fandango framework.

2. We integrate *execution feedback capabilities* into the Fandango tool, interfacing with the instrumented program under test and allowing Fandango to optimize non-normalized feedback values. We release this extension as open-source software.

3. We provide a *catalog of domain-specific execution goals,* including *code coverage*, *function coverage*, *basic-block distance*, *execution path length*, and *heap memory usage*.

4. We present and open-source our *LLVM-based instrumentation and analysis passes,* wrapped in a compiler replacement called `fcc`. This tool outputs instrumented binaries and static analysis information as JSON files.

5. We show that our approach can equally evolve and produce individual *inputs* just as *populations* that fulfill a particular goal, as required by test settings.

6. We demonstrate the effectiveness of our approach through five domain-specific *fuzzing scenarios,* showcasing its versatility and practicality.

The remainder of this chapter is structured as follows: Section 5.2 reviews key concepts in language-based testing and modular fuzzing. Section 5.3 introduces our unified approach, FANDANGOGREY, combining language-based testing with execution feedback. Section 5.4 describes the implementation of FANDANGOGREY, including our LLVM-based instrumentation framework and the extensions made to Fandango. Section 5.5 showcases five domain-specific fuzzing scenarios, demonstrating the practicality and versatility of our approach on real-world programs, followed by potential threats to the validity of our results in Section 5.6. Section 5.7 presents an outlook on future directions, before we conclude in Section 5.8.

5.2 Background

5.2.1 Language-Based Software Testing

Language-based software testing [107] has been proposed as a new way to generate test inputs, satisfying *grammars* (for specifying input syntax) and *constraints* (for specifying input semantics) to specify input languages. The first implementation of this methodology is the ISLa framework [106], using SMT solvers to satisfy the constraints. Although this enabled new capabilities in small-scale settings, it did not generalize well to many real-world input formats, primarily because numerous constraints could not be expressed within the limited SMT-LIB language supported by the solver.

The recent Fandango tool [122] shares the same goal as ISLa but is significantly more flexible in terms of the input constraints it supports—in fact, it allows users to specify *arbitrary semantic constraints* using the full expressiveness of Python. Its key innovation lies in employing *search-based testing*—an algorithm that evolves inputs towards satisfying increasingly more specified input constraints, retaining and mutating the *fittest* candidates.

This approach implies that input generation only needs *checking whether input candidates satisfy the constraints*, rather than requiring them to be expressed in the rigid language supported by SMT solvers. Some constraints, such as checksums, are difficult to satisfy through such a *generate-and-check* approach. Fandango, therefore, also allows attaching imperative generator functions to specific grammar non-terminals. For example, a generator might compute a checksum for one non-terminal and insert the result into another.

Despite these advancements, a key limitation remains: As a black-box technique, Fandango cannot currently incorporate *execution feedback* into the evolution of inputs. This is in contrast to state-of-the-art grey-box fuzzers, which build their success on evolving inputs based on execution feedback such as code coverage. Most grey-box fuzzers, however, are unaware of input specifications and thus apply mutations that often violate the input format, reducing efficiency and preventing users from expressing testing goals within the *input domain*. We aim to bridge this gap by enabling Fandango to leverage *execution feedback* during input evolution, as detailed in Section 5.3.

5.2.2 Combining Input Specifications with Execution Feedback

The idea of combining input specifications with execution feedback is not new. Seminal approaches include *grammar-based whitebox fuzzing* [49], which enhances symbolic execution with context-free input grammars; AFLSmart [85], which combines grey-box fuzzing with *Peach Pit* specifications; and Nautilus [8], which integrates grammars and coverage-guided fuzzing. However, all these approaches are *monolithic*—limited to increasing code coverage as their primary execution goal and constrained to syntac-

tic input specifications. That is, semantic input constraints and alternative execution objectives are not supported.

5.2.3 Modular Fuzzers

FuzzFactory [78] is the first modular approach to enable users to concisely implement, reuse and combine specialized fuzzing strategies—such as *performance fuzzing* [66, 83] or *validity fuzzing* [80]—in a unified framework. Built on top of the AFL fuzzer [121], it simplifies the development of domain-specific fuzzers drastically by providing streamlined interfaces into the fuzzer internals, rendering previously prevalent *ad-hoc* implementations obsolete. Despite these advancements, FuzzFactory still requires familiarity with the internals of AFL and C++ programming. Moreover, while FuzzFactory provides great flexibility in terms of the execution feedback that can be leveraged, it does not allow users to incorporate their domain knowledge regarding input specifications into the fuzzing process.

Similarly, LibAFL [45], a Rust-based framework for modular fuzzer development, supports the composition of multiple fuzzers such as Nautilus or Grimoire [25]. However, it, too, lacks support for specifying semantic input constraints, and using or extending the framework requires a deep understanding of its internals.

Our work extends this idea of modular fuzzing by allowing users *to specify input constraints and execution goals in a declarative and reusable way*, as shown in the next section.

5.3 Evolutionary Constraint Solving Meets Grey-Box Testing

Our goal is to extend Fandango with execution feedback, allowing it to generate inputs that not only satisfy user-defined input constraints but also *optimize execution objectives.*

This will allow us to specify execution goals such as:

- minimizing the distance to a specific code location,
- maximizing the number of executed basic blocks,
- maximizing the amount of heap memory allocated.

Initially, Fandango only supported **hard constraints**, which are input constraints that *must* be satisfied. For instance, validity constraints ensure that generated inputs conform to the input language—a non-negotiable requirement.

In addition, we introduce **soft constraints**, which are not strictly required but represent *desirable properties* that can be optimized. Let us detail how we integrate soft

constraints into the Fandango evolutionary algorithm, followed by a description of how we normalize input fitness with respect to these soft constraints.

5.3.1 Integrating Soft Constraints

Fandango defines the fitness of an input as a function $f\colon T \to [0,1]$, which maps a derivation tree $t \in T$ to the average satisfaction score across all input constraints:

$$f(t) = f_H(t) = \frac{1}{|C_H|} \sum_{c \in C_H} s_H(c,t)$$

Here, $C = C_H$ denotes the set of constraints, consisting solely of hard constraints, and $s_H(c,t) \in [0,1]$ represents the satisfaction score of tree t concerning constraint c.

We extend this definition to incorporate *soft constraints*. The overall constraint set is now partitioned into two disjoint subsets: hard constraints C_H and soft constraints C_S, such that $C = C_H \cup C_S$ and $C_H \cap C_S = \emptyset$. Regarding the evolutionary algorithm, the general idea is to structure the search such that Fandango prioritizes generating inputs that satisfy all hard constraints. Subsequently, those inputs are evolved to optimize the soft constraints. To support this, we define a soft constraint fitness function $f_S\colon T \to [0,1]$, which is only evaluated when all hard constraints are satisfied:

$$f_S(t) = \begin{cases} \frac{1}{|C_S|} \sum_{c \in C_S} s_S(c,t), & \text{if } f_H(t) = 1.0 \\ 0, & \text{otherwise} \end{cases}$$

The overall fitness function then becomes an average of the hard and soft components:

$$f(t) = \frac{|C_H| \cdot f_H(t) + |C_S| \cdot f_S(t)}{|C|}$$

Note that $f(t)$ only evaluates to 1 if both $f_H(t)$ and $f_S(t)$ are equal to 1. Clearly, $f_H(t) = 1$ implies that all hard constraints are satisfied, but how do we interpret the value of $f_S(t)$? We will discuss this in the next section.

5.3.2 Normalizing Soft Constraints

We define soft constraints as arbitrary Python expressions over non-terminals in the grammar, each accompanied by an optimization goal——either `maximizing` or `minimizing`. For instance, the following soft constraint guides the input generation process towards producing increasingly larger values for the ⟨*age*⟩ non-terminal:

maximizing int(⟨*age*⟩)

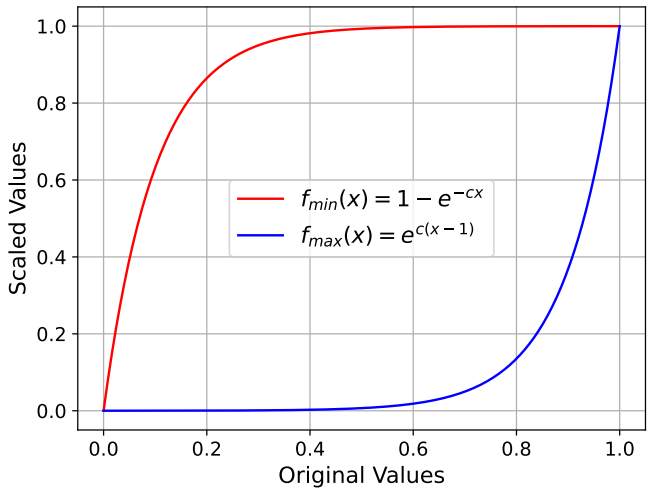


Figure 5.2: Exponential scaling with $c = 10$ is applied to normalized soft constraint fitness values to accentuate improvements near the target boundaries: 0 for minimization and 1 for maximization.

Beyond such properties of the *input domain*, soft constraints can also target execution goals, which are measured by running the program under test on the generated input. However, such execution goals are not naturally *normalized*. For instance, we generally do not know the maximum possible $\langle age \rangle$ value in the context of a larger input specification, the theoretical maximum number of basic blocks that can be covered, or the minimum attainable control-flow distance. As a result, raw fitness values cannot be directly normalized.

