Strategic Parser Fuzzing: Leveraging Domain Knowledge For Input Generation

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

by
Björn Mathis

Saarbrücken, 2024

Day of Colloquium

December 16th, 2024

Dean of the Faculty

Prof. Dr. Roland Speicher

Chair of the Committee

Prof. Bernd Finkbeiner, PhD

Reporter

Prof. Dr. Andreas Zeller
Prof. Dr. Thorsten Holz
Prof. Dr. Gordon Fraser

Scientific Assistant

Dr. Dominic Steinhöfel

Zusammenfassung

Das Testen von Software ist einer der wichtigsten Teile des Entwicklungsprozesses. Ohne Tests würden Programme häufig abstürzen oder Sicherheitslücken aufweisen. Da Testen zeitaufwändig und komplex ist, wurden Techniken entwickelt, um diesen Prozess zu vereinfachen. Zum Beispiel erstellen Fuzzer weitestgehend zufällige Eingaben und testen, wie ein Programm auf diese reagiert.

Insbesondere Software, die Parser zur Eingabeverarbeitung nutzt, stufen wir als interessant ein, aber sie ist auch schwer automatisiert mit Allzwecktechniken zu testen – sie können die syntaktisch korrekten Eingaben zum Testen der Programmlogik nicht generieren. Deshalb präsentieren wir einen neuen Ansatz speziell zur Analyse von ***Software mit rekursiv absteigenden Parsern****.*

Eine zentrale Eigenschaft von Parsern sind die iterativen Vergleiche von Eingabeteilen gegen die Terminale einer entsprechenden kontextfreien Grammatik. Wir zeigen, wie man die Vergleiche aus einer Programmausführung mit anderen, parserspezifischen Eigenschaften kombinieren kann, um syntaktisch valide und diverse Eingaben zu inferieren. In unserer Evaluation kombinieren wir unsere Techniken mit dem Fuzzer AFL. Unsere generierten Eingaben enthalten im Durchschnitt 77,7 % aller möglichen Lexeme mit mehr als drei Schriftzeichen. Wir erhalten eine durchschnittlich um 2,9 und im besten Fall bis zu 17 Prozentpunkte höhere Zweigüberdeckung im Vergleich zu einem Alleinlauf von AFL.

Abstract

Testing software is one of the most important parts of the development process. Without tests, programs would often crash or contain security vulnerabilities. Since testing is time-consuming and complex, techniques were developed to simplify this process. For instance, fuzzers create mostly random inputs and test how a program reacts to those.

Especially software that uses parser for input processing is classified as interesting by us, but it is also hard to automatically test it with general purpose techniques—they cannot generate the syntactically correct inputs to test the program logic. Thus, we present a new approach specifically for analyzing ***software with recursive descent parsers****.*

A central feature of parsers are the iterative comparisons of parts of the input against terminals of a respective context-free grammar. We show, how those comparisons from a program execution can be combined with other, parser-specific features to infer syntactically valid and diverse inputs. In our evaluation we combine our techniques with the fuzzer AFL. Our generated inputs contain on average 77.7% of all possible lexemes with more than three characters. We obtain on average a 2.9 and in best case up to 17 percentage points higher branch coverage in comparison to running AFL alone.

Acknowledgements

This thesis is the result of years of research and such a path cannot be taken alone. A lot of people were part of this journey, thus I would like to express my gratitude here.

First, I would like to thank my advisor Andreas Zeller. For sure, this thesis would not exist without your outstanding help and advice in the last few years! As a source of infinite ideas and someone who always had the right critical questions about my thoughts and results, you always provided the needed guidance.

Also, I would like to thank all my colleagues I worked with throughout the years, it was highly inspiring and a pleasure working with you. I enjoyed our discussions during lunch, at chair retreats, or simply in office, you always had an open mind and the right questions and ideas. For every topic there were plenty of open doors with people happy to discuss them—this is nothing that should be taken for granted.

Additionally, I would like to thank all my co-authors. Hard problems need more than one person to work on and I am happy to have worked with you on the solutions. As a team, we were able to work on our goals from various perspectives, with valuable ideas, which ended up in great results.

Furthermore, I would like to thank my family and friends, who supported me all this time and made it possible to concentrate as much as needed on my research and dissertation. Certainly, without your support it would have been much harder to achieve all of this.

Finally, this work was partially supported with funds from the Bosch Research Foundation in the Stifterverband (Reference: T113/33825/19). Thank you for giving me the opportunity to meet great people and discuss very interesting topics at the various events and other occasions.

Contents

List of Figures

List of Tables

1 Introduction

In a perfect world people would have great ideas that can be digitally solved; those ideas would then be realized into digital products to solve the given problems; these products are delivered to the customers; and those customers use those tools to increase efficiency, enable more precise processes, or just use them for fun in their leisure time. Unfortunately, many things can go wrong along this path, one of the most prevalent ones are bugs. Whenever code deviates from the original idea—from the specification—it might end up in undefined states, which in turn is a problem for the end user. Starting with small bugs that just make the GUI look bad or the pretty printing of an output might not work, up to more serious bugs that let the whole program crash, possibly even leaving data in an undefined state. While in some cases a bug is just an annoying disturbance in the program execution which does not cause any harm (a button is harder to click because it is presented slightly shifted on the GUI), in other cases it might happen that money or valuable goods are lost (the payment process of an online shop crashes or the load controller of a truck randomly unloads the load bed), and in the worst case even lives are threatened (an X-ray machine may use a too high dosage or an important controller in a car or plane crashes, causing the vehicle to get into an unstable state).

Obviously, developers and companies do not deliver code or final products untested. In most cases there is at least some process involved to assure the quality of a product, depending on the use case with more or less regulations and with more or less depth of the testing process. Testing needs time and if one person writes a small tool which is used by some people the motivation to have it tested well is low. When it comes to more critical systems or commercial importance the quality of the product needs to be assured, be it for customer satisfaction, legal reasons or just to avoid being sued. While this costs more resources during development, it also reduces the risk of bugs emerging in production which may cost magnitudes more compared to those bugs found before the delivery of a product.

There are many more reasons why testing should be done or why things might not be tested. In all those cases, be it a small one person project realising a special purpose tool or a large company with a critical system, it needs to be decided how many resources are used to cover the different scenarios the product needs to handle. And not only this, even if resources are given to the testing part of development, many decisions have to be made, e.g. which tools to use, which test inputs to create, and which scenarios and features should be covered.

Manual Tests One option is to perform or write tests by hand, for example at a unit or system level—or any other level one can come up with. This is time consuming though and one needs to come up with all the different scenarios that might arise in the real world to actually cover the functionality of the subject under test properly.

Static Tools Another option is to cover all possible scenarios at once using a static code analysis tool. In theory, a static technique would be able to analyze the code, mark all code locations or paths that are incorrect and the developer can fix those. In practice though, those tools are not yet ready to fulfill this requirement and may never be able to do so in all generality.

Dynamic Tools If one wants to see how the code performs when actually executed on one (or more) machines without the need to write tests manually, many tools exist that aid this process and help with testing the subject under test efficiently. There are options like model based testing approaches, which not only take the software as a target, but also use some input or program model [60, 63] as a baseline for generating tests. Such semi-automated approaches profit from the fact that on the one hand tests can be generated very targeted to the subject under test (the given model defines the quality of the generated tests), but do not require the developer to focus on each and every feature on its own in depth. And even if one does not want to write any model and favors a push button approach, there are often options to do exactly that. For example, the famous *"american fuzzy lop"* fuzzer AFL [159] just needs a program to be compiled with some injected instructions to monitor the program execution (it already comes with a compiler for C performing this task). Once the subject under test is compiled, AFL just needs to know how to start the subject and how inputs are given to the program. Then it is able to explore the input space and in turn covers the different features and parts of the code automatically.

Special Case: Fuzzing Fuzzing is a special case of dynamic testing, having randomness as one of the most important factors while exploring the input space. Certainly, every fuzzer is a dynamic testing tool, but not every dynamic testing tool is a fuzzer—still, there is no clear line to draw regarding what can be counted as fuzzing and what not. Fuzzing started with the idea in mind that if a program accepts an input in any form it must correctly handle every input given to it. This means for valid inputs the subject should actually produce an output and for all other inputs it should gracefully reject the input—specifically without crashing. Now, one could write many of those inputs by hand, or design an input model that defines how the input should look like. But the original idea is much simpler and more automated: let a program generate a random input stream and see how the subject under test reacts to this [11]. And even up until today this is a valid idea to test programs and find bugs [110].

While there are many options to improve software quality (much more than listed above), fuzzing certainly is a well known and used technique to test software products during development, before releases, and even while the code is already in production [1]. **Therefore, this thesis concentrates on how to support fuzzers—more concretely how putting generic domain knowledge into the fuzzing pipeline can improve the overall performance of fuzzing.** Still, we cannot evaluate all the different software domains out there and see how we can apply domain knowledge, hence we pick the **domain of input parsing programs** and evaluate how, if we already know **the subject under test parses inputs with a recursive**

descent parser, we can design a tool that infers more specific knowledge about the subject. As we will later see, our approach is not specifically or solely designed to improve fuzzing. It is rather an analysis technique which can be used in a fuzzing pipeline and we will evaluate in such an environment, but other techniques like grammar learning [54, 65, 81] could also profit from the results our approach produces.

Even though fuzzing is known for decades and a well established method for automatic program testing it still has its limitations. One of the most prominent ones are so called fuzzing roadblocks, e.g. code locations which require a very specific and possibly complex input. Thus, modern generic fuzzers are not completely random anymore. For example, AFL monitors the program execution and extracts information about paths taken for each input executed. Another typical approach is to give one or more complex inputs as samples to the fuzzer and let it reassemble those samples in different ways, making it possible to guide the fuzzer to such otherwise blocked locations and then let it explore the close proximity in the code. Finally, it is also possible to monitor the code for such locations, or in other words: if the code checks for *"magic values"* like concrete numbers or fixed strings it might be interesting to extract those values and apply them to future generated inputs. It is highly likely that such values are needed at some point in the input to satisfy respective conditions using such a value.

Let us have a look at the code in Figure 1.1. In the code we see two functions, one for lexing the input (Line 2) and one for parsing (Line 12). We already talked about *"magic values"*, and especially in the *domain of parsers* such magic values appear very often—every keyword of the underlying language resembles a magic value. In our example we can see such a magic value comparison in Line 5—here we try to lex the input by matching a portion of the input with the string `"sin("`. And not only this, for parsers it is also important in which order the magic values appear in the input. Just as with a natural, spoken language one cannot randomly combine different valid words to a valid sentence (or input when talking about programs). In our example, the token generated by our lexer (Line 7) is parsed in Line 14. Now, the parser requires the next tokens to build a valid expression (Line 16). Thus the next token must be the start of a valid expression, it cannot be an arbitrary valid token. Only if valid words/tokens are correctly combined an input is syntactically valid.

Thus, we want to answer the following question in this thesis: **how can we extract information from the subject under test to fuzz parsers efficiently and without subject specific knowledge**? While producing some input model and letting the fuzzer generate inputs based on this is an option, it has its drawbacks—mainly the manual effort needed to create and maintain such a model. Therefore, we decided to find a way to give the fuzzer *generic domain knowledge about parsers*, which can then be used to infer *subject specific information* that can be used to guide a fuzzer through the code. We use domain knowledge like the *separation of lexer and parser* (as we can see in Line 2 and Line 12) in the code and the requirement of *direct input to magic value comparisons* (as we can see in Line 5) to build a lexer. We present an approach to *analyze parsers* in such a way that we can extract information *out-of-thin-air*, which can be used by subsequent techniques like fuzzing: *a push-button analyzer for recursive descent parsers*.

```
1  //...
2  void next_token_non_whitespace() {
3  //...
4    else if (pos + 4 < input_size &&
5      !strncmp("sin(", input + pos, 4)) {
6    pos += 4;
7    token = SIN;
8    }
9  //...
10 }
11 //...
12 int atom() {
13   printf("ATOM: Parsing %c at pos %d\n", input[pos], pos);
14   if (token == PAREN_L || token == SIN || token == COS) {
15     next_token();
16     if (expr()) {
17 //...
18     }
19   }
20   return integer();
21 }
22 //...
```

Figure 1.1: A short snippet from an arithmetic parser with lexer, which is fully presented in Appendix A-II.

1.1 Objectives

We already talked about push button approaches for fuzzing, and in general they already work well—maybe even surprisingly well considering the simple approach of delivering (mostly) random inputs to the subject under test. Still, when it comes to subjects with stricter input formats and resulting fuzzing roadblocks, such dominantly random tools are in a disadvantage as they need to guess correct input options and need many trials to find valid input parts. Since such push button fuzzers are designed to produce inputs for any kind of software and any domain they are applied to, they need to be as generic as possible. Hence, even if the subject under test only accepts a small set of ASCII characters in its inputs (similar to the comparisons in our example lexer like in Line 5 in Figure 1.1), a generic fuzzer would need to consider any byte sequence it can possibly produce and find out which byte sequences cover new portions of the code and which sequences improve the overall path to the fuzzing goal.

Analyzing Recursive Descent Parsers Arguably, an interesting problem is typically not an easy problem. If it was easy, it would have been already solved and made public or worse: solved and determined as easy enough to not even bother with sharing the solution. Our goal is to find a way to target input parsing programs such that new inputs can be generated more precisely than using random guessing. Recursive descent parsers are on the one hand sufficiently complex,

widely spread, and well used to be considered interesting and on the other hand have enough common features that are typically used and can be leveraged to create a domain specific tool for extracting information about the subject under test, which can be fed into a subsequent tool like a fuzzer to target the programs in this domain more specifically. Also, we want to stay close to the simplicity of a generic push button tool like AFL, which requires nothing more than a special compilation pass and information on how to give inputs to the subject under test—it can be run on any subject without much additional information or effort. Thus, we have to find out what the different subjects in the domain of recursive descent parsers have in common that is on the one hand a problem for generic fuzzers but on the other hand can be described generically for the whole domain for our approach. Or in other words: **what commonalities do the different subjects of the domain have that we can leverage for more efficient analysis of said subjects to generate inputs that are syntactically valid**?

Domain Vs. Subject Specific One might think that a domain specific fuzzer is nothing more than a specialized tool for a few subjects, for example one could write a fuzzer for the domain of C compilers. But, this is not what we have in mind when we talk about a domain in this thesis—we would rather consider such an approach as some form of subject specific, even though there is more than one implementation of a C compiler [137, 140]. It is arguably a fruitful idea to also fuzz C compilers very specifically with such a fuzzer, because this topic and the implementation is complex enough. Still, in this case when we speak about a domain, it would contain all compilers, not only C compilers. For our idea to work, a domain should have one or more overarching features that are common among all implementations and should, in best case, be unavoidable.

For example, a subject specific fuzzer for our example arithmetic parser code snippet in Figure 1.1 would specifically implement the generation of `"sin("` as part of an input (because this is a valid portion of an input as we can see in the lexing part in Line 5). This is then followed by an expression, as specified in our example in Line 16. While this is certainly correct for our example, it would not be correct for subjects that have other input languages. A domain specific tool on the other hand would only make use of common features. In our example a tool that is designed to work on recursive descent parsers could assume that the subject has at least a parsing phase (Line 12) and likely also a lexing phase (Line 2), that the code for the two phases lives in different regions, and that portions of the input are directly compared during parsing/lexing (Line 5).

We decided for the domain of **recursive descent parsers**, hence we plan to solve the following three main objectives for this thesis:

Objective 1: Diverse Syntactically Valid Inputs One of the main problems when targeting parsers in fuzzing are the values used in comparisons as they are constant characters or strings. Those are needed in the parser to test if a certain part of the input is actually valid. Such values are hard to randomly guess, especially for *keywords*, i.e. longer constant strings. Keywords are often atomically checked while parsing, hence a fuzzer can only

advance once the keyword is fully guessed and might not get any feedback about partially correct results. Secondly, even if keywords are known, a parser typically requires them to be in a specific order. Hence, even if portions of the input are valid, the input will get rejected if at least one keyword is not at the correct position. Thus, with current techniques, a lot of resources go into guessing keywords and their ordering. Our approach to solve this objective is discussed in Chapter 3.

Objective 2: Tokenizer Analysis Beyond the above mentioned comparisons, more complex input processors do not only rely on direct input validation, but use a two stage verification process: lexing and parsing. This yields a second indirection during input processing: first the portions of an input are checked for generic validity (*"is the input portion a valid keyword"*), then they are checked for correct ordering. Without a lexer, if a portion of an input was compared against some constant, chances were high that this constant is a valid keyword **and** valid at this position. With the presence of a lexer, this constant is still likely valid, but we have no information about the ordering anymore (the input character comparison happens in the lexer, the parser typically never processes input characters). The validation of keyword correctness and ordering correctness is split, which yields additional execution paths that end up in parsing errors, increasing the overall search space while trying to generate inputs that go beyond the parser. Our approach to solve this objective is discussed in Chapter 4.

Objective 3: Syntactic Correctness And Semantic Variety Once the issues of keyword and ordering detection are solved we are able to generate inputs that go beyond the parsing stage. Typically, once an input passes the parser, a program processes the information this input represents. Thus, it is not only important to generate a few inputs that pass the input validation, we also need a wide variety of such inputs—syntactically but also semantically. Semantic variety can be achieved by altering generic syntactic elements (often *numbers*, *names*, or *strings* allow a wide range of different values) or by recombining valid syntactic blocks (often consisting of more than one syntactic element). Current techniques are already good at doing those two things, but they require some inputs with syntactic variety to alter syntactic elements and recombine syntactic blocks. Our approach to solve this objective is discussed in Chapter 5.