To overcome this, we normalize the fitness of inputs regarding soft constraints using a statistical technique based on the *t-digest* data structure [40]. A t-digest maintains a compact summary of a stream of numeric values, allowing efficient estimation of quantiles without storing the whole history. Importantly, it provides a cumulative distribution function (CDF) that maps a value x to the proportion of prior observations less than or equal to x, i.e., a value in the range $[0, 1]$.

This CDF enables normalization: a high value of $\text{CDF}(x)$ indicates that x is large relative to past observations (favorable for maximization). In contrast, a low $\text{CDF}(x)$ value suggests that x is small (favorable for minimization). We further apply exponential scaling to the CDF to accentuate improvements near the target boundaries—i.e., upper bounds when maximizing and lower bounds when minimizing, as shown in Figure 5.2.

The implementation of our soft constraint score computation is shown in Algorithm 6. We first compute the absolute fitness of the input t concerning the soft constraint c by evaluating the Python expression associated with c in the context of the input (Line 2). We then invoke CDF on the absolute fitness value and store the result in `cdf` before the absolute fitness value is added to the t-digest (Lines 3-4). Finally, based on the

optimization goal, we apply exponential scaling to `cdf` and return this normalized value (Lines 5-10).

Algorithm 6 Soft Constraint Score Computation

```
1:  function s_S(c, t, contrast ← 10.0)
2:      absFitness ← c.FITNESS(t)
3:      cdf ← c.tdigest.CDF(absFitness)
4:      c.tdigest.UPDATE(absFitness)
5:      if c.optimizationGoal = "maximize" then
6:          return exp(contrast × (cdf − 1))
7:      else
8:          return 1 − exp(−contrast × cdf)
9:      end if
10: end function
```

5.4 Implementation

5.4.1 Instrumentation and Program Analysis

We implement a lightweight LLVM-based instrumentation framework that enables the collection of execution feedback from the program under test.

Designed for ease of use, our framework makes it straightforward to compile existing C/C++ projects with instrumentation. Our instrumentation pass is bundled in a tool called `fcc`, which acts as a drop-in replacement for `clang`. This tool allows users to instrument arbitrary C/C++ programs with minimal effort. For example, a typical compilation with fcc looks like:

```
$ fcc -o f *.c
```

The resulting instrumented binary can then be passed to Fandango, which will execute it and collect runtime feedback.

In addition, we provide a companion tool called `fan`, which performs simple static analysis on `fcc`-compiled programs. It generates an interprocedural control-flow graph enriched with source code metadata such as module names, function names, and line numbers. Fandango uses this information to support execution metrics like `DistanceToFunction()` and `DistanceToLine()`.

`fan` is invoked automatically by Fandango, but users can also run it manually to inspect the static analysis output:

```
$ fan f
```

From a technical perspective, `fcc` builds on top of the `wllvm`[1] tool. `wllvm` compiles once to produce standard object files and again to generate corresponding LLVM bitcode object files. During the linking stage, `wllvm` embeds references to these bitcode objects within the final binary.

To recover the LLVM bitcode for the entire program, we use the `extract-bc` tool provided by `wllvm`, which we have wrapped inside our `fan` tool. `extract-bc` extracts the embedded bitcode object files, which are then linked using `llvm-link` into a single whole-program LLVM bitcode file. This unified bitcode is used as the input for our analysis pass, from which we derive the inter-procedural control-flow graph of the program under test.

Let us describe the instrumentation pass (used by `fcc`) and the analysis pass (used by `fan`) in more detail. Figure 5.3 illustrates the overall workflow of `fcc` and `fan`.

① **Instrumentation Pass:** The instrumentation pass operates at the module level. It assigns a unique ID to each basic block within a module and annotates it with the module name and this unique ID. Each module maintains its map, where each entry corresponds to a basic block ID and stores a 64-bit counter. When a basic block $BB_{\mathtt{i}}$ in a module is executed, the counter at position `i` in the module-level BBID map is incremented, ensuring that basic blocks remain uniquely identifiable even after linking, as unique IDs are scoped to the module.

Finally, we register an `atexit()` hook that serializes the execution trace—including covered basic blocks and hit counts—to disk upon program termination. While more efficient mechanisms like shared memory could be used, we opt for this more straightforward approach to keep the prototype lightweight and easy to debug.

② **Analysis Pass:** The analysis pass operates on the whole-program bitcode of the program under test. It traverses all functions and their basic blocks to construct an interprocedural control-flow graph, linking individual intraprocedural control-flow graphs via direct function calls. In addition, it enriches each basic block with source code-level metadata, including the module name, function name, and line number. This information enables Fandango to express execution metrics in a clear and user-friendly manner.

5.4.2 Fandango Integration

We integrate our approach into Fandango through two key modifications. First, we extend Fandango to support *soft constraints*, as described in Section 5.3. Second, we introduce an interface to connect Fandango with a user-supplied program under test (PUT), instrumented using *fcc*.

[1] `https://github.com/travitch/whole-program-llvm`

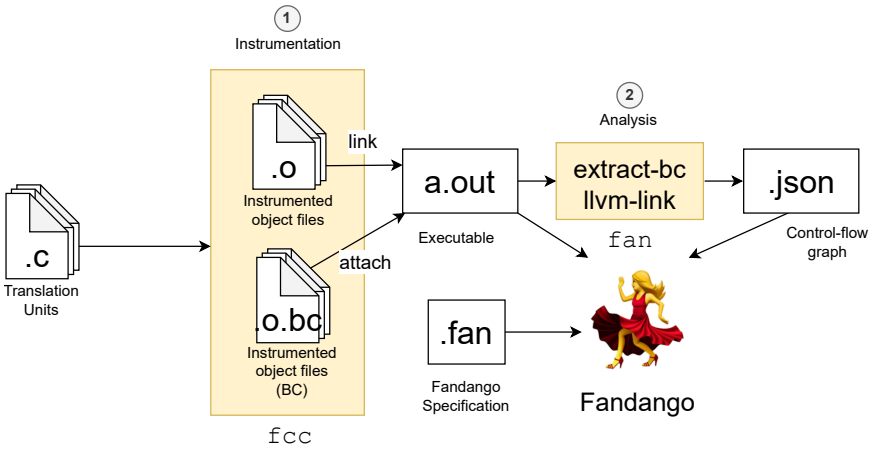


Figure 5.3: Instrumentation workflow with `fcc`. Translation units are compiled twice (with instrumentation) using `wllvm`, producing standard and LLVM bitcode object files. These are bundled into the executable, which Fandango runs to generate execution traces. Our `fan` tool extracts static analysis information from the bitcode object files, enabling the computation of execution metrics (Table 5.4).

Table 5.4: Execution metrics provided by FANDANGOGREY

Fuzzing Goal	FANDANGOGREY Metric	Description
Directed Grey-box Fuzzing [26]	`DistanceToFunction(m, f)`	Call distance to function `f` in module `m`.
	`DistanceToLine(m, l)`	Basic block-level distance to line `l` in module `m`.
	`DistanceToBB(m, bb)`	Basic block-level distance to basic block `bb` in module `m`.
Coverage-Based Fuzzing [121, 99, 44]	`FunctionCoverage()`	Number of covered functions.
	`CodeCoverage()`	Number of covered basic blocks.
	`CoverageInFunction(f)`	Number of covered basic blocks in function `f`.
Performance Fuzzing [83, 66, 78, 1]	`ExecutionPathLength()`	Execution path length measured in basic block hits.
	`PeakBasicBlockHits()`	Number of hits of the most frequently executed basic block.
Excessive Memory Fuzzing [78]	`HeapAllocatedBytes()`	Total size of heap memory allocated.

When the input specification includes the *DynamicAnalysis* keyword, our Fandango extension begins by loading static analysis information extracted from the PUT using the *fan* tool. This information is then processed to enable the use of our execution metrics (Table 5.4). Specifically, control-flow graphs are used to precompute control-flow distances for directed fuzzing. At the same time, debug information is used to map source-level constructs to basic blocks, facilitating more intuitive interpretations of execution metrics.

After this preprocessing step, Fandango executes the PUT on each input that satisfies all hard constraints. For each execution, it reads the execution trace produced by the instrumentation and evaluates the specified execution metric. To improve performance, trace data is cached using a *least-recently-used* (LRU) strategy.

5.5 Domain-Specific Fuzzing Applications

In Section 5.5.1 through Section 5.5.5, we demonstrate the practicality, usability, and versatility of FANDANGOGREY by applying it on five different domain-specific fuzzing scenarios, each targeting a different execution objective. Some of these domains have previously been explored in isolation by other researchers. However, our goal is not to outperform existing fuzzers in speed—an unfair comparison given that our implementation is in Python. In contrast, most state-of-the-art fuzzers are highly optimized and implemented in C or C++. Instead, our key contribution lies in being *the first to integrate execution goals with input constraints in a unified framework.* Our evaluation thus focuses on demonstrating that the combination of an input specification and execution feedback outperforms either component used alone in Fandango. Our catalog of execution metrics currently comprises those presented in Table 5.4, exploring the following five domains:

(A) **Performance Fuzzing**. In this domain, we aim to find inputs that are both small and slow to execute. We demonstrate this by implementing the `SlowFuzz` technique [83].

(B) **Excessive Memory Fuzzing.** In line with the "mem" application of FuzzFactory [78], we aim to find inputs that allocate excessive amounts of heap memory.

(C) **Maximizing Code Coverage.** We aim to find inputs that maximize the number of basic blocks covered by a single input rather than maximizing coverage across the entire input population, as is typical in coverage-guided fuzzers like AFL.

(D) **Directed Grey-box Fuzzing.** In this domain, we implement a technique similar to "Directed Greybox Fuzzing" [26], which aims to find inputs that reach a specific target location in the program.

(E) **Evolving Populations Towards Coverage.** We also implement fuzzing in the style of AFL [121], evolving *populations of inputs* towards maximizing cumulative code coverage—while taking input constraints into account.

We evaluate these domain-specific fuzzing scenarios by running experiments on real-world programs and corresponding input specifications, which we detail in the next section. Each experiment is run for one hour. We repeat each experiment ten times to account for the randomness inherent to fuzzing. After collecting the data from all ten runs, we compute the average and 95% confidence intervals. In our plots, the averages are represented by solid lines, with the surrounding shaded areas indicating the 95% confidence intervals. Each plot compares three different fuzzing setups:

(1) **Execution feedback only:** This setup relies solely on execution feedback to guide fuzzing, using a trivial input specification that can generate arbitrary strings:

$$\langle start \rangle ::= \langle byte \rangle^+$$

(2) **Input specification only:** This setup uses a detailed input specification—consisting of a context-free grammar and input constraints—but does not entail execution feedback.

(3) **Combined approach:** This setup integrates both strategies by combining the execution feedback from setup (1) with the input specification used in setup (2).

The research questions we address in this evaluation are:

1. *Can we combine input constraints and execution feedback concisely?*
2. *Does the combination yield better results than each component alone?*

Note that we do not directly compare against existing fuzzers. For one, fuzzers specialized for one task typically are *optimized* for this very task, at the expense of generality (which we strive for). Also, their *requirements* vastly differ; AFL performance, for instance, largely depends on the quality of the initial seed and the AFL dictionary, whereas the FANDANGOGREY performance depends on the quality of the input specification and the given constraints, making a fair comparison impossible. The contributions of this work focus on control and versatility, and our evaluation is set accordingly.

Evaluation Subjects

We evaluate our approach on four real-world programs written in C and C++. These programs were selected due to their complex and structured input languages, the availability of language specifications, and their prior use in research. Three of the programs—*graphviz* [9], *lunasvg* [114], and *libxml2* [116]—were also used by a recent paper on finding slow but short inputs using grammar-based search [1]. We use these programs in Section 5.5.1—Section 5.5.3 and Section 5.5.5. We further evaluate *cjson* [47], a widely used JSON parser implemented in C, in a directed grey-box fuzzing scenario in Section 5.5.4. In the following, we briefly describe these programs and their input languages.

graphviz [9] is a popular graph visualization software that primarily uses the DOT language for describing graphs. The DOT language has a text-based syntax and allows users to define nodes, edges, and attributes in a structured manner. We use the same syntax specification as in [1], which was extracted from the Bison EBNF grammar that ships with the graphviz source code. While graphviz contains multiple tools for different purposes, our experiments focus on the `dot` tool, which generates visual representations of graphs.

lunasvg [114] is a C++ library for rendering SVG (Scalable Vector Graphics) files. SVG is an XML-based vector image format that describes images using geometric shapes, paths, and text. We reuse the input grammar provided as part of the reproducibility package of [1], which was derived from the W3C documentation[2] and focuses on the path element. lunasvg ships with a command-line tool called `svg2png`, which converts SVG files to PNG images. We use this tool in our experiments.

libxml2 [116] is a C library for parsing XML documents, initially developed for the GNOME project. libxml2 is often used in fuzzing benchmarks [1, 78]. We use the XML specification provided in the reproducibility package of Fandango, which includes both a syntax specification and semantic input constraints–specifically, matching opening and closing tags and ensuring the uniqueness of attributes.

Our evaluation focuses on the `xmllint` command line tool that ships with libxml2.

cjson [47] is a popular lightweight JSON parser written in C. We use a JSON input grammar previously mined specifically for cjson [23]. Since cjson does not include a standalone executable, we build a simple command-line tool that reads an input file and passes it to the parser entry point of cjson.

All input languages considered in this work are text-based. However, no inherent limitation prevents our approach from being applied to binary input languages. Fandango fully supports various byte-level encodings, bitstrings, endian conversions, and checksum computations through grammar-attached generators[3]. That said, constructing input specifications for complex binary formats—such as PNG or JPEG—along with their associated semantic constraints (e.g., checksums) is time-consuming. We leave this for future work, as it lies beyond the scope of this chapter.

5.5.1 Performance Fuzzing

Fuzzing for performance issues has been explored in multiple prior works [83, 66, 78]. We adopt the *SlowFuzz* approach [83], one of the earliest methods in this space. The basic idea is to guide the fuzzer towards finding inputs that take a long time to execute while keeping the input size small—since longer inputs tend to increase runtime trivially.

To quantify the execution cost of an input, we use the `ExecutionPathLength()` metric, described in Table 5.4. Our instrumentation tracks how often each basic block is executed and aggregates this information into a total path length via `ExecutionPathLength()`.

[2] `https://www.w3.org/TR/SVG/`
[3] `https://fandango-fuzzer.github.io/Binary.html`

Consistent with observations in [1], we find that extremely costly inputs can be generated for lunasvg. To maintain experimental practicality, we impose a maximum execution time of ten seconds per input. When an input exceeds this limit, the program is terminated.

Constraint Example 1 (Performance Fuzzing)**.** *In line with prior work [66, 1], we enforce an input length limit of 60 bytes using an input constraint. We model this* performance fuzzing *setup in Fandango using two constraints. First, a* ***hard constraint*** *limiting the input length to 60 bytes:*

$$\text{len}(\text{str}(\langle \textit{start} \rangle)) \leqslant 60 \tag{5.6}$$

Second, a ***soft constraint*** *guiding the fuzzer to maximize the execution path length:*

$$\begin{aligned} \textbf{maximizing } & \text{DynamicAnalysis}\big(\langle \textit{start} \rangle\big) \\ & \text{.ExecutionPathLength()} \end{aligned} \tag{5.7}$$

Results. We apply this *performance fuzzing* setup to three subjects: lunasvg, graphviz, and libxml2. Results for this experiment are shown in the top row of Figure 5.5. Next, we will detail our findings for each subject.

graphviz

When using only execution feedback and a trivial grammar, the fuzzer struggles to make progress, producing only inputs that have very short execution paths, with practically no improvement over time, likely because Fandango cannot construct valid or even semi-valid graphviz inputs from scratch in this setup.

Using only an input specification (i.e., no execution feedback), Fandango generates inputs with considerably longer execution paths. However, improvement stagnates over time without feedback since the fuzzer is unaware of the execution path length in this setup.

The best results are achieved by combining the input specification with execution feedback. In this setup, inputs run substantially longer, with the best-performing inputs reaching execution path lengths exceeding 20 million basic block hits, which is an improvement of approximately 10x over the specification-only baseline.

lunasvg

We observe the same trends as for graphviz. Combining input specification with execution feedback yields the longest execution paths, outperforming both the execution feedback-only and the specification-only configurations.

While performance was the primary focus, our fuzzer also uncovered stability issues in lunasvg. One generated input caused a segmentation fault due to invalid memory

write access, which we reported to the developers[4]. Another input caused the program to hang for over 15 hours; this was also reported as a likely infinite loop[5]. Both bugs were acknowledged and fixed by the developers.

As discussed, we limited the execution time to ten seconds in the final experiments. Consequently, inputs generated using execution feedback and the input specification reach a glass ceiling at an execution path length of about twelve billion. Without this limit, even more expensive inputs could be generated.

libxml2

Interestingly, libxml2 exhibits different behavior. Although combining an input specification with execution feedback outperforms the specification alone, the execution feedback-only configuration produces inputs with significantly longer execution times. We attribute this to the nature of XML parsers: well-formed inputs (as generated using a specification) are relatively cheap to parse, whereas invalid or semi-valid inputs (as produced by a generic grammar with execution feedback) often trigger expensive error-handling paths, resulting in higher execution costs.

5.5.2 Excessive Memory Fuzzing

This domain-specific fuzzing task focuses on discovering inputs that cause programs to allocate excessive amounts of memory. Adopting this domain from FuzzFactory [78], we implement `HeapAllocatedBytes()` (Table 5.4) by instrumenting all `malloc()` and `calloc()` calls in the program under test, and summing the total number of bytes allocated during execution. As in the previous experiment, we enforce a 60-byte input size limit to find short inputs that trigger large memory allocations, possibly indicating a bug.

Constraint Example 2 (Excessive Memory)**.** *The following Fandango constraints model the above setup:*

$$\text{len}(\text{str}(\langle \textit{start} \rangle)) \leqslant 60 \tag{5.8}$$

$$\textbf{maximizing}\ \text{DynamicAnalysis}\big(\langle \textit{start} \rangle\big).\text{HeapAllocatedBytes}() \tag{5.9}$$

Results. The resulting plots for this experiment are shown in the middle row of Figure 5.5. Our findings for each of the three subjects are outlined below.