1.2 Contributions

This section describes the core contributions of this thesis to achieve our goal: *understand which inputs a parsing program wants by using nothing more than source code instrumentation and sample executions*. We make the following contributions specific to the domain of input parsers (in detail to recursive descent parsers) to solve the objectives outlined in Section 1.1:

Contribution 1: Diverse Syntactically Valid Inputs Recursive descent parsers are bound to compare each and every syntactic element of an input against its known keywords; using magic values in the parser comparisons. We make use of this requirement by iteratively adding the magic values observed during subject under test execution (by

instrumenting the program) to already generated inputs (using heuristics to find the most promising substitutions and additions), finally generating syntactically valid inputs. With this technique we want to solve the problem of guessing such keywords, which greatly reduces the search space and thus makes it possible to **generate a wide variety of syntactically valid inputs** specifically for the subject under test—all of this without subject specific knowledge.

Contribution 2: Tokenizer Analysis Tokenizers generally follow certain patterns while generating token values from syntactic elements. We can detect those patterns during program executions and extend our analysis to also include information flow from the tokenizer to the actual parser. This solves two issues at once: **knowledge about the space of keywords and insight into the ordering of keywords**. We collect most keywords known to the subject by observing the comparisons made in the tokenizer. This gives us the syntactic elements every input is built of—*hence the set of keywords*. Second, by tracking information flowing from the input over the tokenizer into the parser, we avoid guessing the ordering of the valid input portions, but have a direct link between input characters and parser comparisons—reducing our problem to *Objective 1*.

Contribution 3: Syntactic Correctness And Semantic Variety The needed analysis of the subject under test for our contributions slows down the execution speed of the program, hence we combine it with a lightweight analysis and instrumentation. First, we infer a set of keywords and input samples, then we run a more lightweight tool using this information. This solves the last objective mentioned: **syntactically correct inputs with a high semantic diversity**. The sample inputs serve as blueprints for recombinations, the set of valid keywords are the building blocks for different mutations, increasing the chance of generating syntactically valid inputs even with lightweight but fast approaches.

We developed two tools: pFuzzer, which only contains the ideas of Contribution 1 and lFuzzer, which contains all contributions. In our evaluation in Chapter 6 we evaluate those tools, finding out how they compare to one of the *state-of-the-art* push-button fuzzers: AFL [160]. With lFuzzer (which incorporates AFL) the resulting inputs contain on average almost 78% of all possible lexemes with more than three characters, showing that our approach is able to generate syntactically diverse inputs.[1] With those inputs we achieve an average branch coverage increase of 2.9 percentage points with up to 17 percentage points in the best case compared to running AFL alone. This shows that the combination of a domain specific subject analysis and *state-of-the-art* fuzzing is able to improve over the *state-of-the-art* approach alone.

1.3 Thesis Structure

Chapter 2 gives an overview on grammars, parsers, input generators, and dynamic tainting (as this technique is later used in our approaches for runtime information retrieval). It draws the theoretical background our approach is based on.

[1] We also combined pFuzzer with AFL, showing how Contribution 2 influences the results.

Chapter 3 starts the technical content of the thesis by describing how lexical elements can be extracted and used to compose valid inputs—**it solves Objective 1: the generation of inputs based on complex comparisons with magic values**. We will describe the approach and explain the limitations and assumptions of this method which lays the foundation for further extensions as detailed in Chapter 4.

Chapter 4 details how using lexical elements only is not sufficient for generating valid inputs, especially in the context of complex input validators—**it solves Objective 2: the generation of inputs in the presence of a tokenizer**. It includes the description of how it is possible to trace lexical elements beyond the lexical stage of an input validator into the parser and enable the techniques from Chapter 3 on more complex subjects. We will also detail the limitations and assumptions, showing the differences to the limitations and assumptions of Chapter 3 introduced by the presented new techniques.

Chapter 5 combines the techniques from Chapter 3 and Chapter 4 with *state-of-the-art* greybox fuzzing to improve the overall variety of inputs generated—**it solves Objective 3: the generation of inputs that are syntactically valid and semantically diverse**. Furthermore, we give an outlook on what could be done to improve the integration even more.

Chapter 6 evaluates the different contributions of our approach in comparison with the *state-of-the-art* fuzzer AFL as a standalone tool. Here we show how syntactically diverse the different generated inputs are in comparison to the AFL generated ones and we show that we can achieve more coverage by applying our analysis technique in combination with AFL compared to AFL standalone. This chapter also lists the different threats to validity we introduce with the design choices we made. Here we detail how the implementation design choices might influence the results and how the evaluation design choices might influence the outcome and generalizability of the evaluation.

Chapter 7 summarizes the related work done in the field of fuzzing and puts our approach in context of this work. Based on this we will detail how this dissertation improves the *state-of-the-art*, extends on existing techniques, and integrates with the current research to improve fuzzing in areas that were previously hard to target.

Chapter 8 lists the key takeaways of this thesis and serves as a detailed collection of what we achieved with our research. It also sums up how this work improved the current *state-of-the-art*, the impact it might have, and the future research that can be done in this field. We conclude this work with some final words.

1.4 Publications And Claim Of Authorship

This whole thesis builds on and includes the research done for the following publications, hence they serve as an overarching reference for all chapters. The research papers are listed in descending chronical order; the author of this thesis is highlighted in bold for each publication:

Learning Input Tokens for Effective Fuzzing [102]
Björn Mathis, Rahul Gopinath, and Andreas Zeller. In proceedings of the 29TH ACM SIGSOFT International Symposium on Software Testing and Analysis—2020.

Parser-Directed Fuzzing [105]
Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. In proceedings of the 40TH ACM SIGPLAN Conference on Programming Language Design and Implementation—2019.

Dynamic Tainting for Automatic Test Case Generation [100]
Björn Mathis. In proceedings of the Doctoral Symposium co-located with the 26TH ACM SIGSOFT International Symposium on Software Testing and Analysis—2017.

Furthermore, the following papers were published but are not part of this thesis:

Systematic Assessment of Fuzzers using Mutation Analysis [58]
Philipp Görz, **Björn Mathis**, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. In proceedings of the 32nd USENIX Security Symposium—2023.

Mining Input Grammars [55]
Rahul Gopinath, **Björn Mathis**, and Andreas Zeller. Software Engineering 2021—2021.

Mining Input Grammars from Dynamic Control Flow [54]
Rahul Gopinath, **Björn Mathis**, and Andreas Zeller. In proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering—2020.

If You Can't Kill a Supermutant, You Have a Problem [53]
Rahul Gopinath, **Björn Mathis**, and Andreas Zeller. In proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops—2018.

Detecting Information Flow by Mutating Input Data [104]
Björn Mathis, Vitalii Avdiienko, Ezekiel Soremekun, Marcel Böhme, Andreas Zeller. In proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering—2017.

Claim Of Authorship During the research of AutoGram [65], Höschele et al. as well as the other co-authors of our paper *Parser-Directed Fuzzing* [105] thought about inferring inputs instead of using existing inputs for their approach. While first experiments were done at that time with the tainting engine designed for AutoGram, the problem was postponed as other research got more focus. During my master thesis I built a tainting engine which provides

similar information to AutoGram like the original tainting algorithm, but instead of analyzing Java code, it works on C code. At that time the idea to infer inputs from program executions was renewed, and while writing *Dynamic Tainting for Automatic Test Case Generation* [100] I briefly discussed how to realize the incorporation of dynamic tainting for input generation.

In detail, I contributed the following to the overall approach presented in this thesis[2]:

- the largest part of the overall approach of iteratively building syntactically valid inputs from executions with input samples as presented in Chapter 3 (and in our *Parser-Directed Fuzzing* paper [105])—especially the heuristics to properly sort the different input character comparisons to improve the knowledge gain for each run and make the input generation possible; some smaller details were contributed by others
- the extension to tokenizing code as discussed in Chapter 4 (and in our *Learning Input Tokens for Effective Fuzzing* paper [102])
- the combination with fuzzing techniques as presented in Chapter 5
- most parts of the implementation—some fixes and additions to the code base were done by others
- the utmost portion of the evaluation as shown in Chapter 6

[2]While being as concrete as possible about the contributions, after years of research together with co-authors it is impossible to list every detailed contribution of every author. As with almost every idea, this thesis is a combination and the result of many fruitful discussions, but while the basic idea of somehow leveraging dynamic tainting for input generation stems from the time of writing AutoGram, the devil lies in the details. This idea was also very briefly mentioned in one paragraph in my master thesis [101]. Also, a basic implementation of our approach (using comparison-values from the program execution as substitutions to generate new inputs) was presented in our preprint *"Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing"* [56]. The utmost portion of the ideas and results presented in this thesis were done by me.

2 | Background

Before we detail our work done in the area of parser analysis and fuzzing, we have a look into the background of this topic. Starting with *grammars*, we give a short overview on the different types as defined in the Chomsky Hierarchy [29]. Then we talk about *context-free grammars* as our main focus lies on parsers based on this type of grammars. Subsequently, we talk about the different types of *parsers* and then focus on *recursive descent parsers*, which are the most common form of manually written parsers for context-free grammars and thus our target domain. In Section 2.3, we talk about *input generation*, also in the context of generating inputs for parsers. Finally, we give some background on *dynamic tainting*, which will later be used in our approach as the fundamental program analysis technique.

2.1 Grammars

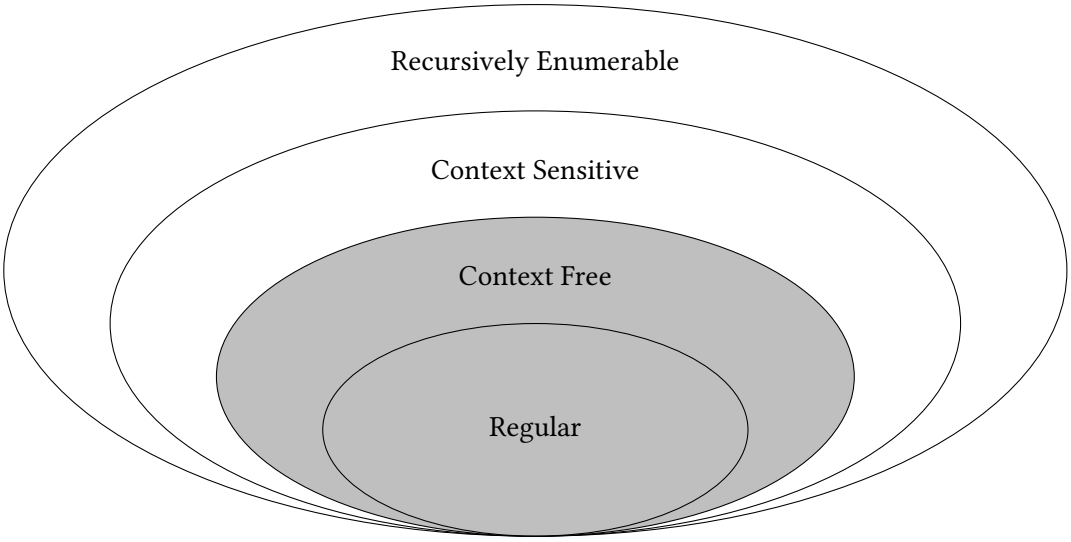


Figure 2.1: The Chomsky Hierarchy for language theory [29]. We will focus on context free grammars in this dissertation.

In Figure 2.1 we see the famous *Chomsky Hierarchy*, the canonical classification system for grammars, taught in nearly any class on theoretical computer science. The Chomsky Hierarchy defines different levels of expressiveness for the different layers of the hierarchy, each layer subsumes the other, shown in Figure 2.1 with the different bubble sizes. One question might be:

"why do we need different levels of grammars, why can't we just express everything in a recursively enumerable language". The answer is again rather practical: with increasing language power comes an increasing parser complexity, e.g. a context-free grammar can be parsed in $O(n^3)$ [36], in fact even slightly faster [148]. A context sensitive language parser though cannot decide in general in polynomial time if a word is a member of a context sensitive language [77] and is as such impractical to use in a real world context. Thus, classifying a given language into the Chomsky Hierarchy also gives guarantees on how complex a parsing process for the language will be in worst case.

A language is defined as a set of sentences over an alphabet, whereas a *"sentence over an alphabet is any string of finite length composed of symbols from the alphabet"* [64] and an *"alphabet or vocabulary is any finite set of symbols"* [64] (a symbol could be for example an ASCII character).

Definition 2.1.1. *"If V is an alphabet, then V^* denotes the set of all sentences composed of symbols of V, including the empty sentence. We use V^+ to denote the set $V^* - \{\epsilon\}$. Thus, if $V = \{0, 1\}$, then $V^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ and $V^+ = \{0, 1, 00, \ldots\}$" [64].*

"The concept of a grammar was originally formalized by linguists in their study of natural languages. Linguists were concerned not only with defining precisely what is or is not a valid sentence of a language, but also with providing structural descriptions of the sentences." [64].

Before we talk about context-free grammars, the grammars we will focus on in this thesis, let us give a general overview on the different levels of the Chomsky Hierarchy and what they express. In this section we will focus on the grammar theory side of those levels. In particular, we shortly explain what is needed to recognize an input as part of a grammar of the respective type.

Definition 2.1.2. *"Formally, we denote a grammar G by (V_N, V_T, P, S). The symbols V_N, V_T, P, and S are, respectively, the variables, terminals, productions, and start symbol. V_N, V_T, and P are finite sets. We assume that V_N and V_T contain no elements in common; that is, $V_N \cap V_T = \varphi[\ldots]$. We conventionally denote $V_N \cup V_T$ by V. The set of productions P consists of expressions of the form $\alpha \to \beta$, where α is a string in V^+ and β is a string in V^*. Finally, S is always a symbol in V_N" [64].*[1]

For each level of the Chomsky Hierarchy there are different restrictions to how the production rules P may look like, whereas regular expressions have the strongest restrictions and recursively enumerable grammars have the least restrictions. In the next paragraphs we bring context-free grammars in context with the remaining levels of the Chomsky Hierarchy while following the

[1] Hopcroft and Ullman use φ as the symbol for the empty set. In this thesis we will use $\emptyset$ outside of citations to represent the empty set. Also, left hand-side and right hand-side of a production is typically divided by the symbol "$\to$"; in this thesis though we use the character "$=$" in the concrete grammars we present (hence outside of general formal definitions), as we follow the notation of the Fuzzingbook [161] (more on this in Section 2.1.1).

explanations of Hopcroft and Ullman [64]. *We omit the formal definitions of regular, context-sensitive, and recursively enumerable languages, because our approach and this thesis only analyzes parsers for context-free grammars.*

Regular Expressions The least expressive level in the Chomsky Hierarchy are *regular expressions*. In a regular expression grammar it is only allowed to use one nonterminal symbol on the left hand-side and the right hand-side must consist of either exactly one terminal symbol or a terminal symbol followed by a nonterminal symbol [64]. Thus, each production step of the grammar makes the resulting string exactly one terminal symbol longer.

Context-Free Grammars The next level in expressiveness with respect to the Chomsky Hierarchy are *context-free grammars*—the grammars that serve as a foundation for recursive descent parsers, the target parsers for our analysis.

Definition 2.1.3. *Formally spoken, a context-free grammar is a four tuple consisting of the following elements [64]:*

1. *Set of terminal symbols V_T*
2. *Set of nonterminal symbols V_N*
3. *One nonterminal symbol, which is the start symbol S.*
4. *A set of productions $\alpha \to \beta$, with each production being:*
   a) $\alpha \in V_N$.
   b) $\beta \in V^+$

In contrast to regular expressions, context-free grammars need not generate one terminal symbol in each production step, but they are **not** allowed to replace a nonterminal symbol with the empty string—informally, they cannot delete any nonterminal symbol once produced.

Context-Sensitive Grammars The left hand-side part in the productions of *context-sensitive grammars* is the considerable difference to context-free grammars, as it allows a context, e.g. one could have a left hand-side $\alpha_1 A \alpha_2$ and a right hand-side $\alpha_1 \beta \alpha_2$ with $\alpha_1 \in V^*$, $\alpha_2 \in V^*$, $\beta \in V^+$, and $A \in V_N$. In this concrete example we would replace A with β, but only if the characters in front of A match α_1 and the characters following A match α_2. A must be in the context of α_1 and α_2. While in context-free grammars the left hand-side can only contain one nonterminal symbol, in context-sensitive grammars the context must be considered when applying a production rule—in context-free grammars we can just replace one terminal symbol with one of its right hand-sides, in context-sensitive grammars the full left hand-side must match and will be fully replaced with the right hand-side. Thus, context-free grammars are even more expressive than context-sensitive grammars. Still, the overall length of the derivation string grows monotonic, as the right hand-side needs to be at least as large as the left hand-side.

Recursively Enumerable Grammars The most powerful grammars are *recursively enumerable grammars* as they do not have any limitation on their design. In contrast to context-sensitive grammars, recursively enumerable grammars can have production rules with a right hand-side which is shorter than the left hand-side, hence during production of words from a grammar, the derivation string can shrink—something that cannot happen for all other grammars.