For graphviz and lunasvg, we observe similar trends as in the *performance fuzzing* domain: combining input specifications with execution feedback consistently outperforms using either in isolation. In the case of lunasvg, substantial memory allocations

[4] `https://github.com/sammycage/lunasvg/issues/221`
[5] `https://github.com/sammycage/lunasvg/issues/222`

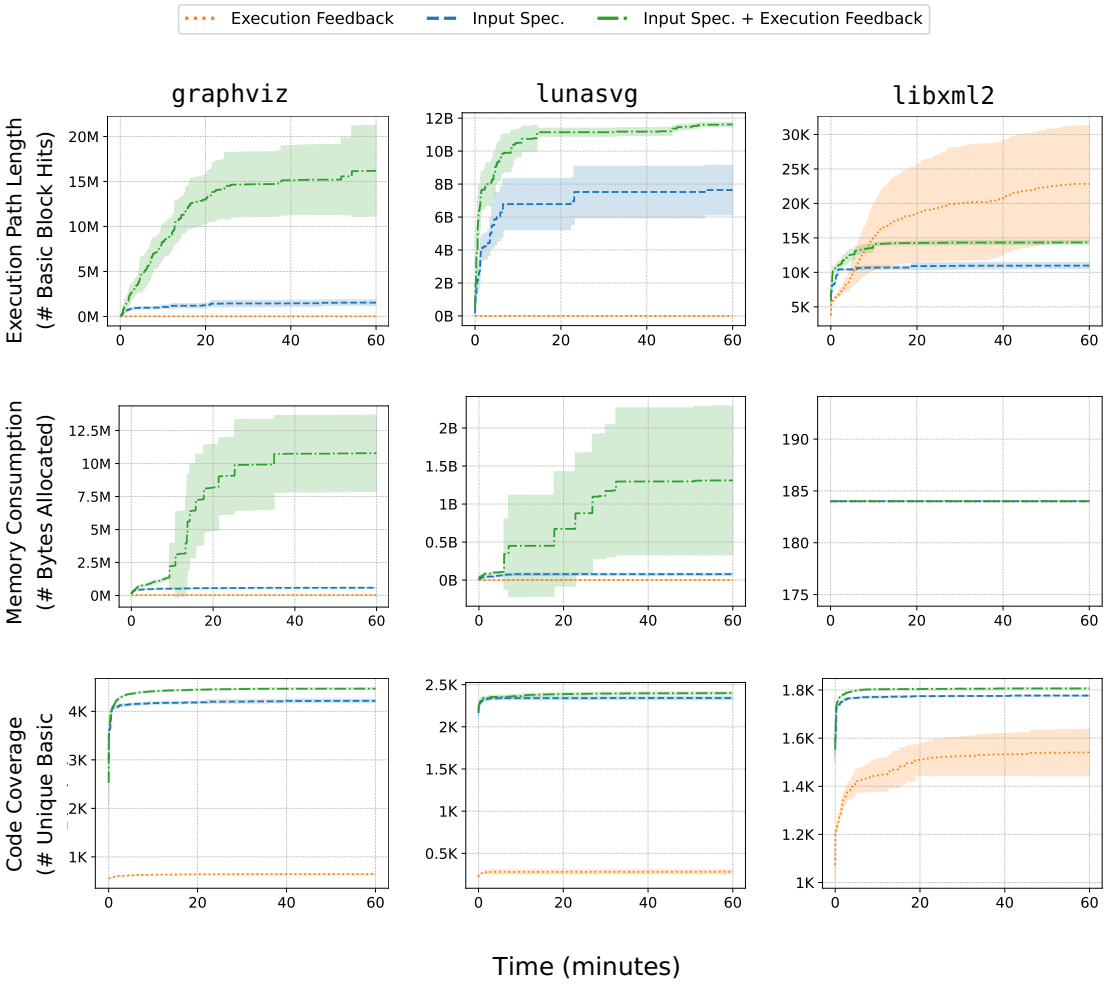


Figure 5.5: Results for Section 5.5.1–Section 5.5.3, comparing the performance of (1) execution feedback only, (2) input specification only, and (3) their combination across three domain-specific fuzzing tasks: maximizing execution path length, heap memory consumption, and code coverage. Each experiment ran for one hour and was repeated ten times; bold lines indicate the average, with shaded areas representing the 95% confidence interval.

are observed. The input below is one of the costliest, allocating approximately 4 GB of memory in a single call:

```
<svg><path d=" M -279 , 91M 30704 32715Z"/></svg>
```

We investigated this issue and found that the allocation originates from the following code, where `width` and `height` appear to be derived directly from the input:

```
const size_t size = width * height * 4;
plutovg_surface_t* surface =
 malloc(size + sizeof(plutovg_surface_t));
```

In contrast, libxml2 shows constant heap allocations across all configurations, suggesting that it does not allocate heap memory dependent on the input.

5.5.3 Maximizing Code Coverage

We introduce this novel fuzzing domain, which aims to identify *individual inputs* that exercise as much code in the program under test as possible. One potential application is finding inputs that simultaneously trigger multiple distinct features, where their interaction—especially in stateful systems—may expose complex, otherwise hard-to-find bugs. We leverage our `CodeCoverage()` metric to implement this domain, which aggregates all executed basic blocks.

Constraint Example 3 (Maximizing Code Coverage)**.** *Our setup requires only one Fandango soft constraint:*

$$\textbf{maximizing}\ \text{DynamicAnalysis}(\langle \textit{start} \rangle)\ \text{.CodeCoverage()} \tag{5.10}$$

Results. The results for this experiment are shown in the bottom row of Figure 5.5. Below, we present our findings for each subject in detail.

graphviz

Using execution feedback alone results in relatively low basic block coverage. In contrast, both input specification-based setups achieve significantly higher coverage. Notably, combining the input specification with execution feedback yields the highest coverage overall.

lunasvg

Again, the two input specification-based setups outperform execution feedback alone. Combining the input specification with execution feedback results in a slight increase

in basic block coverage. We attribute this to the input specification being sufficiently succinct and expressive, allowing Fandango alone to achieve broad coverage.

libxml2

The combination of input specification with execution feedback again performs best within the one-hour experiment window. Interestingly, execution feedback alone also shows steady progress. Manual inspection of the generated inputs revealed that this setup produces a mix of simple valid XML inputs and many invalid inputs, contributing to the measured basic block coverage.

5.5.4 Directed Grey-Box Fuzzing

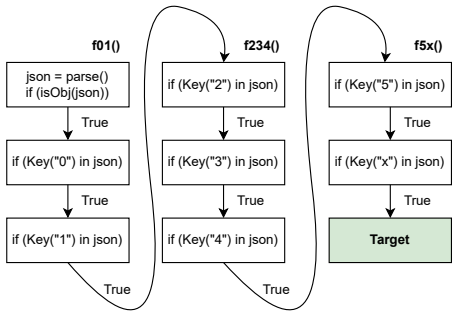
Directed grey-box fuzzing focuses on guiding the fuzzer toward specific target locations in the code to find bugs in the vicinity of that code. One application is, for instance, testing recently modified code. Several approaches have been proposed in this area [26, 31, 32, 39], with the seminal work being *Directed Greybox Fuzzing* by Böhme et al. [26]. Our implementation follows a similar approach.

Initially, we perform the following computations once: We extract the call graph of the PUT (consisting of direct calls only) and intraprocedural control-flow graphs for each function. Using breadth-first search, we compute function-level call distances for each pair of functions and basic block-level control-flow distances for each of the two basic blocks in a function. Consistent with [26], these two metrics are then combined into a single distance metric by penalizing interprocedural transitions with a constant factor of 10, allowing to define a distance between any two basic blocks in the program.

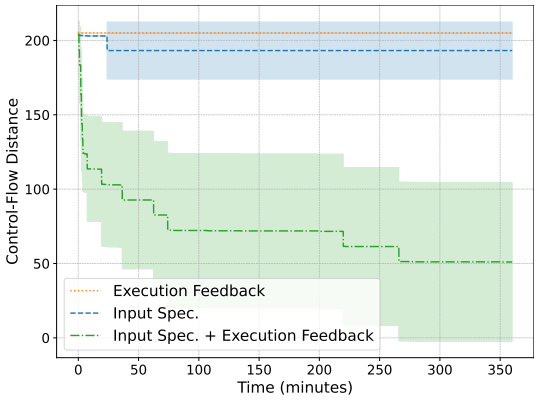
During fuzzing, we then use this precomputed information *to calculate the control-flow distance from an input to a target location.* More precisely, for each basic block executed by the input as indicated by the trace, we look up the distance to the target location, returning the minimum value as the result.

We demonstrate directed grey-box fuzzing using a small JSON-based experiment. In this setup, a harness wraps the `cjson` parser to parse an input file and then performs a sequence of dependent checks on the parsed JSON structure across three functions: `f01()`, `f234()` and `f5x()`. The function `f01()` checks that the parsed input is a JSON object containing the keys `"0"` and `"1"`; `f234()` then checks for the presence of keys `"2"`, `"3"`, and `"4"`; finally, `f5x()` looks for keys `"5"` and `"x"`. All these checks are sequentially dependent; for instance, the presence of key `"x"` is only checked for if `"5"` was found. The goal is to reach a specific *target basic block*, which we set to the one executed when `"x"` is present. This setup is illustrated in Figure 5.6a.