Relaxed Hierarchies While the Chomsky Hierarchy sharply distinguishes its different levels, researchers invented other forms of grammars in the past decades which are more specialized to their specific use case, and as such allow approximations and limitations that let them accept a subset of languages of different levels of the Chomsky Hierarchy while allowing fast parser implementations. A *parser expression grammar* allows the parsing of some non-context-free features while being able to parse inputs in linear time [43]. *Attribute grammars* [80] can be used to attach a semantic meaning to a context-free grammar, or more precisely, to its production rules. For example, in his paper Knuth presents a grammar which, once the input string representing the binary number is parsed, produces the decimal value of this input [80].

In this work we will focus on *context-free grammars* as those are the most commonly used grammars in real world parsers (including the above mentioned adaptions and extensions) [37, 59, 137, 140][2]. They are expressive enough to represent complex language constructs while parsers exist that are sufficiently fast to parse them [148]—there are even parsing strategies that are slightly more restricted than generic parsers for context-free grammars, but still very close and allow faster parsing, even linear time parsing, and in some cases they even allow parsing of non-context-free features [43]. Thus, context-free grammars are the perfect target for our research[3]: used in real world but sufficiently complex such that trivial methods cannot fully automatically test systems that are based on them.

2.1.1 Context-Free Grammars

Being expressive enough to describe complex inputs but also restricted enough to parse those inputs efficiently, context-free grammars are often used as a foundation to build a parser on, e.g. for parser generators like Bison [44]. Popular compilers like Clang and GCC use recursive descent parsers for parsing, built on context-free grammars [37, 137, 140] (even if no concrete context-free grammar is documented). As such, recursive descent parsers are a perfect target for specialized fuzzing, they are often used and are the frontend and input validator of many programs. In Section 2.2 we will take a closer look on those parsers, but first we need to lay some foundations for their underlying grammar format: *context-free grammars*.

Syntax Before we go into details, we need to introduce some terminology and syntax which will be used throughout this dissertation. In Definition 2.1.2 we have already defined a grammar

[2] The developers of Python switched from an LL(*1*) grammar to a parser expression grammar [59, 121], still, this example shows that even such complex systems make use of context-free grammars.

[3] Though, our research concentrates on recursive descent parsers, which indeed parse context-free grammars, but we want to highlight that we cannot claim to be able to handle all context-free grammar parsers with the research done in this thesis.

in general, and in Definition 2.1.3 we presented a definition for context-free grammars. This abstract definition needs to be put into a usable syntax to describe a grammar in a human readable as well as machine interpretable form. In this thesis we use the grammar style as introduced by the Fuzzingbook [161] (using "=" instead of → for production rules): Figure 2.2 shows a context free grammar which represents the language of arithmetic expressions.

```
<START> = <EXPR>
 <EXPR> = <TERM> + <EXPR> | <TERM> - <EXPR> | <TERM>
 <TERM> = <ATOM> * <TERM> | <ATOM> / <TERM> | <ATOM>
 <ATOM> = ( <EXPR> ) | sin( <EXPR> ) | cos( <EXPR> ) | <INT>
  <INT> = <NUM><INT> | <NUM>
  <NUM> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 2.2: A sample grammar for parsing an arithmetic expression. This grammar is an adaption of the arithmetic expression grammar from Fuzzingbook [161] presented in the chapter "Fuzzing with Grammars".

Let us explain on Figure 2.2 how we map the different parts of the context-free grammar four tuple to the syntactical elements of the grammar format of the Fuzzingbook [161] we use. First, we represent **nonterminal symbols** V_N by surrounding a capital lettered string with < and > (e.g. <START>). All other strings except the pre-defined control characters =, |, <, > represent **terminal symbols** V_T (e.g. *+* or *sin()*. The start symbol S is always the nonterminal symbol <START>. Now that we have the building blocks of the grammar, we can define the **production rules** P: Each line in Figure 2.2 contains one rule which starts with a nonterminal symbol followed by = and a set of expansions, divided by |. Each expansion again consists of a combination of terminal symbols and nonterminal symbols. For example, the production rule for the nonterminal symbol <INT> has two alternative production rules: <NUM><INT> and <NUM>, separated by |. With this we can describe context-free grammars in a human readable format and keep them usable by algorithms for *generation* as well as *parsing*. While the input generation is interesting (e.g. when building a grammar based parser [60]), we concentrate on the parsing part, thus we do not go into detail how inputs can be generated with grammars.

Parsing Many programs use a grammar to build a parser, which 1. validates the input according to the format defined by the grammar and 2. builds a derivation tree (see below) which gives structural information about the input that can then be used by the program logic for further computations (e.g. to compile the code to machine instructions, or in a semantic phase to validate further restrictions on the input like def-use dependencies). Thus, let us give one example how input parsing with context-free grammars works on a theoretical level. Parsing starts with the start symbol <START>. Now, since we have a given input, the parser needs to decide which option to choose from (if there is more than one). This is done by looking at the next characters to come and try to greedily match one option to the actual input.

Most importantly, each parsing step is independent from any other parsing step—every rule application can be done *without context*, hence the name context-free grammar. The parsing

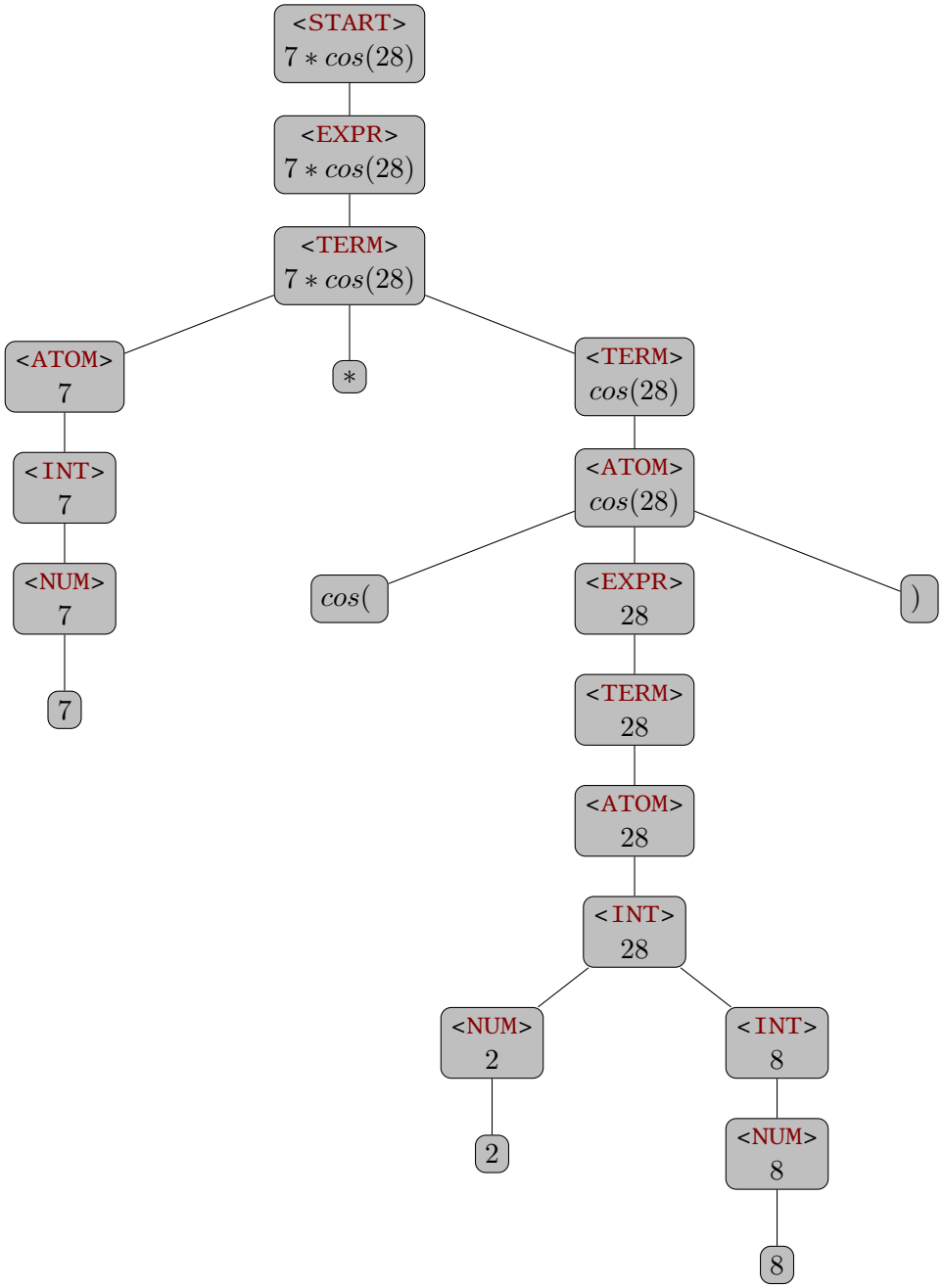


Figure 2.3: A derivation tree of the input $7 * cos(28)$ built based on the grammar from Figure 2.2 applying rules as they would be in a recursive descent parser.

happens with the help of a stack, which is initially filled with the start symbol <START>. At each step, if a nonterminal symbol is at the top of the stack, it is replaced by one of its alternatives, putting the terminal symbols and nonterminal symbols in reverse order on the stack. For example, if <TERM> is replaced by its alternative <ATOM> * <TERM>, then <TERM> is first put on the stack, then *, and then <ATOM>. If a terminal symbol is at the top of the stack, it must match the current position of the input. If so, the terminal symbol is removed from the top of the stack and the position pointer for the input advances. For the sake of simplicity, we assume that we always chose the correct alternative in our example below, otherwise we would need to apply backtracking (choosing another alternative for a nonterminal symbol if the chosen alternative was not correct). Backtracking is typically avoided in real world parsing by designing the grammar such that there is always only one correct alternative to apply, thus we do not discuss this here. If a terminal symbol does not match the current input position and we do not have any production rule to backtrack, parsing fails. The parsing process is successful only if the stack is empty **and** the end of the input is reached.

Figure 2.3 shows how the input $7 * cos(28)$ is parsed *step by step, rule by rule.* In step 1 the nonterminal symbol <START> at the top of the stack is replaced[4] with the nonterminal symbol <EXPR>, the input position pointer does not advance. We can decide to use <EXPR> as there is no other choice for <START>. Now, we replace <EXPR> with one of its alternatives: <TERM>. The next step is more interesting, because we remove <TERM> and add <TERM>, *, and <ATOM> to the stack—in reverse order as defined in the production rule. We replace <ATOM> with <INT>, then <INT> with <NUM>, and then <NUM> with 7.

Now we have our first terminal symbol at the top of the stack, which matches the current position of the input (we are still at the start). Thus, we remove the 7 from the stack and advance the input position. This results in * being the top of the stack, which again matches our input position; thus we remove the top of the stack again and advance the input position. Now, we have <TERM> again as top of the stack, which is replaced with <ATOM>. Next, we remove <ATOM> from the top and add), <EXPR>, and then cos(. These parsing steps are analogously applied until the input position is one position behind the last character (hence the complete input was consumed) and the stack is empty. We successfully parsed the input.

Design Patterns Grammars used for writing parsers tend to avoid the following [3]:

Left Recursion There must not be any derivation from a nonterminal symbol A such that $A \rightarrow^+ A\alpha$ for any string α, i.e. A should not have a production alternative starting with A neither should it be possible to have one or more derivations that end up with A again without adding at least one terminal symbol. For example, this simple left recursive production <A> = <A>α | β would be rewritten to two productions: <A> = β<A′> and <A> = α<A′> | ϵ (with α and β being any string). They ensure that neither <A>

[4] We use replace for the actual operation of removing the top of the stack and adding the nonterminal symbols and terminal symbols from its alternative.

nor <A′> appear first in the production rules, avoiding left recursion. There are algorithms for rewriting more complex left recursive productions [3].

Left Factoring Similar to left recursion, it is useful to make decisions on production alternatives immediately, hence productions rules like <A> = $\alpha\beta_1$ | $\alpha\beta_2$ are discouraged, as one would need to parse α first before the decision can be made if the rule with β_1 or β_2 should be taken (with α, β_1, and β_2 being any string). Hence, this rule is split in two: <A> = α<A′> and <A′> = β_1 | β_2. Now, the decision is deferred to <A′> and it can be immediately decided if β_1 or β_2 should be chosen.

Those rules make it possible to avoid backtracking and large lookaheads, as the parser can always immediately decide which production alternative to choose. The grammar in Figure 2.2 does not contain any left recursion but would still need to be left factored (e.g., the <TERM> rule has three alternatives that start with <ATOM>). For better readability, we keep the grammar as it is.

From the usage of context-free grammars without left recursion and left factoring in real world, we gain some guarantees that can be used in our research. In Section 2.2 we will go into details regarding the parser implementation specific patterns that can be used to infer inputs, in the following we detail the grammar theoretical foundations used in our approach:

1. First and foremost, at each input recognition step there is *exactly one production rule that matches* the pattern/the character to consume.
2. The input is *recognized left to right*, at each step typically one lexeme is consumed (to decide for the next production rule to choose).
3. The tuple *top of the stack and current input lexeme to consume* uniquely defines the next steps taken by the recognizer

Those fundamental design patterns of context-free grammars are also reflected in recursive descent parsers, hence we can look for such patterns in the parser code and in the execution of the parser. In Section 2.2 we will see how those patterns are implemented into recursive descent parsers and Chapter 3 and Chapter 4 will discuss in detail how those design choices are used in our approach.

2.2 Parser Theory

Parsing can be considered as one of the most fundamental parts of computer science. Most programs once went through a parsing step, translating the code written and understood by humans into a machine usable format. In the *Dragon Book* [3] in Chapter 1.3.1 Aho et al. describe the history of compiling starting in the *1950's* with simple *mnemonics* used in assembly languages to make the assembly code easier to read and use for developers. Those mnemonics were then extended with parameterized macro instructions to re-use frequently used instruction sequences (similar to functions today). In the late 1950's the first high level languages like

FORTRAN [139], COBOL [69], and LISP [143] were introduced [3], all of them obviously needed to be parsed before they can be compiled to bytecode. In the following we describe how such a parsing process is done, and what fundamental rules are used during parsing—rules that we can use in our approach.

Context Free Languages Context free language parsers are reasonably efficient while also being complex enough to parse a large variety of real world languages [137, 140]. Still, while the worst case runtime lies in $O(n^3)$ (there are algorithms like Valiant presented that are faster than $O(n^3)$ [148]), unambiguous grammars can be parsed in $O(n^2)$ and most context-free grammars even in $O(n)$ [36].

Context-free parsers are typically extended with more advanced features that make the language they can consume not context free anymore. One of the most prominent examples is XML. While most of the language can be parsed with a context-free parser (opening and closing characters for tags, attributes, values, ...), some very specific parts are in general not context free. A context-free parser cannot decide if an opening and closing tag match in general, as this is a context-sensitive feature. Still, one can use a context-free parser to parse the XML input (initially ignoring if tags are matching)—and then add some simple code, which decides on each closing tag if it actually matches the opening tag. A context-sensitive parser would be able to do the same, but they are less efficient in general [77].

Building Parsers There are different ways to produce a parser from a formal language definition. For all languages from the Chomsky Hierarchy parsers can be written by hand and are written by hand. In many cases though, different levels of parser generators are used, depending on the language complexity. Parsers for context-free languages (and those that are close enough to being context free, see the XML example from above), are often provided directly as a library [114, 145, 146]. If the language is not as common as JSON or XML, a developer can also use a parser generator like ANTLR [136], Bison [44] or YACC [134], often in combination with some pre-defined language description. Those generators take a language specification like the one in Figure 2.2 and produce code that parses the language described in the formal grammar. The benefit of those tools is that one needs not to write the code on its own and in every language it should be used, but rather write the formal description and let the generator compile a parser based on it. Those generators typically also allow to write grammars that are more complex than context free but are not fully context sensitive.

Alternative Parsing Methods In recent years developers came up with other methods that avoid writing a formal grammar but also have the benefit of not writing the full parsing code from scratch. One reason might be that the average programmer needs to parse inputs from time to time but is not very familiar with formal languages.

Combinatorial parsing is a method in which not one parser is written, but many, which are combined to larger parsers that are then used in subsequent recombinations. Frost et al. first used it to create natural language processors [46]. In their paper they describe a system, which

answers a small set of natural language queries about our solar system. Hutton builds upon this technique to design parsers for context free languages. In his paper he translates a typical context-free grammar in BNF into a combinatorial parser [68]. This combinatorial parsing technique brings the theoretical world of parsing closer to developers.

Similar to combinatorial parsing is PEG [43], with Pyparsing [119] as an instance of how a parser for PEG can be built. Here the grammar can be built on Python code level, using Python syntax, while still being close to a context-free grammar format. With this method it is easy for developers to write such a parser manually, possibly following a given grammar, but keeping the language in which the parser is written the same as the language which consumes the parsed input.

Table driven parsers have a very special way of parsing inputs. Instead of using comparisons and control flow, they use *"states to keep track of where we are in a parse"* [3]. Still, this technique makes it difficult to read the code and writing a table driven parser by hand would be hard, hence such parsers are typically generated from an underlying grammar (if at all, for larger projects typically other parsing methods are used [37, 137, 140]).

Recursive descent parsers use control flow and comparisons to decide how each and every character is handled during parsing. They are the recommended parser type in the literature (Aho et al. mention top-down parsing as one of the most common parsing techniques for hand written parsers in Chapter 4 of their *Dragon Book* [3]) and are therefore also taught to programmers around the world. Thus, recursive descent parsers are very often found in mature projects and many parsers are written with this technique by hand [37, 137, 140]. Furthermore, those parsers are not only very common, but also often lack a formal, machine readable definition of the input language.

In this dissertation we focus on recursive descent parsers that are solely written in code and thus typically do not have an underlying machine readable specification. The main reason is that if a machine readable specification exists, it makes more sense to use this specification for any application like input generation and fuzzing as domain knowledge is always beneficial. Still, those parsers typically follow an underlying grammar, even if not explicitly defined, and are written in a top-down fashion. Thus, in the following we explain top-down parsing in more detail and set the ground for our explanation of parser analysis and fuzzing done in this thesis.

2.2.1 Top-Down Parsing

"Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in pre-order [...]. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string." [3]—that is how Aho et al. introduce the chapter on top-down parsing in their book. This section will mostly follow their descriptions. Essentially, the derivation rules are applied based on lookaheads of one or more characters on the input stream, a so called LL(k) parser [3]. Such a parser starts

with the start symbol of the underlying grammar and applies derivation rules on the leftmost nonterminal symbol (the first nonterminal symbol in the parse tree in pre-order) until either no more derivations can be applied that match or all derivations are applied. If no matching derivation is found, top-down parsing can backtrack over previous derivations to the point where a nonterminal symbol can be expanded with another production rule. This backtracking and application of alternative production rules is done until no more applications are possible, then the input is deemed as not parsable.

As we can see, if we would apply top-down parsing naïvely, the parser might have to backtrack many times and try out in worst case an exponential amount of production rules. To avoid this, grammars for parsing with a top down parser are typically written in a way that a small amount of lookahead is already sufficient to decide for the correct derivation rule and backtracking is not needed anymore, making the parse process much faster. In Section 2.1.1 we have shown *left recursion* and *left factoring* transformations that can be used to achieve this feature for most context-free grammars. Therefore, grammars are often in the *LL(1)* format if they are used for parsing—meaning that a lookahead of one lexeme is sufficient to decide which production rule to use, there is no alternative and as such, if the parser gets stuck and cannot apply any production rule at any point, the input cannot be parsed, no backtracking is needed [3].

Recursive Descent Parsing While top-down parsing is a general form of parsing inputs, the focus of this work lies on one of the most commonly used instance of top-down parsing: *recursive descent parsers*. From the *Dragon Book* [3]: a *"recursive-descent parsing program consists of a set of procedures, one for each nonterminal symbol. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string"*. As such, a recursive descent parser implicitly implements the structure of the underlying grammar, making it on the one hand easy to write and modularize, such that it can be adapted with regards to future changes in the original grammar and on the other hand well analyzable. A recursive descent parser written by the textbook follows some coding rules—rules that we can *identify*, *analyze*, and *make use of* while querying the subject under test.

Figure 2.4 shows a generic top-down parsing algorithm for a nonterminal symbol—every nonterminal symbol is typically parsed the same way, including the start symbol. First, in Line 2 a nonterminal symbol is chosen for further parsing. While iterating over all elements of the production rule X_i in Line 3 there are three options: X_i is either an nonterminal symbol, the same terminal symbol as the current input symbol, or none of them. In the nonterminal symbol case (Line 4), the procedure for parsing the nonterminal symbol X_i is called recursively, which must exist as in top-down parsing each nonterminal symbol has its own parsing procedure. In the terminal symbol case (Line 6), the algorithm just advances the input symbol. If none of the other options is correct, the algorithm is in an error state (Line 8). Now, in the case of backtracking we would not immediately error out in Line 8 but rather choose another production rule in Line 2 and iterate again over the production elements. Only if there is no production rule left in A we can actually stop the algorithm with an error (which might result in further backtracking to and in the caller of the method `A()`). We can also use an LL(*1*) grammar and

lookaheads to determine the correct production rule. This reduces the parsing effort by making backtracking unnecessary (there is just one possibly correct production rule to choose in Line 2) while also making sure that if we reach Line 8 there is actually a parsing error, as there is no other way to parse the input.

```
1  void A() {
2    Choose an A-production, A → X1X2 ... Xk;
3    for (i = 1 to k) {
4      if (Xi is a nonterminal symbol )
5        call procedure Xi();
6      else if (Xi equals the current input symbol a)
7        advance the input to the next symbol;
8      else /*an error has occurred*/;
9    }
10 }
```

Figure 2.4: A high level top-down parsing algorithm for parsing a nonterminal symbol as defined in the *Dragon Book* [3].

For building an LL(*1*) top down parser, Aho et al. [3] describe two functions: `FIRST()` and `FOLLOW()`. Both describe sets of terminal symbols—`FIRST()` describes the set of terminal symbols at the beginning of strings that are derived from a given nonterminal symbol, `FOLLOW()` the set of terminal symbols that can appear immediately after a given nonterminal symbol.

In prose, we can describe `FIRST()` as follows (capital letters are single grammar symbols):

1. `FIRST(`X`)` contains X if X is a terminal symbol—and nothing else.
2. `FIRST(`X`)` contains the union of `FIRST()` of all production alternatives—for each alternative $Y_1Y_2 \dots Y_k$ `FIRST()` is evaluated by adding `FIRST(`Y_1`)` and checking if ϵ is in `FIRST(`Y_1`)`, and if so, iteratively `FIRST(`Y_2`)`, `FIRST(`Y_3`)` etc. is added until one does not contain ϵ. If all `FIRST(`Y_1`)` to `FIRST(`Y_k`)` contain ϵ, ϵ is added to `FIRST(`X`)`.
3. Finally, if X produces ϵ, then ϵ is added to `FIRST(`X`)`.

`FOLLOW()` can be described as detailed below (again, capital letters are single grammar symbols, α and β are strings of grammar symbols):

1. `FOLLOW()` of the start symbol is the end of input marker.
2. `FOLLOW(B)` contains `FIRST(`β`)` except ϵ if there is any production rule with a right hand-side $\alpha B \beta$.

3. FOLLOW(B) contains FOLLOW(A) if A either has a production rule with a right hand-side αB or A has a production rule with $\alpha B \beta$ with FIRST(β) containing ϵ.

With FIRST() and FOLLOW() we can define the key properties of an LL(*1*) grammar (following the definition of the *Dragon Book* [3]), which are also important for our approaches that we later explain in this thesis. Say we have a nonterminal symbol A that produces either the string of grammar symbols α or β:

1. FIRST(α) and FIRST(β) are disjoint sets (hence not both α and β can derive the same terminal symbol as first character and only at most one of them can derive ϵ)

2. if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint (and vice versa).

The combination of both rules ensures that a lookahead of one character is sufficient to decide which derivation rule will be applied. Since FIRST(α) and FIRST(β) are disjoint, we can check if the looked ahead lexeme *a* is in either FIRST(α) or FIRST(β) and apply the respective rule. If it is in none we can check if it is in FOLLOW(A) and then apply the rule, which contains ϵ in its FIRST() function, as we need to use an epsilon production and then apply whatever comes after the production of A. If the upcoming lexeme is not in one of the sets, then there is no correct derivation rule and we can stop parsing without backtracking, as we defined the LL(*1*) grammar exactly for this purpose.

With this parsing technique we get a set of guarantees that can be used to analyze a subject under test implementing a recursive descent parser:

1. The parser does not backtrack; a lexeme, once consumed, will not be parsed again.

2. All lexemes of the respective FIRST() and FOLLOW() sets need to be checked before rejecting an input.

3. Each lexeme, depending on the already read characters, introduces a new derivation.

4. The lexemes in FIRST() and FOLLOW() at each state during parsing define the possible characters for that state, they contain only and all valid characters.

5. Each time a nonterminal symbol will be parsed, the parser calls the respective parsing function of the nonterminal symbol.

In Chapter 3 and Chapter 4 we will see how those guarantees can be used to efficiently analyze such a parser for input inference (e.g. by using heuristics that rely on those guarantees for guidance). We will also see that those guarantees do not hold in all circumstances, but most, making the input inference still efficient.

Lexer Typically, complex parsers do not work on the input byte stream directly, they rather have a dedicated module/code location which handles the input strings, converts them to some internal tokens (e.g. enums) and gives them to the parser which then decides if the token it just received is actually valid at the current time in parsing. As written in the *Dragon Book* [3] in Chapter 3.1.1, lexical analyzers are typically used for three different reasons:

1. *Simplicity of design*: handling input structures on the character level is inherently different from the actual parsing step (e.g. handling of whitespaces).
2. *Compiler efficiency*: if split up, the compiler can use very efficient lexing techniques before actually parsing a token.
3. *Compiler portability*: the lexer reads the input directly from the system, meaning that it needs to handle details about the system I/O, which can be abstracted away with the help of a lexer.

While the presence of a lexer does not change the parsing guarantees mentioned above, it inherently changes the implementation style. In Chapter 4 we will see how the lexer influences our approach and what we do to still infer inputs from a parser, even in the presence of a lexer.

2.3 Fuzzing And Input Generation

Context We will later combine our parser analysis with a fuzzing approach, hence we introduce the concept of fuzzing here. The whole story began in 1988 with Barton Miller et al. [109].[5] They started a project to test Unix tools with randomly generated strings. From todays perspective we would call this random blackbox fuzzing: the fuzzer generates inputs without domain or program knowledge and runs the program with those inputs (possibly through a fuzz wrapper—some code or tool that makes it possible to run the subject under test with the generated inputs). Interestingly, even though their approach of generating random strings as inputs is "somewhat naive" [109] (the wording they are using), they were able to crash between 24% and 33% of the 88 tested unix utilities on each of the seven different versions of Unix [109].

Miller and his team did not only crash those programs, they also checked the crash reasons and compiled a list of different bugs and how to avoid them. Interestingly, even over 30 years later, the typical bug sources still exist, e.g. *memory bound checking*, *system call return checks*, and *pointer value sanity checks* [109, 110]. Languages like Java and Python certainly handle some of those bug sources automatically (one cannot access arrays out of bounds without raising an exception in those languages; the execution will not end up in an undefined state in such cases [113, 120]). In C though (so in domains where execution speed and hardware proximity matters), such guarantees are not given and likely will never be given. Creating an array like "`int a[100];`" and then accessing the allocated array out of bounds("`a[999] = 5;`") will not be prevented by any compile time or runtime measures and results in undefined behavior,

[5] The paper itself is published in the ACM library since 1990, the work on this though already began 1988 [11].

in best case the program errors out with a *Segmentation Fault*, in worst case the program keeps on running non-deterministically [72].

Fuzzing has evolved since then but this basic idea did not change much in the last 30 years: the English Wikipedia page on fuzzing [152] describes it as a mainly automatic process to generate random or semi-random inputs to trigger unwanted program behavior: *"An effective fuzzer generates semi-valid inputs that are 'valid enough' in that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program and are 'invalid enough' to expose corner cases that have not been properly dealt with"* [152]. On 12. January 2024, the ACM website lists 760 citations of the initial fuzzer paper by Miller et al. [12], which just reflects a part of the overall fuzzing research done in this field. A recent study by Miller et al. has shown that even though fuzzing is widely adopted in many fields, the most basic random string fuzzer still finds bugs in the unix utilities [110]. *"100k+ CPU cores, mostly n1-standard-1 vms"* [1]: that is the amount of resources Google uses according to Abhishek Arya at *21. August 2021* for running ClusterFuzz [138] , showing that even in large tech companies fuzzing is very popular.

While the blackbox approach of fuzzing originally proposed by Miller et al. already finds a significant amount of bugs in software, *fuzzing was not and is not exhaustively researched.* Böhme et al. compiled a list of different open questions in fuzzing in 2021 [18], including questions like how to fuzz more types of software, find more bug types (part of the *oracle* problem), and finding more difficult bugs—bugs that are guarded by complex conditions.

Input Generation Techniques The ideas varied a lot throughout the years and the lines between fuzzing and other input generation techniques became blurry. Some decided to include more information about the subject under test, e.g. *coverage information* like in AFL [160], but also more in depth information provided by *static and dynamic analysis* [41, 123]. Other techniques *combine different existing fuzzers*, leveraging their strengths and weaknesses [28]. A common way of *incorporating domain and target specific knowledge* is by providing a model of the input format, which is then used for input generation by the fuzzer. This model can be incorporated in the fuzzer [154] and even extended with known bugs, which are varied and brought into a new context [63]. Some techniques take the *model in a machine readable form*, hence they are more generic—any target which input format can be represented by the respective model can be fuzzed. For instance, one could use a grammar [60] as input model. There are techniques that *infer such models*, often from a set of inputs [49, 54, 65, 132, 133], combining the idea of simple push-button techniques like blackbox and greybox fuzzing with more advanced techniques that use underlying models. Other approaches to handle complex conditions in the subject under test are *symbolic execution and concolic execution*, either directly applied [23] or to be combined with fuzzing [135]. Though, those approaches typically suffer from the *path-explosion problem* [24, 135].

Simple blackbox input generation is very program agnostic—the fuzzer generates inputs randomly, possibly without any feedback and the program is run on those inputs. The only

needed information is how to start the subject under test and how to provide an input to it. Obviously, once the fuzzer uses instrumentation it is bound to programs that can be instrumented with the instrumentation framework (at least if the maximal fuzzing performance should be achieved). This target dependency is not only restricted to simple instrumentation, some fuzzers are bound to domains, e.g. the approach by Godefroid et al. [49] is designed to use a neural network to infer an input grammar from the subject under test using given and newly generated sample inputs, hence it should only be used on such subjects. Approaches like CSMITH [154] are specifically designed to test one specific kind of subject, e.g. C compilers, hence their target subjects are very restricted.

Complex Input Formats and Fuzzing In the domain of complex input formats, push-button techniques tend to struggle with generating valid inputs. The main reason is their insufficient capabilities to correctly generate valid tokens in a correct order. Think about this: we have a parser for arithmetic expressions and want to generate the input $sin(2) + cos(8)$ with a fuzzer that has no domain knowledge and possibly only coverage feedback from the program. In order to trigger the sin branch of a parser it would need to blindly generate the letters *s*, *i*, and *n*—only then the keyword sin is generated and can be detected by the parser. The likelihood for this to happen using the basic ASCII characters (128 characters) is 128^3. The same holds again for cos[6]. Hence, in the presence of a parser a fuzzer typically relies on some form of a manually crafted model or needs a high amount of resources to solve the complex comparisons in a parser to produce syntactically valid inputs.

Though, the problem is: *where does the model knowledge come from*? If the subject accepts very generic inputs (e.g. any JSON or XML input), a generic fuzzer or domain description can be used. In many cases though, the consumed input of the subject under test is too specific and the domain specification needs to be written by the maintainers of the subject. This may introduce bias into the description (like *missing syntactical features*, *too restricted or lenient input types*, or *missing terminal symbols*). Also, one major roadblock for implementing fuzzing as a part of the CI/CD pipeline is the effort to set up a fuzzer. Even though simple greybox techniques require no more than using a specific compiler that performs the fuzzer instrumentation in the subject under test, practitioners (users from the industry) still see usability as one of the most important points in todays fuzzing research [18]. Hence, we can assume that practitioners will tend to not maintain a model specification in parallel to their code base, not even speaking of writing an initial one. Thus, for testing parsers we would like to have a low influence of and workload for the developers when setting up an input generator.

Miscellaneous There are also other domains that need to be tested like *graphical user interfaces* [21, 75] or *protocols* like PROTOBUF [51] (which can be tested with the LIBPROTOBUF-MUTATOR [52] that mutates protobuffers that can be used for input generation). In all generality, every input generator could generate a string and a wrapper "around" the subject under test

[6] This example assumes the typical way of matching keywords in parsers by using direct string comparisons—in this case comparing the constants *sin* and *cos* against input characters. Such a comparison would not give any coverage feedback up until the full keyword is generated and the comparison outcome is altered.

converts it to a valid input like a sequence of clicks or API calls. Still, in those cases it makes sense to have some *"domain specific"* input generators for the respective domain. But not only this, input generators can also make use of new techniques: for example, some researchers work on using large language models to improve input generators [83]. It is out of scope for this dissertation to list all the work done in input generation throughout the last 30 years.

Input generators can be used in nearly any domain, produce any kind of input, and can be combined with many different other research areas to improve the overall performance. The target domain of an input generator and its underlying design (random inputs, symbolic execution, ...) are completely unrelated. In general though they are at least adapted to their execution environment, hence the inputs they generate are valid in the sense that they are not rejected because they are different from what the execution environment requires. For example, a GUI input generator [21, 75] will produce clicks and other valid inputs for GUIs—it might fail though to log into an application, because then the input generator would need target knowledge instead of domain knowledge.

Summary We have seen: the basic idea of fuzzing is simple, the most basic input generator one can think of could be written in one line of Python code (ignoring the needed imports):