The key challenge is that the fuzzer must first generate syntactically valid JSON inputs to pass parsing and then evolve those inputs through mutations to reach the target



(a) Schematic control-flow diagram of the `cjson`-based directed grey-box fuzzing experiment. All branches must evaluate to `True` to reach the target basic block (ID "2" in our Fandango constraint).



(b) Result of the directed grey-box fuzzing experiment.

Figure 5.6: Directed grey-box fuzzing: (a) control-flow schematic and (b) experimental results of the `cjson` experiment.

basic block. To implement this experiment, we configure the fuzzer to minimize the control-flow distance to the *target basic block*, which in this example has the ID `"2"`.

Constraint Example 4 (Directed Grey-box Fuzzing)**.** *Again, a single Fandango soft constraint suffices:*

$$\textbf{minimizing}\ \text{DynamicAnalysis}(\langle \textit{start} \rangle) \\ .\text{DistanceToBB}(\text{“harness.c”}, \text{“2”}) \tag{5.11}$$

We ran this experiment for six hours, comparing fuzzer configurations based on execution feedback, input specifications, and their combination. The results are shown in Figure 5.6b.

Results. The execution feedback-only configuration fails to make any progress towards the target. In contrast, the input specification-only configuration makes limited progress and even reaches `f234()` in one of ten runs, as it can generate valid JSON inputs and produces the first two keys by chance. However, without execution feedback, it cannot progress further.

The combined configuration—input specification plus execution feedback—performs best. It makes steady progress and reaches the target basic block in seven out of ten runs within the six-hour experiment. This suggests that FANDANGOGREY effectively combines specification-based input generation with directed fuzzing guided by execution feedback.

One of the target-reaching inputs is shown below. Note that all required keys are present, and the key "X" is uppercase, which works as cjson performs case-insensitive key lookups.

```
{"2":true,"0":"/","5":"","4":true,"3":false,"X":"","1":"/"}
```

5.5.5 Evolving Populations

The domain-specific fuzzing approaches discussed so far evolve *single inputs*, optimizing them towards desired runtime behaviors. However, during testing, one may not only be interested in a single (failing) test input, but rather in a *population of inputs* that collectively maximize a certain metric such as coverage. This is particular prevalent in coverage-guided fuzzers like AFL, which also evolve and produce a population of inputs.

By default, the Fandango framework is set to evolve single inputs. However, we can easily set it up to evolve *populations* instead. To this end, we transform the input grammar so that it produces a *population of inputs* instead of just a single input. For example, if the original grammar is defined as (omitting the definition of ⟨*input*⟩ for brevity):

$$\langle start \rangle ::= \langle input \rangle$$

we replace the production rule of the start symbol with:

$$\langle start \rangle ::= \langle input \rangle \{\mathsf{K}\}$$

where `K` is the desired population size, and `{K}` means repetition, as with regular expressions. As a result, Fandango applies mutation and crossover to evolve entire populations of inputs.

During evolution, Fandango does not pass the full string derived from ⟨*start*⟩ as the input to the program under test (as it would represent the entire population). Instead, it uses each individual ⟨*input*⟩ subtree as a separate test input, as enforced by the constraint shown below.

Constraint Example 5 (Evolving Populations)**.** *The following Fandango constraint models this approach:*

$$\textbf{maximizing} \left| \bigcup_{\substack{\langle input \rangle \\ \in \langle start \rangle}} \text{DynamicAnalysis}(\langle input \rangle).\text{CoveredBasicBlocks}() \right| \tag{5.12}$$

For readability, we present this constraint in mathematical notation. In the actual constraint, we make use of Python built-ins such as `set.union()` and `len()` instead.

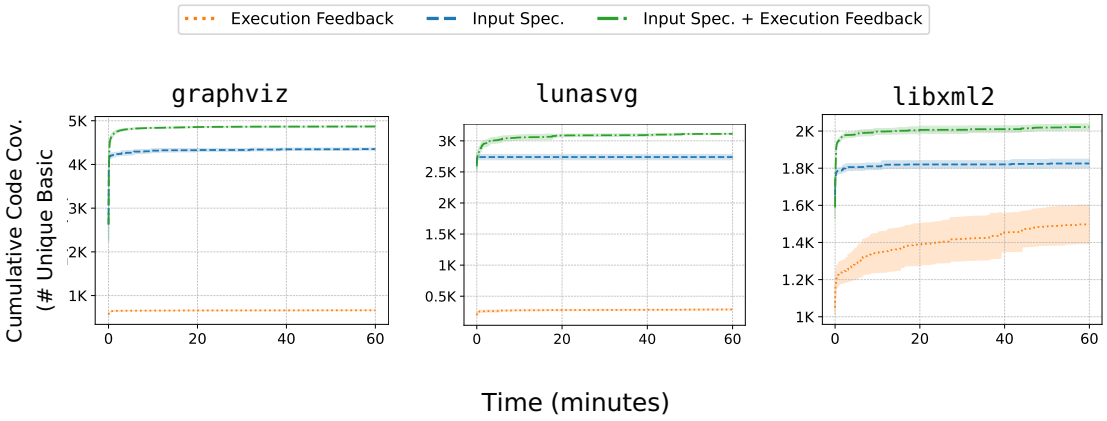


Figure 5.7: These results of Section 5.5.5 show that coverage-guided fuzzing can be effectively implemented within our framework, and that combining input specifications with execution feedback yields better performance than using either in isolation.

The ⟨*input*⟩ non-terminals are obtained as the children of ⟨*start*⟩ using Fandango's `children()`[6] function. Our primitive `CoveredBasicBlocks()` extracts the set of all basic block IDs covered by a given input from its execution trace, and is also used internally by our `CodeCoverage()` metric.

As in previous experiments, we evaluate this approach by running it on three subjects for one hour each, repeating the experiment ten times. We compare three configurations: using execution feedback alone, using input specifications alone, and using both in combination. For this experiment, we set the population size to $K = 10$.

Results. The experimental results are shown in Figure 5.7. Consistent with previous experiments, combining input specifications with execution feedback (i.e., coverage guidance) yields the best performance across all subjects, while input specifications alone still outperform execution feedback alone. In addition, for every subject, the cumulative code coverage achieved by the *populations* in this experiment consistently exceeds the single-input code coverage reported in Section 5.5.3, demonstrating the effectiveness of population-based optimization.

Note that Equation (5.12) illustrates just *one* goal and just *one* way to combine information from individuals. By modeling populations as individual inputs, FANDANGOGREY can evolve a population towards arbitrary goals regarding input shapes or executions, statistical distributions, or combinations thereof. On top, one could also model the Fandango individuals as *sets of populations*, where each (sub-)population is set to achieve a different goal. All this illustrates the expressiveness and the versatility of FANDANGOGREY.

[6] `https://fandango-fuzzer.github.io/DerivationTree.html#accessing-children`

5.5.6 Summary of Results

In summary, our evaluation demonstrates that

1. **Specifying fuzzing goals as constraints is very concise.** As shown in constraint examples 1 to 4, FANDANGOGREY requires but *one to two lines of constraints* to express common fuzzing goals. This sharply contrasts existing approaches [83, 66, 78], which typically require users to extend the actual implementation.

2. **Constraints can equally refer to inputs and execution.** Constraint examples 1 and 2 demonstrate how to combine constraints related to the input language (say, Eq. (5.6)) with constraints related to the execution (say, Eq. (5.7)). Many more such combinations are possible.

3. **Specification-guided (black-box) fuzzing benefits from execution feedback.** In all our experiments, we found that coverage feedback improves the performance of fuzzing over specification guidance alone.

4. **Coverage-guided (white-box) fuzzing benefits from input specifications.** In all our experiments, we found that input specifications improve the performance of fuzzing over execution feedback alone. The one exception is libxml2, where *invalid* inputs achieve the highest execution path length; we plan to extend our approach to support such out-of-specification inputs.

5. **Evolving populations.** Our approach allows evolving and producing individuals as well as *populations of inputs*, specifying goals for each individual or the population as a whole.

6. **A modular approach.** FANDANGOGREY is the one approach that combines all the above benefits, allowing to specify and combine input languages, black-box testing goals, white-box testing goals, and evolution of individuals or populations in a modular and freely adjustable fashion.

5.6 Threats to Validity

As with all empirical studies, our work is subject to various threats to validity. We outline our mitigation strategies below.

External Validity refers to the extent to which the results can be generalized to other subjects or contexts not included in the study. To mitigate this threat, we selected a diverse set of real-world programs and input languages, all used in prior fuzzing research. Moreover, while our implementation currently supports a set of nine execution metrics, it can easily be extended, as the underlying evolutionary algorithm is agnostic to the used metric.

Internal Validity refers to whether causal conclusions can be drawn from the results. We mitigated this threat by building our prototype on top of Fandango, which has been used in prior research, and applying it to established fuzzing domains. We also manually verified the correctness of our implementation through unit tests prior to running experiments.

Construct Validity refers to the extent to which the chosen metrics accurately measure the intended objectives. We used standard metrics from prior work to mitigate this threat, such as execution path length.

5.7 Discussion and Future Work

Mining input specifications. As we demonstrated, input specifications can be effectively combined with execution feedback, a promising direction for enhancing fuzzing. A key challenge, however, is that input specifications are not always readily available, and their manual construction is time-consuming and error-prone. A potential solution to this problem is to automatically mine input specifications from the program under test. Prior work has primarily focused on extracting syntactic input structure both using black-box [64, 5, 67] and white-box approaches [57, 52, 24, 23, 43], but largely ignores semantic constraints. As part of our future work, we plan to investigate the automatic inference of *semantic* input constraints.

Efficient implementation. Our current FANDANGOGREY prototype is based on Fandango, which is implemented in Python. While this facilitates rapid development and flexibility, it introduces a performance overhead compared to highly optimized fuzzers such as AFL++ [44]. Reimplementing Fandango in a compiled language, with efficient interfacing to the target program, could significantly improve performance and scalability.

A library of fuzzing approaches. In the future, we plan to extend our set of domain-specific fuzzing approaches, giving users an even richer toolbox of fuzzing techniques that they can freely combine with *input constraints*. One of the approaches we plan to implement is IJON [7]—by supporting soft constraints such as `maximizing DataCoverage(module, variable)`. In addition, we aim to extend our coverage-guided fuzzing capabilities by implementing common optimizations, such as *input minimization* and *power schedules*, which are out of scope of this work, which focuses on combining input constraints with execution feedback and demonstrating the fundamental feasibility thereof.

5.8 Conclusion

In this work, we presented FANDANGOGREY, a novel approach for specifying *execution goals in conjunction with input constraints*, enabling rich grey-box fuzzing applications where inputs are first shaped according to user-defined constraints and then evolved to optimize execution goals. We implemented a prototype of our approach by extending the Fandango tool with instrumentation-based execution feedback built on LLVM. We provide a comprehensive catalog of domain-specific execution goals that enable users to express their testing objectives in a declarative and user-friendly manner. To demonstrate the practicality and versatility of our approach, we applied it to five domain-specific fuzzing tasks across four real-world C and C++ programs. We believe our approach opens up new possibilities in fuzzing, and we are excited to see how the community will leverage this in future research.

6 Conclusion and Future Work

This chapter concludes this dissertation, but not research on *grammar inference and fuzzing*—good news for the next generation of researchers!

Grammar inference and fuzzing are deeply rooted in formal language theory and program analysis—fields that have been studied for decades. What makes these fields interesting is that many of the core problems are *undecidable*. This ensures that new research questions and discoveries will continue to emerge, advancing both theoretical understanding and practical applications.

In the following, we start by outlining the main focus of this work, followed by a summary of our key contributions presented in Chapter 3 to Chapter 5. We then conclude with an outlook on promising directions for future research.

This dissertation focuses on advancing grammar-based fuzzing research. Grammar-based fuzzing enables the efficient generation of diverse, syntactically valid inputs, making it particularly well-suited for testing programs that process structured data. Despite these advantages, we have identified and addressed significant challenges:

1. **How can we apply grammar-based fuzzing when no input grammars are available?** Existing techniques to mine input grammars require a representative set of *sample inputs* together with a *parse oracle* to infer context-free grammars. However, sample inputs may not be available, or may provide only incomplete coverage of the input language.

   Our contribution: We introduced the first symbolic execution-based technique that learns context-free grammars directly from recursive descent parsers, requiring *no sample inputs*.

   Our evaluation shows that the mined grammars are *highly accurate*. In addition, our evaluation comprises a large set of nontrivial parsers implemented in C and C++, demonstrating the practical applicability of our approach.

2. **How can we systematically measure the quality of inferred grammars to compare grammar miners?** To drive progress in the field of grammar inference, it is important to benchmark competing approaches with robust evaluation metrics to understand how well they perform. A common strategy is to compare a mined grammar against a golden (ground truth) grammar. However, since grammar equivalence is undecidable, grammars are typically compared using approximate measures of accuracy.

   Existing work estimates *grammar precision and recall* by sampling 1,000 inputs from both the mined and golden grammar and checking how many of these inputs can

be parsed by the other grammar. However, we found that the sampling technique used is not systematic, and the choice of 1,000 inputs is arbitrary.

Our contribution: We introduced a systematic evaluation technique that enumerates all k-paths in both the mined and the ground truth grammars, derives inputs from them, and computes precision and recall using a similar *parse-check* approach. Unlike random sampling, this method ensures coverage of all grammar features in diverse contexts, enabling a more thorough assessment of grammar accuracy. In addition, it offers fine-grained insights into *which* language features were successfully learned.

We applied this technique to grammars mined by five state-of-the-art approaches and identified discrepancies between previously reported accuracy and our reproduced results.

3. **How can we combine language-based testing with grey-box fuzzing?** Fuzzing techniques can be divided into two categories. Language-based black-box fuzzers—such as ISLa and Fandango—are able to generate inputs that satisfy syntactic and semantic input constraints defined by an input specification, but lack execution feedback. As a result, they cannot, for example, evolve inputs towards covering more program code. In contrast, grey-box fuzzers—such as AFL—use execution feedback to evolve inputs, but they do not typically incorporate input specifications. This means that their input mutations frequently generate inputs that the program under test rejects due to syntactic or semantic violations.

   Our contribution: We developed a *unified approach* that bridges these two paradigms. Specifically, we extended Fandango to incorporate execution feedback into its evolutionary algorithm: it first generates *valid* inputs, which are then evolved to optimize execution objectives. To this end, we concisely integrated execution feedback primitives into the Fandango language.

   Our evaluation demonstrates that this combination enables a range of domain-specific fuzzing scenarios—including performance fuzzing, directed fuzzing, and coverage-guided fuzzing—and that combining execution feedback with input constraints consistently outperforms using either technique alone.

Despite these advances, several open challenges remain. We highlight the most promising directions—some of which were previously discussed in the respective chapters—hoping to inspire future research.

Mining input grammars in other domains. Our work on sample-free input grammar mining has focused on *recursive descent parsers*. Tailored to their specific implementational characteristics, it enables a comprehensive inference of their accepted grammars. While recursive descent parsers are widely used, many

other types of input processors could likewise be exploited in domain-specific analyses.

For example, future work could explore techniques for learning *binary input languages* (as opposed to text-based languages typically handled by recursive descent parsers) or for inferring *communication protocols* involving multiple parties. Inferring such input languages directly from implementations would similarly yield models that support both program understanding and systematic testing.

Mining semantic input specifications. Mining context-free input grammars is just the start. Beyond learning syntactic constraints of an input language, an important but challenging goal is to infer *semantic input constraints*—properties that cannot be expressed in a context-free grammar alone. First studies in this direction have explored the inference of ISLa constraints [20] and attribute grammars [87].

For a long time, progress in this area was impeded by the lack of sufficiently expressive yet manageable specification languages. General-purpose programming languages tend to be too unwieldy for concisely modeling input constraints (including quantifiers), while formalisms such as attribute grammars or context-sensitive grammars are also cumbersome. Another difficulty is that *declarative constraints* are easier to extract from programs or executions than *imperative computations*, but require a solver to generate satisfying inputs.

A promising step forward is the recently released Fandango tool, which provides a concise and intuitive language to express input constraints over grammar non-terminals and comes equipped with a solver based on an evolutionary algorithm. We believe it could serve as an ideal platform for hosting inferred input constraints.

To extract such input constraints from programs and executions, we outline two complementary approaches. First, symbolic or concolic execution could be used to track input constraints at the byte level, which would then need to be lifted to the grammar level, as sketched in [20]. Second, black-box techniques could be employed in the spirit of [61, 42], which synthesize input constraints from atomic patterns that serve as hypotheses and are iteratively refined or discarded based on experiments with continuously generated passing and failing inputs.

We look forward to seeing how others will leverage the findings of this dissertation and are excited for what comes next!

Bibliography

[1] Ziyad Alsaeed and Michal Young. "Finding short slow inputs faster with grammar-based search". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1068–1079.

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.

[3] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6. URL: https://doi.org/10.1016/0890-5401(87)90052-6.

[4] Dana Angluin and Michael Kharitonov. "When Won't Membership Queries Help? (Extended Abstract)". In: *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*. Ed. by Cris Koutsougeras and Jeffrey Scott Vitter. ACM, 1991, pp. 444–454. DOI: 10.1145/103418.103420. URL: https://doi.org/10.1145/103418.103420.

[5] Mohammad Rifat Arefin et al. "Fast Deterministic Black-box Context-free Grammar Inference". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, 117:1–117:12. DOI: 10.1145/3597503.3639214. URL: https://doi.org/10.1145/3597503.3639214.

[6] *Arithmetic Expression Parser*. https://github.com/fbuihuu/parser/blob/master/rdp.c. Accessed: 2024-04-11.

[7] Cornelius Aschermann et al. "Ijon: Exploring deep state spaces via fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1597–1612.

[8] Cornelius Aschermann et al. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/.

[9] AT&T Labs Research. *Graphviz: Graph Visualization Software, Version 2.47.0*. https://gitlab.com/graphviz/graphviz/-/archive/2.47.0/graphviz-2.47.0.tar.gz. Accessed: 2025-05-01. 2021.