```
while True: print(str(random.randbytes(random.randint(0,1000))))
```

But once we start to think about different variations, extensions, and improvements to this basic idea, the possibilities seem to be infinite. With the approach that we will discuss in this thesis we try to *keep the simplicity of push button input generators*, while *producing complex, syntactically valid inputs* that cannot be generated with a *state-of-the-art* push button approach.

2.4 Dynamic Tainting

One of the main information drivers in our tool is dynamic tainting, hence we will explain its basic concepts. There exist several dynamic tainting tools [16, 31, 151], all of them have their advantages and disadvantages. The tainting engine Pirate [151] in PANDA [117] for instance is based on QEMU [122] and as such runs on a virtual machine. It translates the executed QEMU code to LLVM IR for further analysis during the offline taint tracking phase. This tool is able to analyze binaries by leveraging virtual machines and translating the code to an intermediate representation; other tools run on other abstraction levels, e.g. directly on the binary [31].

In general, dynamic tainting works as follows: first, one needs to define what should be tainted. Typically, one would taint bytes coming from a certain source, e.g. bytes from a *memory location*, a *function return value*, or bytes from an *input source*. Those bytes are dynamically followed through the program execution[7], for each instruction of the underlying language. In this section we restrict ourselves to data-flows only, i.e. taints are only propagated if there is

[7] That's why it is called dynamic tainting; in contrast to static tainting which would calculate possible byte propagations without running the code.

a direct information flow between two values. Or in other words: a taint is not propagated over indirect information flows, which can be summarized as information flows over control flow. Taking control flow taints into account usually results in almost all registers and memory locations tainted, especially if the initial taints stem from the input (almost every decision in the program is based on the input as the main purpose of a program is processing input information). Therefore, typical tainting algorithms concentrate on data-flows as they are much more targeted and useful for most use cases.

Example: Direct and Indirect Information Flows

In the following code snippet we have an example of a direct information flow from variables *a* and *b* to variable *c*:

```
// a and b contain taints
int c = a + b;
```

The value in *c* stems directly from the sum of *a* and *b*—there is a direct data flow to *c*. Such flows, where information is stored from a tainted value *X* to a memory or register *Y*, are the direct data flows we are focussing on.

In contrast to that, there could also be indirect information flows:

```
// a and b contain taints
int c = 0;
if (a == b) {
    c = 1;
}
```

The value in *c* after executing all lines is dependent on the tainted values in *a* and *b*. Still, the values from *a* and *b* do not flow via a store instruction into *c* but only influence *c* via the code structure. Those indirect information flows over control flow instead of data flow produce a much noisier taint result.

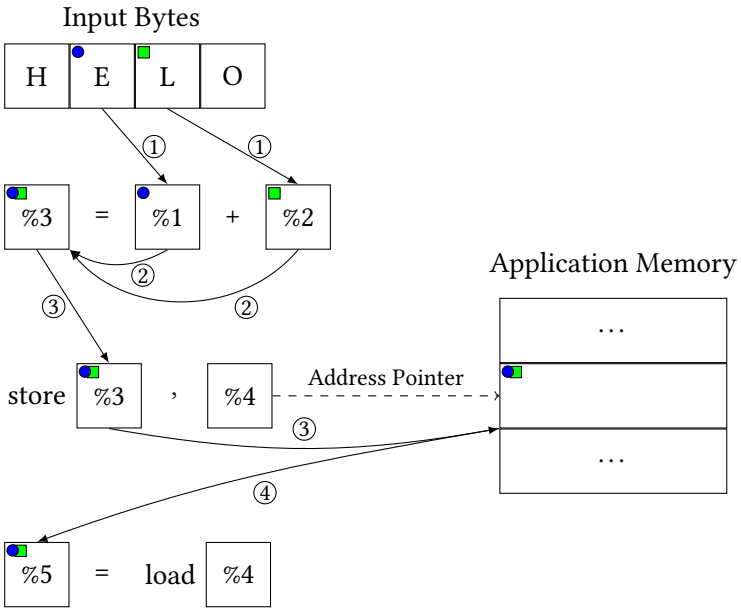
End Example: Direct and Indirect Information Flows

Since we will later work on the LLVM bitcode level, we explain dynamic tainting on this level of abstraction. Figure 2.5 shows three LLVM bitcode instructions, an addition in Line 3, a store instruction in Line 4, storing the result of the addition at the memory location denoted by the second value, and a load instruction in Line 5, loading the just stored value from memory back into an LLVM bitcode register. In the following example we illustrate how dynamic taint propagation works for an addition, a store instruction, and a subsequent load instruction. Even though the example is short, it already contains different building blocks of the underlying system we are analyzing: *a binary instruction*, *writing to memory*, and *loading from memory*. Most of the instructions we are analyzing are similar to one of those three, hence the taint propagation works analogously for them.

```
1 ; Reading two 8 bit integers from the input into %1 and %2
2 ;...
3 %3 = add i8 %1, %2
4 store i8 %3, i8* %4
5 %5 = load i8, i8* %4
6 ;...
```

Figure 2.5: A sample LLVM bitcode excerpt. The sum of registers *%1* and *%2* is calculated and stored to register *%3* (Line 3). Then the content of register *%3* is stored at memory location *%4* (Line 4) which is again loaded into register *%5* in Line 5.

Example: Dynamic Taint Propagation



Using the code from Figure 2.5, let us assume that in Line 3 the registers ***%1* and *%2* contain bytes from the input**, we can be even more specific and assume that *%1* contains the byte from index one of the input and *%2* the byte from index two. This is reflected at the top of the example picture above, input byte one is marked with a blue circle, input byte two with a green square. The arrows labelled with 1 denote that those two input values flow into registers *%1* and *%2*.

After the addition in Line 3 the register *%3* has the taints of *%1* and *%2* attached to it, hence the taints from input byte one and two—the taint flow is visualized with the arrows marked with the number 2. Register *%3* is accordingly marked with the circle and the square indicating the taints 1 and 2 are attached to register *%3* at that point.

In Line 4 this information is stored at the memory address referenced by register *%4*. Hence the taint engine taints this memory location with the taints of byte 1 and 2 from the input. Concretely, the taints flow from *%3* into the according memory location as shown with the arrows number 3.

Finally, in Line 5 the value is again loaded from the memory location addressed by register *%4* and stored in register *%5*, hence register *%5* gets tainted with the taints of byte 1 and 2 from the input, detailed with arrow 4.

To sum up: after all three lines were executed, the information which originally stemmed from input bytes 1 and 2 now lies in register *%5*, each arrow number shows one step of the execution (label *1* is the storing of the input bytes in *%3*, *2* is the addition, *3* is the execution of the store instruction and *4* is the load instruction). The picture also shows which registers and memory locations contain which taints after the execution of the code snippet. A taint can only be deleted by overwriting the respective memory location or leaving the scope in which the register is valid, i.e. it can only be deleted if the respective storage location does not contain any trace of the the tainted value anymore.

End Example: Dynamic Taint Propagation

3 | Extracting Lexical Elements

In Section 1.1 we outlined the objectives for our thesis and in Section 1.2 we explained the different contributions to those objectives. **This chapter contributes to solving our first objective: generating inputs based on complex comparisons and magic values**. In the following sections we lay the foundations of our approach: *how we can generate a set of syntactically valid and diverse inputs using nothing more than a subject under test which starts its input consumption with a recursive descent parser.* We aim to generate those inputs *out-of-thin-air* using dynamic tainting and heuristics adapted to the domain of recursive descent parsers. Our goal is the generation of an input set that contains only syntactically valid inputs that fully cover the parser code and thus representing the set of (syntactic) features of the input language.

Scope The class of parsers we are focussing on (recursive descent parsers) are typically built around terminal symbols and specifically encode them as constants that are used while parsing inputs. As we can see in Figure 3.1, the general structure and terminal symbols of our original grammar as seen in Figure 3.2 are reflected in the functions and constants of the code. We will make use of this structure to infer, directly from the program code and without any prior knowledge, which inputs are valid and generate them. In contrast to grammar learning approaches like MIMID [54] or the approaches by Sochor et al. [132, 133] we will not infer an abstract representation of the input format but rather analyze the code structure dynamically to apply program specific mutation operators on already generated seed inputs. With this method we can iteratively generate valid inputs that cover the different branches of the program's parser and hence the syntactical features the program encodes. In this chapter we mostly ignore the fact that some recursive descent parsers use a lexer before parsing the input—only the heuristics for ranking the possible next inputs to find the next most promising input try to reduce the impact of this simplification of parsers. In Chapter 4 we extend our technique to also specifically analyze recursive descent parsers with lexers.

Approach Overview *In short, our proposed technique works as follows: we use dynamic tainting to analyze the subject under test while running on different inputs to find valid prefixes and invalid suffixes in inputs, replace those invalid suffixes with valid ones, and thus iteratively generate syntactically valid and diverse inputs.* The following abstract algorithm shows a very brief overview on our approach:

```
1 def inputInference(subject):
2   inst_sut = instrument(subject)
3   input_queue = [random.nextChar()]
4   while True:
5     inp = queue.pop()
```

```
    trace = inst_sut.run(inp)
    taints = taintEngine.analyze(trace)
    new_inputs = extract_inputs(taints, inp)
    input_queue.sortAndAdd(new_inputs)
```

```
int term() {
    skip_whitespace();
    if (atom()) {
        skip_whitespace();
        if (input[pos] == '*' || input[pos] == '/') {
            pos++;
            return term();
        }
        return 1;
    }
    return 0;
}

int expr() {
    skip_whitespace();
    if (term()) {
        skip_whitespace();
        if (input[pos] == '+' || input[pos] == '-') {
            pos++;
            return expr();
        } else {
            return 1;
        }
    }
    return 0;
}
```

Figure 3.1: An excerpt from the arithmetic expression parser parsing the *term* and *expr* nonterminal symbols from the grammar from Figure 3.2. The full parser can be found in Appendix A-I.

Concretely, we are starting with an input parsing program as input to our approach. We instrument and compile the subject (such that it outputs an execution trace for later analysis) and fill our queue of possible inputs with one input: *a random character*. Then the overall generation loop starts, taking one input from the queue, running the instrumented program with it, which in turn produces an execution trace. We use dynamic tainting to analyze the trace and report interesting code locations, like comparisons, including their taints to the input generator. Those comparison taints are used to find locations (i.e. input characters) in the original input that can be substituted, e.g. with the values they were compared to. Recursive descent parsers typically have the terminal symbols directly encoded as constants in the comparisons in the

```
<START> = <EXPR>
 <EXPR> = <TERM> + <EXPR> | <TERM> - <EXPR> | <TERM>
 <TERM> = <ATOM> * <TERM> | <ATOM> / <TERM> | <ATOM>
 <ATOM> = ( <EXPR> ) | sin( <EXPR> ) | cos( <EXPR> ) | <INT>
  <INT> = <NUM><INT> | <NUM>
  <NUM> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 3.2: A sample grammar for parsing an arithmetic expression. This grammar is an adaption of the arithmetic expression grammar from the Fuzzingbook [161] presented in the chapter "Fuzzing with Grammars". The original grammar is presented in Appendix A-III.

code. Because a parser typically consumes an input from left to right and stops when finding the first invalid lexeme, we substitute the last found substitution location—the location at which the parser stopped and thus the first invalid portion of the input. Using domain-specific heuristics, we add those new input candidates into our input queue and query the next, most promising input and run the subject under test again.

3.1 Approach

If one thinks about grammars, lexical elements cannot be left out of the equation. Being the building blocks of each grammar, they are the fundamental parts of each input. In a grammar a lexical element is represented by a terminal symbol, hence the set of terminal symbols V_T in the grammar also build the set of lexical elements. As explained in Section 2.2, recursive descent parsers heavily rely on comparisons which directly encode the terminal symbols as constants in the code. In the following we will show how to generate valid inputs with dynamic parser analysis.

How Parsers Are Built In Figure 3.1 we can see the direct mapping of nonterminal symbols as functions and terminal symbols in control flow and comparisons. The left hand-sides of the production rules are reflected in the methods, the *expr* and *term* production rules (as presented in Figure 3.2) are translated to their own functions in the C code. The return value of the function indicates if the nonterminal symbol was successfully parsed. The right hand-sides are implemented as function bodies of the methods of the respective nonterminal symbols, comparing constants (the terminal symbols) or calling the respective functions of the nonterminal symbols.

For parsing, we typically need to skip whitespaces before consuming another relevant terminal symbol, hence the function calls to *skip_whitespace()* at different positions in the code, e.g. in Line 2. More interesting though is Line 16 which directly follows the flow of the grammar, parsing the nonterminal symbol *term* as a function call to the function *term()*. All nonterminal symbols are parsed by calling their respective functions, letting them consume the terminal symbols in the derivation tree of the nonterminal symbol greedily and checking the return code.

On the other hand, terminal symbols are much easier to parse. In their case the parser only needs to check if the character(s) at the current position match the terminal symbol, e.g. in Line 18 it tries to parse the + and − terminal symbols. If successful, the parser cursor is advanced equal to the number of characters consumed, in this case one character (Line 19). After reading a + or −, an *expr* must follow, hence the parser tries to successfully parse the *expr* nonterminal symbol in Line 20 and returns the respective value.

A production rule is a combination of optional lists of terminal symbols and nonterminal symbols that can be applied. In the *expr* parsing function we can see the optionality reflected in the control flow. If in Line 18 neither a + or − was consumed, the parser knows that *term* was already consumed and returns true immediately (Line 22) as the operator and a following *expr* is optional. Similar, if the parsing of a *term* fails right at the beginning of the *expr* parsing step, the parser can immediately stop parsing and return false (Line 25).

As described above, the lexical elements, i.e. the terminal symbols, are used in the code to consume characters while parsing. Concretely, they are used to guide the parser character by character, lexeme by lexeme through the parsing steps. Hereby, each lexeme is greedily consumed while parsing, i.e. on matching a lexeme the upcoming next parsing step is defined.

Resulting Guarantees This method of parsing gives us two general concepts which we can use for generating valid inputs with our technique: a) terminal symbols are reflected as constants in the code and each terminal symbol needs to have a corresponding constant in the respective comparisons and b) terminal symbols are only compared at the position at which they are valid and the parser checks all valid options before stopping parsing.[1] This means, if we look at comparisons during program execution we will see all valid terminal symbols at some point (if we cover all features of the grammar) and if we have a valid prefix but an invalid suffix, the parser will compare the invalid suffix (at least partially and from left to right) with all valid options.

Parser Analysis By relying on those guarantees, we can build an algorithm which is able to create valid inputs iteratively by observing several program executions with carefully selected inputs. The basic idea is the following[2]:

```
1 def input_generation_and_execution():
2   inp = random(printable_characters)
3   frontier = PriorityQueue()
4   comparisons, is_valid = run_and_trace(inp)
5   while True:
6     if is_valid:
7       report(inp)
8     for comp in comparisons:
```

[1] We will later see that those two assumptions do not hold in all generality in the real world, but to some extent. For the moment, let us assume they always hold.

[2] There are more details to the algorithm which are omitted here for readability.

```
9         frontier.add(inp, comp)
10      inp = mutate(frontier.pop())
11      comparisons, is_valid = run_and_trace(inp)
12      if is_valid:
13        continue
14      inp = inp + random(printable_characters)
15      comparisons, is_valid = run_and_trace(inp)
```

Figure 3.3 also visually represents this generation loop. Our approach starts with nothing but a random input character, given to the program (Line 2). We could also start with an empty input, but typically a parser would not perform any calculations on such a value, it would get rejected or accepted right away and we will not gain any information. The subject under test's parser compares our random input with all possible alternatives it knows, it checks all possible valid starting bytes for a valid input. Our tool can observe these comparisons by looking at the taint trace produced by our dynamic tainting engine when running the program on the input (Line 4).

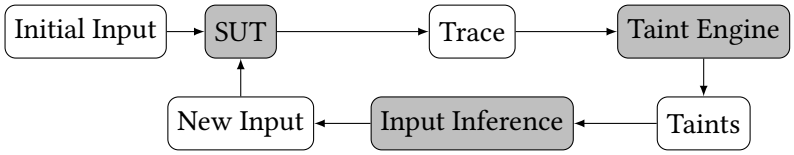


Figure 3.3: The input generation loop of our inference algorithm.

We check if the just executed input was syntactically valid and report-worthy (e.g. *the program has exit code zero and the input covers code that was not covered by any other valid input beforehand*), and if so we report the input as valid (Line 7). The comparisons are put into a priority queue based on a generic heuristic classifying recursive descent parser comparisons and their inputs (Line 9). The input that caused the comparison on the subject is stored alongside.

Now, the most promising comparison is retrieved from the frontier and a comparison specific mutation is applied on the input stored with the comparison (Line 10). Typically, the input characters that were used in the comparison are replaced with the constant value they were compared to; the characters that may appear in the input after the characters used for replacement are discarded. This new input is first checked for validity by running the program on the input (Lines 11 and 12) and if so is used to start the next iteration of the loop (Line 13)—i.e. it is reported and a new input is taken from the queue.[3] If the program did not finish with exit code 0 or the input did not cover new code, we apply a random addition to the input as we assume the input to be at least a valid prefix and we want to force the parser to compare the

[3]Technically, if this input without random extension did not let the program exit with exit code 0, the program would not be traced to increase the overall execution speed. The tracing is only needed for the randomly extended input in the second step.

next character (Line 14). This new input is again given to the subject under test and the loop restarts (Line 15).

Sample Execution Figure 3.4 gives an example for a happy path, showing how our approach would generate an input on the parser presented in Figure 3.1 (or respectively the full parser as presented in Appendix A-I). We start with the random input *'@'*, give it to the subject under test and let the instrumented program process the input. The resulting trace contains the comparisons done on the input character, resulting in the possible substitutions *'('*, *'sin('*, *'cos('*, *and the numbers zero to nine*. One value is selected, in this case the opening parenthesis, and the *'@'* character is replaced with this value. Now it is checked if the new input is valid (it is not) and a random character is added to the newly generated input, resulting in *'(&'*. The next loop iteration is performed with the generated input, resulting in an even longer valid prefix: *'(2'*, which is extended with an equals character. After another loop iteration, we get our final result: *'(2)'*, a valid input for our expression parser.