[10] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018), 50:1–50:39. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[11] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in

Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_24.

[12] Clark Barrett, Daniel Kroening, and Tom Melham. "Problem Solving for the 21st Century". In: (2014).

[13] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[14] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories". In: *Handbook of model checking* (2018), pp. 305–343.

[15] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. "SMT-COMP: Satisfiability Modulo Theories Competition". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 20–23. DOI: 10.1007/11513988_4. URL: https://doi.org/10.1007/11513988%5C_4.

[16] Clark W. Barrett and Cesare Tinelli. "CVC3". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 298–302. DOI: 10.1007/978-3-540-73368-3_34. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_34.

[17] Clark W. Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_14.

[18] Osbert Bastani et al. "Synthesizing program input grammars". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 95–110. DOI: 10.1145/3062341.3062349. URL: https://doi.org/10.1145/3062341.3062349.

[19] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. ""Synthesizing input grammars": A replication study". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 260–268.

[20] **Leon Bettscheider**. "Learning Test Input Constraints from Branch Conditions". In: *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 248–250. DOI: 10.1109/ICSE-COMPANION58688.2023.00067.

[21] **Leon Bettscheider**, Marius Smytzek, and Andreas Zeller. "Combining Input Constraints with Execution Goals". In: *IEEE Conference on Software Testing, Verification and Validation, ICST 2026, Daejeon, South Korea, May 18–22, 2026*. 2026.

[22] **Leon Bettscheider** and Andreas Zeller. "How Good are Input Grammar Miners? An Empirical Study". In: *Proceedings of the IEEE/ACM 48th International*

Conference on Software Engineering, 2026, Rio de Janeiro, Brazil, April 12-18, 2026. 2026.

[23] **Leon Bettscheider** and Andreas Zeller. "Inferring Input Grammars from Code with Symbolic Parsing". In: *ACM Transactions on Software Engineering and Methodology* (2025).

[24] **Leon Bettscheider** and Andreas Zeller. "Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. ACM, 2024, pp. 522–526. DOI: 10.1145/3663529.3663790.

[25] Tim Blazytko et al. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1985–2002. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko.

[26] Marcel Böhme et al. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 2329–2344.

[27] Walter H. Burkhardt. "Generating test programs from syntax". In: *Computing* 2.1 (1967), pp. 53–73. DOI: 10.1007/BF02235512. URL: https://doi.org/10.1007/BF02235512.

[28] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[29] *CALC CPP Parser*. https://github.com/SaturnMatt/SimpleArithmeticParser/blob/main/Parser/parser.cpp. Accessed: 2025-02-10.

[30] *CGI-Decode*. https://github.com/vrthra/Cmimid/blob/master/examples/cgi_decode.c. Accessed: 2024-04-11.

[31] Hongxu Chen et al. "Hawkeye: Towards a Desired Directed Grey-box Fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, 2018, pp. 2095–2108. DOI: 10.1145/3243734.3243849. URL: https://doi.org/10.1145/3243734.3243849.

[32] Yaohui Chen et al. "SAVIOR: Towards Bug-Driven Hybrid Testing". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002. URL: https://doi.org/10.1109/SP40000.2020.00002.

[33] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Trans. Comput. Syst.*

30.1 (2012), 2:1–2:49. DOI: 10.1145/2110356.2110358. URL: https://doi.org/10.1145/2110356.2110358.

[34] Noam Chomsky. "Three models for the description of language". In: *IRE Trans. Inf. Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813. URL: https://doi.org/10.1109/TIT.1956.1056813.

[35] Alonzo Church. "A Note on the Entscheidungsproblem". In: *J. Symb. Log.* 1.1 (1936), pp. 40–41. DOI: 10.2307/2269326. URL: https://doi.org/10.2307/2269326.

[36] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd ed. Amsterdam, Boston, Heidelberg: Elsevier/Morgan Kaufmann, 2011. ISBN: 9780120884780.

[37] Thurston H. Y. Dang, José Pablo Cambronero, and Martin C. Rinard. "Inferring Drop-in Binary Parsers from Program Executions". In: *CoRR* abs/2104.09669 (2021). arXiv: 2104.09669. URL: https://arxiv.org/abs/2104.09669.

[38] Martin Davis, George Logemann, and Donald W. Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (1962), pp. 394–397. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[39] Zhengjie Du et al. "Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2440–2451. DOI: 10.1145/3510003.3510197. URL: https://doi.org/10.1145/3510003.3510197.

[40] Ted Dunning. "The t-digest: Efficient estimates of distributions". In: *Software Impacts* 7 (2021), p. 100049.

[41] Bruno Dutertre. "Yices 2.2". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744.

[42] Martin Eberlein et al. "Semantic Debugging". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. Ed. by Satish Chandra, Kelly Blincoe, and Paolo Tonella. ACM, 2023, pp. 438–449. DOI: 10.1145/3611643.3616296. URL: https://doi.org/10.1145/3611643.3616296.

[43] Max Eisele et al. "GDBMiner: Mining Precise Input Grammars on (Almost) Any System". In: *Leibniz Transactions on Embedded Systems* 10.1 (2025), pp. 1–1.

[44] Andrea Fioraldi et al. "{AFL++}: Combining incremental steps of fuzzing research". In: *14th USENIX workshop on offensive technologies (WOOT 20)*. 2020.

[45] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin et al. ACM, 2022, pp. 1051–1065. DOI: 10.1145/3548606.3560602. URL: https://doi.org/10.1145/3548606.3560602.

[46] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. "Comparison of Context-Free Grammars Based on Parsing Generated Test Data". In: *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4,*

2011, Revised Selected Papers. Ed. by Anthony M. Sloane and Uwe Aßmann. Vol. 6940. Lecture Notes in Computer Science. Springer, 2011, pp. 324–343. DOI: 10.1007/978-3-642-28830-2_18. URL: https://doi.org/10.1007/978-3-642-28830-2%5C_18.

[47] Dave Gamble and contributors. *cJSON: Ultralightweight JSON parser in ANSI C*. https://github.com/DaveGamble/cJSON/releases/tag/v1.7.18. Accessed: 2025-05-01. 2024.

[48] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 519–531. DOI: 10.1007/978-3-540-73368-3_52. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_52.

[49] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 206–215. DOI: 10.1145/1375581.1375607. URL: https://doi.org/10.1145/1375581.1375607.

[50] *Golden Grammar for LISP from ANTLRv4*. https://github.com/antlr/grammars-v4/blob/master/lisp/lisp.g4. Accessed: 2025-02-10.

[51] Rahul Gopinath. *ANTLR4 to FuzzingBook JSON Grammar Converter*. https://github.com/vrthra/antlr2json. Accessed: 2025-03-07.

[52] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "Mining input grammars from dynamic control flow". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 172–183. DOI: 10.1145/3368089.3409679. URL: https://doi.org/10.1145/3368089.3409679.

[53] Rahul Gopinath and Andreas Zeller. *Building Fast Fuzzers*. 2019. arXiv: 1911.07707 [cs.SE]. URL: https://arxiv.org/abs/1911.07707.

[54] Nikolas Havrikov and Andreas Zeller. "Systematically Covering Input Structure". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 189–199. DOI: 10.1109/ASE.2019.00027. URL: https://doi.org/10.1109/ASE.2019.00027.

[55] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: a grammar-based open source fuzzer". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*. Ed. by Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir. ACM, 2018, pp. 45–48. DOI: 10.1145/3278186.3278193. URL: https://doi.org/10.1145/3278186.3278193.

[56] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA,*

USA, August 8-10, 2012. Ed. by Tadayoshi Kohno. USENIX Association, 2012, pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[57] Matthias Höschele and Andreas Zeller. "Mining input grammars from dynamic taints". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 720–725. DOI: 10.1145/2970276.2970321. URL: https://doi.org/10.1145/2970276.2970321.

[58] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[59] *JSON Parser*. https://github.com/vrthra/mimid/blob/master/Cmimid/examples/json.c. Accessed: 2024-04-11.

[60] *JSON Parser*. https://github.com/DaveGamble/cJSON/blob/master/cJSON.c. Accessed: 2025-02-10.

[61] Alexander Kampmann et al. "When does my program do this? learning circumstances of software behavior". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 1228–1239. DOI: 10.1145/3368089.3409687. URL: https://doi.org/10.1145/3368089.3409687.

[62] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2%5C_9.

[63] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[64] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. "Learning Highly Recursive Input Grammars". In: *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 456–467. DOI: 10.1109/ASE51524.2021.9678879. URL: https://doi.org/10.1109/ASE51524.2021.9678879.

[65] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[66] Caroline Lemieux et al. "Perffuzz: Automatically generating pathological inputs". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 254–265.

[67] Feifei Li et al. “Incremental Context-free Grammar Inference in Black Box Settings”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. ACM, 2024, pp. 1171–1182. DOI: 10.1145/3691620.3695494. URL: https://doi.org/10.1145/3691620.3695494.

[68] Zhiqiang Lin and Xiangyu Zhang. “Deriving input syntactic structure from execution”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. Ed. by Mary Jean Harrold and Gail C. Murphy. ACM, 2008, pp. 83–93. DOI: 10.1145/1453101.1453114. URL: https://doi.org/10.1145/1453101.1453114.

[69] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. “Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications”. In: *IEEE Trans. Software Eng.* 36.5 (2010), pp. 688–703. DOI: 10.1109/TSE.2009.54. URL: https://doi.org/10.1109/TSE.2009.54.

[70] *LISP Parser*. https://github.com/mystor/simple-lisp-parser-in-c/blob/master/parse.c. Accessed: 2024-04-11.

[71] *LoopInfoWrapperPass (LLVM)*. https://llvm.org/doxygen/classllvm_1_1LoopInfoWrapperPass.html. Accessed: 2024-03-28.

[72] Alexandru Marginean et al. “Sapfix: Automated end-to-end repair at scale”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 269–278.

[73] Björn Mathis, Rahul Gopinath, and Andreas Zeller. “Learning input tokens for effective fuzzing”. In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. Ed. by Sarfraz Khurshid and Corina S. Pasareanu. ACM, 2020, pp. 27–37. DOI: 10.1145/3395363.3397348. URL: https://doi.org/10.1145/3395363.3397348.

[74] Björn Mathis et al. “Parser-directed fuzzing”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 548–560. DOI: 10.1145/3314221.3314651. URL: https://doi.org/10.1145/3314221.3314651.

[75] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[76] *mJS Parser*. https://github.com/cesanta/mjs/blob/master/mjs.c. Accessed: 2024-04-11.

[77] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/

978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[78] Rohan Padhye et al. "Fuzzfactory: domain-specific fuzzing with waypoints". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–29.

[79] Rohan Padhye et al. "Semantic fuzzing with zest". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. Ed. by Dongmei Zhang and Anders Møller. ACM, 2019, pp. 329–340. DOI: 10.1145/3293882.3330576. URL: https://doi.org/10.1145/3293882.3330576.

[80] Rohan Padhye et al. "Validity fuzzing and parametric generators for effective random testing". In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. IEEE / ACM, 2019, pp. 266–267. DOI: 10.1109/ICSE-COMPANION.2019.00107. URL: https://doi.org/10.1109/ICSE-Companion.2019.00107.

[81] Weiyu Pan et al. "Grammar-Agnostic Symbolic Execution by Token Symbolization". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 374–387.

[82] *PARSON Parser*. https://github.com/kgabis/parson/blob/master/parson.c. Accessed: 2025-02-10.

[83] Theofilos Petsios et al. "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 2155–2168.

[84] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[85] Van-Thuan Pham et al. "Smart Greybox Fuzzing". In: *IEEE Trans. Software Eng.* 47.9 (2021), pp. 1980–1997. DOI: 10.1109/TSE.2019.2941681. URL: https://doi.org/10.1109/TSE.2019.2941681.

[86] Andreas Pointner. "Mining Attributed Input Grammars and their Applications in Fuzzing". In: *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023, pp. 493–495. DOI: 10.1109/ICST57152.2023.00059. URL: https://doi.org/10.1109/ICST57152.2023.00059.

[87] Andreas Pointner, Josef Pichler, and Herbert Prähofer. "Inferring Attributed Grammars from Parser Implementations". In: *CoRR* abs/2507.13117 (2025). DOI: 10.48550/ARXIV.2507.13117. arXiv: 2507.13117. URL: https://doi.org/10.48550/arXiv.2507.13117.

[88] Paul Purdom. "A sentence generator for testing parsers". In: *BIT Numerical Mathematics* 12 (1972), pp. 366–375.

[89] *Reproducibility package for Arvada*. https://github.com/neil-kulkarni/arvada. Accessed: 2025-02-21.

[90] *Reproducibility package for Kedavra*. https://github.com/Sinpersrect/kedavra. Accessed: 2025-02-21.

[91] *Reproducibility package for Mimid*. https://github.com/vrthra/mimid. Accessed: 2025-02-21.

[92] *Reproducibility package for Stalagmite*. https://github.com/leonbett/stalagmite. Accessed: 2025-02-21.

[93] *Reproducibility package for TreeVada*. https://github.com/rifatarefin/treevada. Accessed: 2025-02-21.

[94] Microsoft Research. *Bitvectors*. Accessed: 2024-10-25. 2024. URL: https://microsoft.github.io/z3guide/docs/theories/Bitvectors/.

[95] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[96] Michael Schröder and Jürgen Cito. "Grammars for free: Toward grammar inference for Ad Hoc parsers". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 41–45. ISBN: 9781450392242. DOI: 10.1145/3510455.3512787. URL: https://doi.org/10.1145/3510455.3512787.

[97] Michael Schröder and Jürgen Cito. "Static Inference of Regular Grammars for Ad Hoc Parsers". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2 (2025), pp. 113–143.

[98] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*. Ed. by Gernot Heiser and Wilson C. Hsieh. USENIX Association, 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[99] Kostya Serebryany. *libFuzzer: a library for coverage-guided fuzz testing*. 2016. URL: https://llvm.org/docs/LibFuzzer.html (visited on 08/11/2025).

[100] João P. Marques Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433. URL: https://doi.org/10.1109/12.769433.

[101] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: CENGAGE Learning, 2013. ISBN: 978-1-133-18779-0.

[102] SMT-COMP Organizers. *SMT-COMP 2024: The Satisfiability Modulo Theories Competition*. Accessed: 2024-10-28. 2024. URL: https://smt-comp.github.io/2024/.

[103] SMT-LIB. *Logics*. Accessed: 2024-10-25. 2024. URL: https://smt-lib.org/logics.shtml.

[104] SMT-LIB Initiative. *SMT-LIB Theories*. https://smt-lib.org/theories.shtml. Accessed: 2024-10-29.

[105] Jonette M Stecklein et al. "Error cost escalation through the project life cycle". In: *14th Annual International Symposium*. JSC-CN-8435. 2004.

[106] Dominic Steinhöfel and Andreas Zeller. "Input invariants". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, Novem-*

ber 14-18, 2022. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 583–594. DOI: 10.1145/3540250.3549139. URL: https://doi.org/10.1145/3540250.3549139.

[107] Dominic Steinhöfel and Andreas Zeller. "Language-Based Software Testing". In: *Commun. ACM* 67.4 (2024), pp. 80–84. DOI: 10.1145/3631520. URL: https://doi.org/10.1145/3631520.

[108] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*. Ed. by Kunle Olukotun et al. IEEE Computer Society, 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186. URL: https://doi.org/10.1109/CGO.2015.7054186.

[109] STP Developers. *STP: A Decision Procedure for Bitvectors and Arrays*. https://stp.github.io/. Accessed: 2024-10-25.

[110] Robert Swiecki. *honggfuzz*. 2010. URL: https://github.com/google/honggfuzz (visited on 08/11/2025).

[111] The LLVM Project. *UndefinedBehaviorSanitizer (UBSan) — Clang Documentation*. 2013. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on 08/11/2025).

[112] *TinyC Parser*. https://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c. Accessed: 2024-04-11.

[113] Alan M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proc. London Math. Soc.* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[114] Samuel Ugochukwu. *LunaSVG: SVG Rendering Library, Version 3.2.1*. https://github.com/sammycage/lunasvg/releases/tag/v3.2.1. Accessed: 2025-05-01. 2025.

[115] *UnifyLoopExitsPass (LLVM)*. https://llvm.org/doxygen/classllvm_1_1UnifyLoopExitsPass.html. Accessed: 2024-03-28.

[116] Daniel Veillard and the libxml2 contributors. *libxml2: The XML C Parser and Toolkit, Version 2.9.7*. https://gitlab.gnome.org/GNOME/libxml2/-/releases/v2.9.7. Accessed: 2025-05-01. 2017.

[117] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 579–594. DOI: 10.1109/SP.2017.23. URL: https://doi.org/10.1109/SP.2017.23.

[118] Stephen Woodcock and Jay Falletta. "A numerical evaluation of the Finite Monkeys Theorem". In: *Franklin Open* (2024), p. 100171.

[119] Zhengkai Wu et al. "REINAM: reinforcement learning for input-grammar inference". In: *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2019, pp. 488–498.

[120] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

[121] Michal Zalewski. *American Fuzzy Lop (AFL) Fuzzer*. 2013. URL: https://lcamtuf.coredump.cx/afl/ (visited on 08/11/2025).

[122] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. “FANDANGO: Evolving Language-Based Testing”. In: *Proceedings of the ACM on Software Engineering* 2.ISSTA (2025), pp. 894–916.

[123] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[124] Andreas Zeller et al. “Efficient Grammar Fuzzing”. In: *The Fuzzing Book*. Retrieved 2023-11-11 18:18:06+01:00. CISPA Helmholtz Center for Information Security, 2023. URL: https://www.fuzzingbook.org/html/GrammarFuzzer.html.

[125] Andreas Zeller et al. “Mining Input Grammars”. In: *The Fuzzing Book*. Retrieved 2023-11-11 18:18:06+01:00. CISPA Helmholtz Center for Information Security, 2023. URL: https://www.fuzzingbook.org/html/GrammarMiner.html.

[126] Andreas Zeller et al. *The Fuzzing Book*. Retrieved 2024-07-01 16:50:18+02:00. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/.

[127] Xiaogang Zhu et al. “Fuzzing: a survey for roadmap”. In: *ACM Computing Surveys (CSUR)* 54.11s (2022), pp. 1–36.