Summary We start with a random input, run it on the subject under test to gain initial information about the parser. Then we sort possible mutations inferred by parser comparisons into a priority queue backed by a heuristic, which is designed to increase code and input feature coverage with generated inputs. With this method we are able to query the parser iteratively, gaining more and more information about it and ultimately generate a variety of syntactically diverse inputs covering the features of the parser.

3.2 Parser Analysis Implementation

In Section 3.1 we outline the foundation of approach, showing the basic idea of how syntactically valid and diverse inputs can be inferred from a parser. While this general idea is certainly sufficient to produce inputs based on parser analysis, the technical details (of our implementation but also of actual parser code) are also important to make the approach applicable in practice. We will first detail how and which information is extracted from the program, then we explain the technical details to make the approach work on actual program code.

3.2.1 Dynamic Tainting For Parser Analysis

Most of the dynamic tainting engine which is used as an information provider for this approach was originally developed for my master thesis [101] and further explained in the publication *Dynamic Tainting for Automatic Test Case Generation* [100]. Thus, the conceptualization and implementation of the dynamic tainting engine is not part of the contributions for this thesis. Still, we explain the concepts here again, as they are central for the approach. Both of our publications [102, 105] use the dynamic tainting engine, hence in this section we use the latest version of the implementation as baseline for our explanations (i.e. the implementation used for *Learning Input Tokens for Effective Fuzzing* [102, 103]), but leave out token specific extensions that are discussed in Chapter 4.

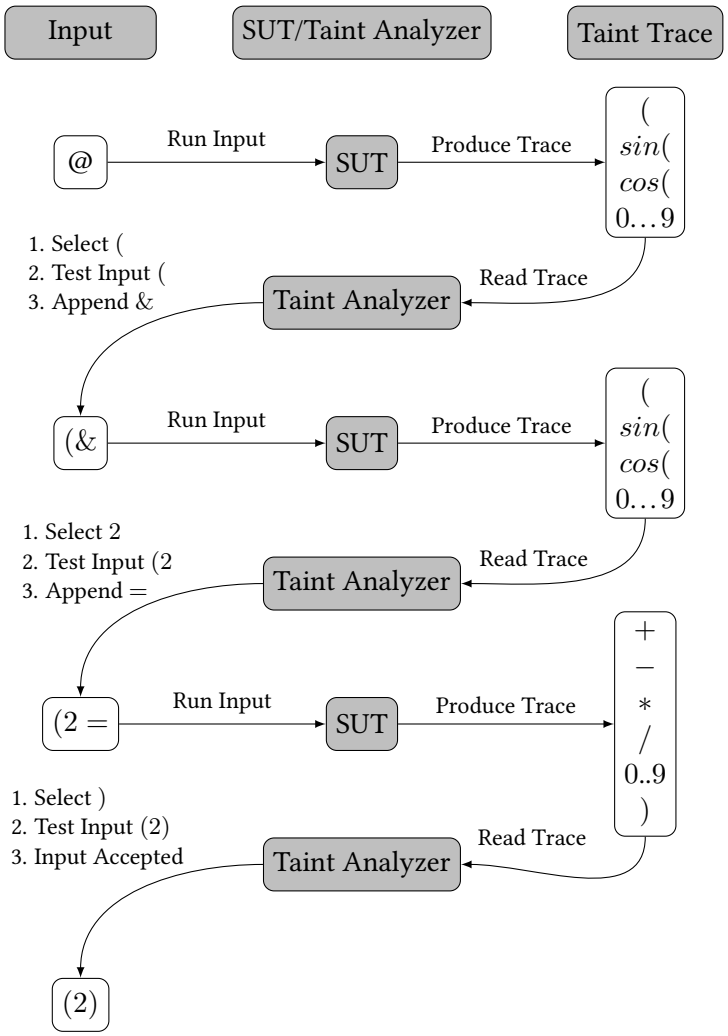


Figure 3.4: A sample input generation with our inference algorithm (this shows an idealized run to avoid details that would make the example more complex). The presented taint traces (nodes on the right) are figuratively shown as the comparisons on the last character of each run as those are the ones used by our taint analyzer algorithm. The actual taint trace would contain all executed instructions which are then analyzed by our taint analyzer to get the comparisons on all input characters. For better readability we combined the tainting engine and input inference into one node—the taint analyzer.

In Section 3.1 we have already seen that lexical elements are encoded as constants in the code and used in comparisons. Still, we cannot use all comparisons from the code but need to filter out those that are actually involved in parsing. Thus, this section explains how we extract the respective information from the subject under test and make it usable for our input generation step.

Overview Figure 3.5 details our implementation of the dynamic tainting engine: we start with the source code of the program under test, *instrument* it and extract *static information/metadata*, *compile* the instrumented program to a runnable binary, *run* it on one or more inputs, store the *execution traces*, and finally analyze the traces together with the extracted static information to produce information about the *taint propagation*. The program version without instrumentation is later used in our input inference loop for executions that do not need any instrumentation. We decided to use LLVM bitcode [88] as the level of abstraction and write the tainting engine on our own—this gives us a better control over the system. In the following we detail each step and justify how the different parts are used for inferring inputs. We will not include every implementation detail, as this would be out of scope for this thesis, but we explain the main parts needed for input inference.

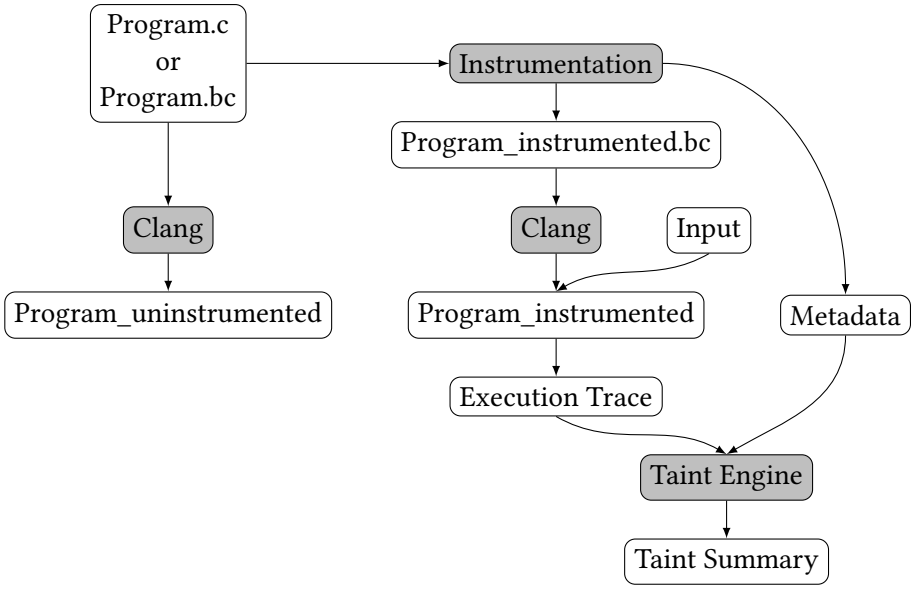


Figure 3.5: Instrumentation and taint workflow. Shaded elements indicate tools, unshaded elements indicate (input) files and products of the tools.

Static Metadata Besides the dynamic information extracted from the subject under test while running, we also extract metadata—*information about functions and their arguments, about global variables, union, and struct types.* In the following we present metadata elements extracted from the program MJS and prettified for better readability [26, 103] to show the relevant information we extract:

Structure Information

```
{
  "sn": "tok",
  "s": 128,
  "el": [{"e": "tok"}, {"e": "len"}, {"e": "ptr"}]
}
```

The key "sn" on the one hand defines that this JSON object describes a structure type, on the other hand its value is the name of the structure type. The "s" value describes the overall size of the structure in bytes, i.e. the sum of the sizes of all structure elements.[4] The "el" key references a list of JSON objects which contain the name of the different structure elements extracted from debug information. The information from the "el" fields are not actively used for the presented version of our approach and are just an extension to the tainting engine.

Union Information

```
{"un": "anon.0", "s": 64}
```

For unions we have a similar setup, but here we only track the name (indicated by the "un" value) as well as the size (the "s" value). Also, the size for unions is the size of the largest element, not the sum of sizes.

Function Information

```
{
  "f": "mbuf_remove",
  "ar": [
    {"a": "mb", "t": "mbuf*"},
    {"a": "n", "t": "i64"}
    ]
}
```

For function calls we first need to know the name of the function (the "f" value). Now, for function calls during execution the tainting engine needs to create a new tainting scope and propagate the taints from the caller registers to the callee registers. Therefore, we store information about the arguments in an array ("ar"). Each element contains the name of the argument register at the respective position ("a") as well as the type "t"[5]. The register name and position is used to determine which named register in the new tainting scope will contain the respective taint from the argument of the caller.

Global Variable Information

```
{"gv": ".str.71", "t": "[3 x i8]", "v": "-l"}
```

[4] The structure name and the size information is partially calculated from the type definition in the LLVM bitcode, partially extracted from debug information.

[5] Similar to structure and union information as described above, the type is not actively used.

A global variable has a name (“gv”), a type (“t”; in this case an array with three eight bit elements) and an initial value “v” for global variables which names start with *.str*. To the best of our knowledge, those variables contain string constants. This information can be used in our tainting engine for comparisons, i.e. a call to the C function *strcmp()* might reference a global variable. As our approach heavily relies on comparisons against constant values, it is crucial to know compile time constants. With this information we can taint the string constant values with a specific taint, tracking the information coming from those values during runtime. In this example, the logged value is `-1`.

Instrumentation Before we can analyze the program execution, we first need to make the program dynamically analyzable. Therefore, we instrument the program under test using an LLVM optimization pass [92] working on LLVM bitcode. Since we are implementing a proof-of-concept implementation, we restrict ourselves to programs written in C, this makes the implementation of the tainting engine easier. At first one might think LLVM bitcode is completely language agnostic (or at least mainly language agnostic), as this is the basic idea of the framework. While this is true for running the instrumentation in general, it is hard to instrument and follow taints through the standard library of the original programming language.

Instrumenting the standard library is very costly in terms of execution time when running the linked subject under test, hence we decided to write function summaries for the most important library calls, i.e. calls that handle strings and characters. For example, we special handle *strcmp()* by reporting the two compared strings instead of just reporting the pointers; also we handle calls to conversion and comparison functions like *strtod()* or *isdigit()* by reporting mock-comparisons to the input generation, which can later be used to replace the respective characters with pre-defined values that are valid for the respective function. For the function *strchr()* it is checked if the search character is tainted and a comparison is reported if so. We also handle the vice versa case in which, if the search character is not tainted, we check if the searched string is tainted—in fact we approximate and check if the first character of the string is tainted and does not stem from a string constant. If so we report a special comparison named *strsearch* between the searched (and tainted) string up until the search was successful and the character that was used for searching, i.e. we simulate which characters the call to *strchr()* would actually use for comparison. Implementation details about all wrapped functions and how they are handled can be found in the replication package of lFuzzer (also including the implementation of pFuzzer) [103].

This wrapping of standard library functions comes with benefits and drawbacks: on the one hand we are no longer language agnostic, we restrict ourselves to C, on the other hand, we can decide for each function how our system should behave. For example, instead of reporting the internals of the function *strcmp()*, we can just report a string comparison to the input generator.

Used Passes/CLI Options To reduce the instrumentation complexity, we use the following LLVM passes and command line options:

-g Includes debug information about the LLVM bitcode file [87] which is analyzed and instrumented by us.

-S For the LLVM optimizer, the documentation says: *"Write output in LLVM intermediate language (instead of bitcode)[…]"* [91]. For Clang is says: *"Run the previous stages as well as LLVM generation and optimization stages and target-specific code generation, producing an assembly file[…]"* [87]. This makes it easier for us to apply our tools as we produce and use intermediate compilation results, which we need to be in the LLVM intermediate language.

-x c *"Treat subsequent input files as having type language[…]"* [87], whereas the language is C in our case, because we only target C code with our prototype implementation.

-O3 *"Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster)[…]"* [87]. The optimization level O2 is described as follows: *"Moderate level of optimization which enables most optimizations[…]"* [87]. This optimization level is used for the final compilation including the instrumented code, hence the program likely executes faster as with a lower optimization level.

-emit-llvm *"Generate output files in LLVM formats, suitable for link time optimization. When used with -S this generates LLVM intermediate language assembly files…"* [87]. Makes it possible, and because of the combination with the "-S" option also easier, for us to apply our tools on the intermediate compilation products (especially the metadata extraction which analyzes the human readable LLVM intermediate language).

-D_FORTIFY_SOURCE=0 *"Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions […]. For some functions, argument consistency is checked; […]"* [85]. Setting it to 0 should deactivate this. The concrete implementation in Clang might be different from the implementation in GCC [130], but as we deactivate the feature it is not important how it is concretely implemented.[6]

-reg2mem Puts registers in memory which removes `phi` nodes from the LLVM bitcode: *"This file demotes all registers to memory references[…]"* [89]. *"At runtime, the 'phi' instruction logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block"* [88]. Due to technical reasons, the phi node would need a special handling in the tainting engine. With this pass we avoid adding this complexity to our prototype implementation.

-instnamer *"This is a little utility pass that gives instructions names[…]"* [89], which also means that every assigned register gets a name. This name can then be used in the tainting engine to uniquely reference the different LLVM bitcode registers.

[6] We do not have citations for the concrete version of clang/gcc used in the evaluation, but the generic citations for `-D_FORTIFY_SOURCE` should be sufficient to explain the command line argument.

-strip-debug *"This option causes opt to strip debug information from the module before applying other optimizations[...]"* [91]. It is used to avoid getting errors if our instrumentation breaks the debug information.

-disable-verify Disables the LLVM bitcode verifier [90]. We expect the final transformation to be correct, though some intermediate results might not pass the verifier (e.g. broken debug information, which is stripped anyways in the end).

-fno-inline Deactivates some code inlining during compilation [45]. Even though the command line flag is only documented for GCC, CLANG has some efforts to keep the command line compatible with GCC in this regard [86]. This makes it easier to follow the trace later as we want to have the original code structure as defined by the developer.[7]

Tracing One Instruction To track how data flows from one instruction to another we need to collect information about the executed instructions. This means, we first need to instrument every instruction in the source code such that, once it is executed, the runtime information we want to trace is actually logged. A snippet of such an instrumented LLVM bitcode instruction would look like the following for a return statement (the example is extracted from an instrumentation from the replication package [103]; prettified and shortened):

```
1  call void @tracerllvm_instructionHeader(
2    i64 1,
3    i8* getelementptr inbounds ([19 x i8], [19 x i8]* @27, i32 0,
     ↪  i32 0),
4    i8* getelementptr inbounds ([1 x i8], [1 x i8]* @11, i32 0, i32
     ↪  0),
5    i8* getelementptr inbounds ([5 x i8], [5 x i8]* @2, i32 0, i32
     ↪  0)), dbg 943
6  %79 = zext i32 %tmp36 to i64, dbg 943
7  call void @tracerllvm_callOperandInt(
8    i8* getelementptr inbounds ([6 x i8], [6 x i8]* @79, i32 0, i32
     ↪  0),
9    i64 %79,
10   i8* getelementptr inbounds ([4 x i8], [4 x i8]* @13, i32 0, i32
     ↪  0)), dbg 943
11 call void @tracerllvm_instructionEnd(), dbg 943
12 ret i32 %tmp36, dbg 943
```

In the code above, the return statement (Line 12) is the only statement of the original code, all other statements are added for tracing. The needed information is collected and printed before the actual instruction is executed. The added functions just log the information to a file.

At first, in the call starting from **Line 1**, standard information about the instruction is collected (the same information is collected for most instructions). The first argument is the opcode of

[7] Again, the evaluation might have used a different code version of clang, but this feature is not crucial for the final results. Also all tools in the evaluation, if they use CLANG, used the same version of CLANG.

the instruction, in this case 1 (a return instruction). The second argument is the name of the surrounding function, the third argument is the name of the LLVM bitcode register about to be assigned, and the fourth argument is the type (as string) of the assigned register[8].

In **Line 6** we prepare the tracing of the operand of the return instruction by converting its type and storing the result in a temporary register. LLVM bitcode is typed, hence to avoid writing helper functions for each type individually, we cast values coming from the original code to a few pre-defined types, in this case to a *64-bit integer value*. If we would not do this, we would have to write several `tracerllvm_callOperandInt()` functions, one for each integer type (e.g. for *i1*, *i2*, ...).

The casted value is then used in **Line 7** to collect information about the operand itself (if there is more than one operand for the instruction we would add conversion and tracing calls for all operands). The first argument is the name of the LLVM bitcode register containing the value of the instruction operand, the second argument is the value of the operand, and the third argument contains the type of the register. In fact, besides the function `tracerllvm_callOperandInt()`, there are also functions for other types like `float`, `double`, `string`[9], or `vectors`; all working very similar to the `integer` logging function, collecting similar information but internally handling the value correctly according to its type.

The function in **Line 11** ends the logging step by merging all collected information into one JSON object which is then written and flushed to the tracing file.[10]

Safety Measures The linked library takes care to not trace any code inside itself, before the first instruction of the program main is called, and after main returned. Thus, we inject a tracer function as the first instruction in the `main()` function that opens the file the trace is written to and initializes the tracer as well as a function right before the return statement of `main()` that closes this file. Hence, we ensure that tracing information is only written once the file is opened and not written anymore once it was closed. Also, it should not happen that the tracer code calls instrumented code. To avoid such border cases, we put guards into the tracer that ensure no tracing is done while a tracing function is active (i.e. higher up on the call stack).

Whenever a new logging element is started, i.e. a new instruction is started to be traced, the previous logging is stopped beforehand by closing the logging element in the trace and flushing the information to the file to keep data integrity. This ensures that the information is correctly

[8] In this case no register is assigned, the value is just returned by the function. Thus the name of the assigned register is an empty C string and the type is *void*.

[9] C itself does not have string values, but there are functions working on either zero terminated char pointers or fixed width char arrays which can be interpreted as strings. If we instrument such a function, we interpret the used pointers as pointers to strings, hence we also dereference the pointer and output the referenced string.

[10] Besides the mentioned standard logging functions we implemented some special functions that trace other information. For example, we track basic block entering (e.g. for later coverage collection), but also special file information and other additional information to handle library functions like `fscanf` and `fgets` better.

put into our trace, such that we do not lose any information. With those safety measures we try to get as much tracing information as possible while avoiding unreadable traces and crashes due to our instrumentation.

Compilation Once the program is successfully instrumented, we get an instrumented bitcode file from the LLVM optimizer which applies our instrumentation pass on the original input bitcode file. This instrumented file can be compiled with Clang to an executable, which also includes libraries that contain the actual implementation of our instrumentation framework (the LLVM pass only adds function calls to our framework, no actual implementation).

Execution And Tracing The resulting binary can now be used like the original subject under test, except that it will produce a trace file. In detail, it produces an exact list of called instructions together with the concrete values used by each instruction. For some function calls we also print additional information, e.g. the strings used by the function calls. The trace is stored in a file alongside the subject under test and is also at least partially printed if the subject crashes (a behavior that is not uncommon when giving random inputs to a program)—thus, in most cases we extract at least some information about the program execution.

We log the following information, depending on which type of information we want to trace. The presented examples (from runs from the replication package [103]; prettified and shortened) explain how the tracing is done—they represent the different elements in the trace:

Command Line Input

```
{"av": [
  {"e": [
    {"p":"140726853641969","c":"/"},
    {"p":"140726853641970","c":"h"}
  ]}
]}
```

For a typical command line input of the program we trace the following information:

- `av`: The array of program arguments
- `e`: One program argument
- `p`: The pointer value to one character of one program argument
- `c`: The character stored at the respective position in memory

If the user wants to trace information coming from the command line, the tainting engine can mark those pointer locations as tainted. Thus, input from the command line is always reflected in the trace, in particular there is one JSON object with the key "`av`" and an array of elements. Each element is one command line argument stored in the second argument of the programs' main function (in C this argument is typically `char** argv`). For each such string we iterate the characters and trace the pointer value ("`p`") and the character given to the subject under test ("`c`").

Standard Instruction

```
{
  "oc": 30,
  "fn": "ini_parse_stream",
  "ar": "tmp43",
  "at": "i8",
  "a": [{"o": "tmp42", "v": "140726853639376", "t": "i8*"}]
}
```

For a typical LLVM bitcode instruction we trace the following information:

- `oc`: The opcode of the specific instruction
- `fn`: The name of the function surrounding the instruction
- `ar`: The name of the register in which the result of the instruction is stored (if existent, otherwise an empty string)
- `at`: The type of the result register
- `a`: An array of instruction arguments, the number of elements depends on the instruction
  - `o`: The name of the register used in this argument (for constants a dummy name is used here)
  - `v`: The value of the argument register or constant
  - `t`: The type of the argument register or constant

This information is used in the tainting engine to correctly propagate taints from the *arguments* to the *result registers* (or in cases like store instructions to other locations like the memory). In fact each instruction (as defined by the *opcode*) has specific semantics how taints should be propagated. For example, for a binary instruction like adding two values, the taints of both operands are propagated to the results; for a store instruction the taint of the stored register is assigned to the memory location and vice versa the taint of a memory location is assigned to the target register on loading.
Especially the *types* and *names* of the registers are important. The tainting engine maps the taints to the register names and the types define how many bytes are occupied if a taint is written to memory[11]. *Values* are important in cases like memory writes, where the location is defined by the value of one of the operands of the instruction. Also, the constant values of comparisons are obviously important, as they may be terminal symbols that we need for our input inference. The *function name* is mostly used for triggering *method entry* and *exit* events (which are used to calculate a stack), as well as marking functions as lexing (which will get important in Section 4.1.1). To avoid writing tracer functions specific for each an every instruction, we log all needed information for every instruction, reducing the implementation effort for the tracer, as it can handle most instructions equally and does not need specific implementations for the different instructions.

[11] This information is important as one could write a four byte integer to memory and then read single bytes from the written location. All four bytes need to be tainted, to make sure that all single byte reads are tainted as well.

Function Call


```
{
  "i": "  %call13 = call i8* @rstrip(i8* %tmp21), !dbg !121",
  "fn": "ini_parse_stream",
  "ar": "call13",
  "at": "i8*",
  "a": [{
    "o": "tmp21",
    "v": "140726853639376",
    "t": "i8*"
  }, {
    "o": "rstrip",
    "v": "18446744073709551574",
    "t": "i8* (i8*)*"
  }],
  "opt": []
}
```


For a typical function call in LLVM bitcode the following information is traced:

- `i`: The instruction string in LLVM bitcode format
- `fn`, `ar`, `a`, `o`, `v`, and `t` are the same as with a standard instruction.
- `opt`: Used for additional information for manually wrapped functions, like *fgets* and *gets* (prints the read string, including the source it is read from, and a manually defined opcode, which does not exist in LLVM, specific for the function)

A method call is a two step operation in our tainting engine: first, the engine initializes which registers of the caller are used as arguments for the callee—for propagating the taints once the callee is called—and setting the return register. The next step is a special instruction injected as first instruction into each method. This instruction triggers the building of the new call stack: initializing the argument registers correctly, setting the return register which will later be used when a return instruction is encountered, and setting a new taint scope in the tainting engine. This also means: if a function call to a library function is done, the second tracing call is not present and no new scope is added. For some *wrapped functions* (determined by checking the name of the called function in the *"i"* value[12]), we do not prepare a new stack, but call a specific handler in the tainting engine which takes care of correctly propagating taints and reporting complex comparisons (e.g. calling `strcmp()` causes a report of a string comparison).
In general we trace an instruction before it is executed to get the value the instruction works on, i.e. the *"inflowing"* values. Some functions though change the memory pointed to by arguments or return an important value, for those functions we put the trace call after the function to get the updated information instead.

[12]The exported string often contains another debug information pointer compared to the actual instrumented instruction. Though, we do not use this information, thus we can ignore this deviation in the resulting bitcode file.

Taint Tracking Once the program is instrumented and executed, we have to analyze the resulting trace to actually report our taint results. While the tainting engine implements the standard features of dynamic tainting as explained in Section 2.4 (defining the taint propagation for each instruction and initial taint creation), we also added features that are targeted to our use case of generating valid inputs for parsers.

First of all, we need to define which bytes to mark initially as tainted. In our case, we mark bytes coming from the *command line* or *stdin* as tainted, those are the typical input sources. Including file I/O would be possible but was not done in the evaluation of our prototype as we standardized the input source for all subjects to *stdin* to make the evaluation easier.[13] Once the initial taints are defined we need to make sure that they are correctly propagated through the program execution. Therefore, we define for each relevant LLVM bitcode instruction the taint propagation semantics and implement them in our taint engine. On top of that, we define semantics for some commonly used standard library functions. This gives us a taint engine which can follow data-flows on a byte level through the program execution.

One important feature of our tainting engine is querying taints at useful code locations. We need to define at which positions taints should actually be reported—not all instructions in the code are equally important. As already mentioned in Section 2.2.1, parsers implement their parsing information in comparisons against constant values (or at least values that do not come from the input). Hence, we monitor code locations that compare values and report respective taint information for those code locations in a form that can be used by our input inference module as presented in Section 3.2.2.

We observe LLVM bitcode comparison instructions (ignoring floating point comparisons as characters are typically represented as integer values). The most obvious instruction to observe is the comparison of two integers with `icmp` [88]. If at least one of the values is tainted we assume this to be a character comparison and report a tainted comparison by adding it to the comparison trace[14] which will be used for input inference (Section 3.2.2).

Also, we make use of the knowledge that the subject under test is written in C code and wrap different comparison functions from the standard library [103]. For example, if we see a call to *strcmp()* we can query the underlying memory locations (the compared strings) and see if some or all bytes of exactly one of the strings is tainted (ignoring taints attached to string constants). If so, we report a string comparison to the taint trace.

[13] This is not a limitation of the approach itself or the capabilities of dynamic tainting, both could handle file I/O as well—our approach partially implements the handling of file I/O. It is just easier during evaluation to have the same input source for all subjects to reduce specialization of the evaluation scripts towards the subjects.

[14] In general we filter out non-printable character comparisons, i.e. any character with an ASCII code less than 32 (which is a whitespace) except the line feed character (ASCII value 10) and any character with ASCII code greater equal 128. Typically such characters are not used in the formats we analyze: human readable inputs. This also filters some whitespace characters like a horizontal tab, which we accept as an approximation. Also, we consider comparisons against the *eof* character only if the compared input character is the last character of the input to reduce noise in the output.

Calling *strlen()* with a tainted string would taint the resulting integer value with the taints from the string used and marking them as a *strlen() taint*. We only do this if the string has a length greater zero and is not a string constant. We also attach such a string length taint for specific LLVM `sub` instructions. We interpret both operands as pointers and check if they point to a tainted byte. For the first operand we check the byte in front of the pointed location, because if the subtraction is a string length calculation the pointer might point to the zero terminator of the string which might not be tainted. If both locations are tainted bytes, we assume the approximation to be correct—the second operand points to the first character of a string and the first operand points to the first character after the string. Hence, we take the taints of the whole memory location between the two bytes (excluding the memory location the first operand points to, as this might be the zero terminator of the string) and attach them to the subtraction result.[15] If this value is now used in any comparison with a constant between one and twenty, we report a string length comparison. The string length comparison is an approximation and we limit the lengths to values that are typically used as input token sizes to reduce noise in the tainting output. With this special comparison the input inference module knows which string length a part of the input might need to have and may generate an input with matching length. Other function wrappers are also implemented tailored to the function they belong to.

Summary In this section we detailed how we made use of and extend general tainting ideas to create a tainting engine which is able to track characters through a parser and report valuable information which can be used in following steps to build syntactically valid inputs. The following section will explain how the generated taint trace is used to infer inputs from various program executions.

3.2.2 Input Inference

Inference Loop In this section we explain how we use the tainting trace (Section 3.2.1). We have already presented the pseudo code for our inference loop in Section 3.1. In general we *start with a random character*, let the program *run on this character*, *collect possible mutations* based on the comparisons done, *put them in a priority queue* and then *let the program run again*. We also check for every run if a syntactically valid input was already found[16] and also extend presumably valid prefixes with random character extensions to gain more information about the program under test. **In this section we will explain two crucial details of the inference loop:** the *comparison types used for input mutation* and the *calculation of the heuristic to rank the different possible mutations into a priority queue.*

Comparison Selection To reduce the number of options for mutations, we select *comparisons using the last compared character*—this is typically the first faulty character of the input, as a

[15] Note: for the prototype implementation it might happen that in rare border cases this approximation is wrong and cause a crash in the tainting engine, which might introduce random mutations.

[16] We discard inputs that are larger than 199 characters in most cases to avoid overly large inputs that typically only contain repeating syntactic features—only in some border cases they might be generated and executed, but typically we do not run the subject under test with inputs that are much larger or build new inputs from them.

recursive descent parser parses from left to right. In detail, we first select the last comparison[17] as we assume for a recursive descent parser to consume characters from left to right, hence we assume the last comparison should also contain the last compared character. For this comparison we extract the first, and thus typical minimal, used input index that was part of the comparison, this is our index used for selecting all comparisons that include this index—hence the **start of the last compared lexeme**. We also include comparisons using the operator *strcmp()* and *strlen()* that could consume more characters than exist in the input.

Pruning With pruning we check if we can discard all mutations for an execution trace to reduce the number of mutations to only relevant ones. For this, we check two things in the trace: *do we have recurring comparisons on the last few characters* and *is there any looping comparison done on input characters*.

For the recurring comparisons we check if the operands used in the comparisons on the last index are the same as the operands used on the indexes minus four to minus two (to have some reference set). In this case we can assume that the comparison is always the same on the last few characters and thus we can prune the trace information as there are likely already smaller inputs in the priority queue which handle the same code and produce similar mutations. We likely do not gain any new information at this point.

The check for new branches between comparisons is done by checking if there are ten or more different starting indexes in consecutive real comparisons (no *strlen()* or end-of-file comparison) excluding the maximum index and above without any change in stack or new branch covered[18] between those comparisons. A typical example would be, having a loop which takes one input character after another and compares it against some value, neither calling any other function nor running different code within the loop—for example, collecting the read characters in a second string. If this loop iterates too often, the whole input would be pruned as we assume that earlier generated inputs with less characters likely already triggered the loop often enough and more characters will only increase the loop count without triggering new code after the loop finished.

Out Of Bounds Index If the inputs are not pruned, we check if the found maximum index is larger than the size of the input. This might indicate an incorrect trace or other incorrect information from the trace and is used as a general fallback. In this case we add just one mutation, which appends a random character (from the Python constant *string.printable* [144])

[17] When selecting the last comparison, we filter out comparisons which involve some approximation of our tainting engine. For example, comparisons of type *strlen* might not always actually show the wanted length of an input string, but might be used for something else. Such approximations are fine during input inference, as the generation loop would test them by running the subject under test on the resulting input and discard incorrect approximations quickly. Having the last actually compared character though is of higher importance, as we need a valid prefix and only replace the first character that caused the parser to report an error. Hence, we under-approximate and filter comparisons approximated by the tainting engine.

[18] New means in this case not covered in this execution beforehand.

to the original input. This way, we run a similar input again while putting a new character outside of the checked bounds to analyze how the subject under test reacts to this new input.

Comparison Types All filtered comparisons are used to create a set of possible mutations to the input they are stemming from. Hereby we iterate through the list of comparisons, and based on the type we replace the respective input index with a new value:

Switch For *switch*-statements we take every option of the switch and use it as a replacement. For switch statements every case is specifically implemented by the developer of the parser, hence we assume all cases to be important for the overall parsing success.

Strlen For *strlen*-statements we just generate a string consisting of only *'a'* characters in the length the *strlen()* comparison expects. For this to understand one has to understand the *strlen()* operator: the tainting engine uses this operator if an integer value is compared against another integer and a string length taint is involved, meaning that the tainting engine assumes the comparison to be a comparison which checks if the input portion has a certain length. By adding a substitution with a string that has this length we try to fulfill this requirement, hence we add a random but fixed string with only *'a'* characters.

Conversion *Conversion*-statements stem from *strto* standard library functions—in this specific case from *strtod* and *strtoul*. For both we replace the characters that were tried to be converted to strings from a random but fixed list of strings that represent the respective numeric values, while fixed means we defined a list of different options the respective function would accept. For example, for *strtod* one value we defined is `3.0E2` and for *strtoul* `-0x1F` was defined. Hence, we get for each conversion a list of possible substitutions that can be applied. The list of functions and replacements can be extended.

Other For all other statements we just take one of the compared values from the operands and use it as replacement.[19] For an equality comparison like *strcmp()* or character equality (e.g. "`c == 'a'`") there is just one option which could be taken, but in some cases we have comparisons where characters are checked to be in a range (either by comparing if they are larger or smaller than a certain value; or in some cases a certain value is first checked to be larger and then smaller, or vice versa, indicating a specific range the character should be in)[20]. In this case any character in the range is fine.

Random For each run, we also add two "random" mutations, i.e. we add two mock-comparisons simulating a comparison of the last compared input character against a number and a

[19] Due to an implementation artifact from older versions of the tainting engine, end-of-file comparisons get replaced with either the character "-" or "1", which introduces some noise in the generated inputs, but this noise might also be helpful to explore new parts of the subject under test.

[20] For example, say we have two comparisons in our tainting engine, the first comparison is $i_1 \geq 65$ and the second one is $i_1 \leq 90$ (with i_1 being the input character at index 1), then only one comparison is given to our inference engine: $i_1 \in [A \ldots Z]$, i.e. i_1 being a character from the list containing all characters between *'A'* and *'Z'*. Hence, we combine the two separate character comparisons to one range comparison. We just take the characters used for comparison as range boundaries, hence if a greater than or less than operator is used we might get an off-by-one error, which can be used during input generation to test possibly wrong but close to specification inputs.

letter. In some cases it happens that letters or numbers are implicitly accepted if nothing else matches, i.e. there is no concrete comparison against those values. In other cases, the comparison might be the opposite: *the value is checked to not match another value*. This often happens if variables or numbers are parsed: the code could check if an input character *'C'* is greater or equal than the letter *'A'* and less or equal than the letter *'Z'* (analogous for numbers and the characters *'0'* and *'9'* as well as non-capital characters). It is equally valid though, to check if the input character *'C'* is less or equal than *'@'* and greater equal than *'['* (analogous for numbers and the characters *'/'* and *':'* as well as non-capital characters) and if not accept the character or number as valid. Hence, we approximate such comparisons by adding them into the pool of possible substitutions. Those random inputs will be ranked alongside other comparisons with substitutions of length 1 and are thus, when looking at the overall input generation loop, mixed into the stream of comparisons.

Heuristics We filter the taint trace for the comparisons on the last character and apply the following heuristics to rank the resulting comparisons for their usefulness. The numbers in the list also indicate in which order each part of the heuristic is applied. When comparing two comparisons, the second item of the heuristic value is only used if both comparisons have the same value for the first item. Analogously, the third item is only used if the second item is equal. The input ID is always a tie-breaker, as it is strictly monotonously increasing. A smaller heuristic value ranks the comparison higher in the queue. Our heuristic contains the following:

1. Number of inputs that already took the same path
2. A combination of different values:
   - Length of the input (if new branches were covered; else 100)
   - + average stack size after the comparisons of the last compared character/lexeme (if new branches were covered; else 0)
   - − number of newly covered branches (if new branches were covered; else 0)
   - + sum of the length of the same prefixes in already found valid inputs
   - + 1 for every 5 inputs on the path of the generation tree
   - − the length of the substitution times 2
3. The size of the difference of the stack to the parent
4. The length of the input
5. The input ID

Item 1 Let us explain for each part of the heuristic how it used and why we incorporated the value into the heuristic. Item 1 of the list, the number of inputs with the same path, is used to avoid duplicates.

Typically, whenever we talk about paths, we mean the branches traversed until the occurrence of the last comparison of a character or lexeme in front of the maximum index (in general the start of the last lexeme) including the branches traversed until the next comparison. Except for some border cases, this collects the branches until the last character is started to be parsed—*which approximates the coverage based on the valid part of the input.* The last lexeme is often invalid because we append random characters during input generation to new inputs. For this path calculation we store the starting basic block of each branch traversal exported by the tainting engine, i.e. only the first time a branch is traversed in a run it is also exported. This is an under approximation of the actual path, but reduces the resources needed while still being precise enough for our needs. If we have already generated an input that followed the same path through the program, it is worth checking this input first and evaluating all resulting options of it. Hence, inputs on new paths are preferred, similar to one of the heuristics used by AFL [160].

Item 2 This value combines different aspects of the performed execution which cannot be put in a specific order, but the combination of all values is important.

Input Length The calculation starts with the length of the input that was used to generate the trace. In general small inputs are better, as we want a diverse set of inputs containing diverse grammar features. This value is only considered if new code was covered by the input, because only in this case we want to prefer shorter inputs. Otherwise, it is more important to find inputs that cover new code, thus the value is set to 100 for inputs that do not cover new code.

Average Stack Size We then add the average stack size seen after every comparison of the highest index character up to the next comparison, as we want to penalize larger stacks (remember that each new nonterminal symbol of the grammar introduces a new function on the stack during parsing until the nonterminal symbol is fully parsed). *In detail*: we first collect the highest starting index used in any comparison, collect all stack changes (function call or return) between a comparison starting with the highest index up until the next comparison or the end of the trace, and get the sizes of the stacks of those events. Once we collected all stack sizes in one list, we calculate the average stack size (sum of stack sizes divided by length of the list).[21] If this value is not available, a dummy value of 100 is used. If no new code was covered by the input, this value is ignored.

A larger stack means we are likely deeper down in a recursion. The less complex the input, the better, hence we want to avoid large recursion depths, which means we favor inputs with a small recursion depth—if we do already cover new code with the valid prefix.

Newly Covered Branches Inputs that cover new branches that were not covered by previously found valid inputs are worth to explore[22]—the more branches are covered, the more

[21] Actually, the original *Parser-Directed Fuzzing* [105] paper has a slight error when describing this part of the heuristic, we describe the correct version here.

[22] The coverage information we use for this heuristic is the same as for the part of the heuristic that calculates the same paths taken (Item 1).

interesting the input is. Still, if the input already covers new code, we want it to be short and we want to find a suffix which makes the input syntactically valid. Hence, if new code is covered, *input length*, *average stack size*, and *this value* are considered—making the input short and preferring inputs that cover more code and have less open syntactic features (less parsing functions on the stack). Hence, we include the number of branches that were not covered beforehand.

If we run out of inputs that cover new code, i.e. if this value is 0, we are on a plateau in our search space. *Input length*, *average stack size*, and *this value* are not considered and instead a value of 100 is taken, penalizing this input vastly. In this case we prefer inputs that are different from other, already generated inputs—hence we use the *length of the same prefixes*, the *number of ancestors* and the *substitution length* as described below. For inputs that do not cover new code it might even be important to increase *input length* and *stack size*. For example, in general parsing an *else* keyword successfully can only work if we have already seen an *if* keyword. Thus, we would need to append the *else* keyword to an input like `if (1 < 2) {...}`, likely resulting in a higher stack and longer input.

Same Prefixes We store the already found valid inputs in a set[23] and for every newly generated input we sum up the length of equivalent prefixes, which yields a similarity value. Inputs with a small similarity value are preferred, because they likely yield more syntactic diversity. For example: for a programming language we already generated the inputs `while(3<4)...`, `while(3<=4)...`, and `if(3>4)...`. Now it might happen that during generation we have the choice to build our next input based on the prefix `while(3>=` or `if(3>=`. While in theory they are similar, we already have two inputs with `while`, hence generating an input starting with `if` gives more diversity in the final set (which might be interesting for input format learners). Hence, this heuristic would show that the `while(3>=` prefix has a similarity score of 14 (there are two valid inputs with a same prefix length of 7), while `if(3>=` only has a similarity of 5 (there is only one valid input with a same prefix length of 5), hence we prefer `if(3>=` for now.

Number Of Previous Inputs Every input is generated from a parent input and a mutation based on a comparison—resulting in a generation tree. We want to have a breadth-first search like expansion algorithm (which prefers shorter, grammar-feature rich inputs), hence we prefer inputs that are close to the expansion root (which does not necessarily correlate with the input size as the last character could be replaced several times, leading to the same input size but a larger expansion path). To avoid having a breadth-first search only algorithm we went for some middle ground and allowed an expansion path length of five before penalizing the input, hence only every fifth input on the path will increase this value.

Substitution Length The length of the substitution is also important as we do not only have character substitutions but also complete lexeme substitutions (from string comparisons in the code). While character substitutions are important (as control bytes and filler

[23] An input is stored if it causes the subject under test to finish with exit code zero or a timeout and covers new branches not yet seen when considering all covered branches of the execution.

values), the larger lexeme values are the more interesting ones, as they often enable new features in the code. Thus, we prefer larger lexeme substitutions over single character substitutions, in detail we prefer longer over shorter substitutions because we subtract the length of the substitution times two. This feature is specifically interesting if we use our produced inputs as seeds for a mutation based greybox fuzzer like AFL [160], as those fuzzers struggle with generating such large lexemes if they are consumed as one entity in the code. Without further knowledge, the fuzzer does not get gradual coverage feedback while fuzzing but rather has a coverage plateau until the correct lexeme is found.

The combination of different values is needed to adapt the heuristic based on other, already generated inputs. For example, if we have a really long input that covers many different branches, it might still be useful to first try a set of shorter inputs that cover less branches. Those shorter inputs might be easier to close (make syntactically valid) than the larger one because they might have less features opened (e.g. less opened parenthesis that need to be closed). On the other hand, if we have a slightly larger input which covers much more new code, it might be worth exploring this prefix first. If we can make it valid fast we will gain much more coverage (thus likely covering more features) in a shorter time. Hence, no portion of this part of the heuristic should be preferred over the other.

Item 3 Also, we check how much the generated input differs stack-wise from the input it was produced from. With this value we want to prefer inputs that are the farthest away from the parent comparison, hence inputs that produce diversity in the program coverage. *If both stacks are different* at a certain point, we return the size of the common prefix, hence comparisons that deviate earlier in the call stack are preferred, those explore other parts of the program. *If the new stack is larger (and the parent stack is equal to the start of the new stack)*, we return the size of the new stack as we want to prefer inputs with a smaller stack. *Otherwise (if the parent stack is larger and the new stack is equal to the start of the parent stack)*, we return the size difference between the new stack and the parent stack. This case is different from the others, here we prefer inputs that are closer to the parent comparison. The reason is that in such cases we might be on the *"closing"* path of a valid input—we found a character that finishes a feature and reduces the stack size. It is possible that somewhere higher in the parent stack other parsing options could be explored that we could miss if we close too fast, hence we want to perform those closing calls in small steps, so we prefer stacks that are closer to the parent stack.

Item 4 and Item 5 Finally, if all other parts are equal, we prefer shorter inputs over larger ones with Item 4—especially since a parser does not backtrack, hence equal prefixes should not change the execution for different suffixes. The final tie-breaker is Item 5: we prefer older inputs over newer ones using a strictly monotonously increasing ID. If two inputs have an otherwise identical heuristic value, we use older inputs first to maintain the temporal order. This way we get a heuristic that uniquely sorts every input into the generation queue.

Queue Recalculation Obviously some parts of the heuristic value are dependent on already found valid inputs, e.g. we incorporate the *number of newly covered branches in relation to the*

branches covered by valid inputs and *the same prefix metric is based on already found valid inputs.* Hence, once a new valid input is found, we need to update the respective part of the heuristic value for all other inputs in the queue that were not already explored—we re-evaluate the queue. For each value in the queue we store all important information to calculate its heuristic value, hence once we update the information gathered from the valid inputs, we can iterate over all inputs in the queue and re-build it using this new information.

Derivation Tree With our priority queue backed by the heuristic value explained above, we can select promising inputs fast and run the subject under test to collect information about the parsing step and generate syntactically valid inputs. In fact, since almost every input except for the initial one is a mutation of a previous input, the inheritance spans a derivation tree, like the one in Figure 3.6. The root of the tree is the initial value used for our derivations. From there, different child nodes are produced (the values in the nodes), based on the comparisons seen on the last character (the labels on the edges). The nodes show the new input including the randomly appended character, the inputs are tested first without the randomly appended character for validity. The gray path would be the perfect path through the tree until the valid input "(2)" is produced. In some cases it might happen that a comparison does not lead to a larger prefix; in those cases the children of such a node would have the same size but other replacements. In fact, it could even happen that the size of a child node is smaller.

3.3 Limitations And Assumptions

Nothing is perfect and the ideas and contributions in this thesis are no exemption from this. In Section 1.2 we already discussed the contributions of this thesis. To make clear under which circumstances those apply, each chapter will contain a section about limitations and assumptions. The limitations introduced in this chapter will partially be relaxed in Chapter 4 and Chapter 5. Those chapters will introduce improvements over the approach introduced here.

The subject under test's parser must be a recursive descent parser.
This is the most important limitation. The reason for this is simple: a recursive descent parser reflects the underlying structure of the parsed format, which we can use. Especially, we require the parser to perform character comparisons at some point (or comparisons on character ranges). Recursive descent parsers typically fulfill this requirement and are therefore very well suited as test subjects. Another typical implementation detail of recursive descent parsers is that during parsing comparisons between input characters and constants are only done if the constants are (with a very high likelihood) valid at the very position in the input. We will make use of this to decide which characters to use during input inference.

The subject under test should parse left-to-right.
Jumps to already consumed lexemes during parsing might misguide our input inference. Concretely, if a parser jumps back and forth between the input bytes while parsing, our

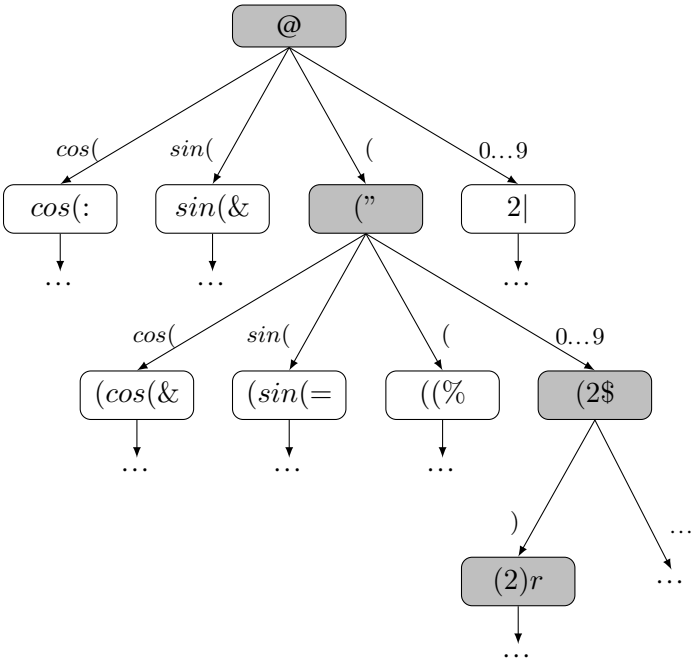


Figure 3.6: A sample input derivation tree, based on our arithmetic parser (Appendix A-I), which shows how the different inputs are related to each other. Each node also contains a randomly appended character that would be used to build the resulting nodes from the comparisons on that character. The gray nodes show the path through the tree which results in the input "(2)". Even if a valid input was found (like "(2)"), another random character will be appended to check if further larger syntactically valid inputs can be produced. For better readability we do not show all comparisons here.

approach would not be able to select the correct characters for replacement. We assume that a parser accepted a character as valid if it starts comparing the subsequent characters during parsing, because this parsing style is recommended by the *Dragon Book* [3] and a typical pattern in compiler and parser design.
If the parser would not consume the input from left to right, we would need much more backtracking. For example, suppose the parser accepts function definitions, e.g. `int sin(){return 5;}`. One could imagine a parser that first accepts any function name (in this case `sin`), then parses the argument list (here empty) and the body; and then checks if the function name is in a list of pre-defined function names (so in the terminal symbol set). In such a case our approach would generate the function definition including the body step by step, needing several iterations, only to backtrack to the function name, replacing it with a value from the pre-defined list and then restart from there.

Semantic correctness is ignored.
This means our tool is blind to the semantic validation phase of the subject under test which typically comes after the recursive descent parser. This comes with a second

problem: if the subject reports parsing errors non-distinguishable to semantical errors, our approach would have to also generate semantically correct inputs which is out of scope (as this would again mean backtracking). Many parsers work in phases: they first check for syntactical correctness and then check the semantics of an input. Now think of generating a valid C function definition, containing lots of syntactical elements—only after the whole function definition was generated syntactically valid, the semantic phase of the parser might recognize that a variable is not defined. This causes our approach to delete everything after and including the first undefined variable usage, then building up another syntactically valid function definition with a different suffix after replacing the invalid variable with a valid one. Hence, the approach would need many more iterations as it cannot be certain that a syntactically valid prefix is also semantically valid without generating a full syntactically valid value.

A lexer in the subject under test will decrease the input generation efficiency.

A lexer usually accepts a lexeme ignoring its position in the input, converts it to an internal token, and uses this token in the parsing phase—which might then reject the value because it expects another lexeme at the position in the input. Technically, this means that the character comparisons we rely on happen in the lexer, not in the parser itself. The approach explained in this chapter though assumes that if a character at a certain position in the input is compared against another character, the compared constant is syntactically valid for this position. If this is not the case, the pool of possible replacements (i.e. the search space) for each character in the input increases substantially, increasing the overall runtime for generating valid inputs.

The input generation heuristics are tailored to recursive descent parsers.

While also being part of the limitations beforehand, we want to make the usage of heuristics and assumptions tailored to recursive descent parsers an explicit item of this list. Section 3.2 details how we assume the code to look like and which parser specific execution information we use to generate inputs. In this limitation, we want to clarify that in theory a developer could implement a parser in any imaginable way, causing our heuristics to either be of no use—or in worst case make them guide the input generator in a wrong direction. In Section 2.2.1 we already show how a parser by the textbook looks like by presenting the concepts of Aho et al. [3]. In Section 3.2 we justify the used heuristics based on those basic concepts. We believe that most developers actually follow those theoretic parser designs.

Subjects under test must be written in C and compilable with Clang.

This is mostly a technical shortcoming, and a limitation that is only bound to the current implementation of our approach, not the overall idea—the framework can be implemented for most other languages as well. Our general assumption is that the input characters can be tracked throughout the execution of the subject via dynamic tainting and that we can query the dynamic tainting engine on comparisons to gain knowledge about which input characters are compared against which constants. Also, we require distinguishable

function calls to build a stack. Both are features that most programming languages have, hence the restriction is not severe on a theoretical level.

3.4 Summary

In this chapter we introduced our basic idea for generating syntactically valid and diverse inputs for programs with a recursive descent parser. We start with a *random input*, *analyze the comparisons* done on the input characters (using dynamic tainting), and use the *comparison values to create new inputs*. We implement a tainting engine which is specifically designed for our needs during parser analysis. The resulting taints, in combination with our recursive descent parser specific heuristics, are used to iteratively build syntactically valid inputs for the subjects under test. The heuristics also include information about the program executions like covered code, which makes it possible to focus not only on the syntactic validity of the inputs, but also on their syntactic diversity.

4 | Detecting Syntactical Conditions

In Section 1.1 we outlined the objectives for our thesis and in Section 1.2 we explained the different contributions to those objectives. **This chapter contributes to solving our second objective: generating inputs from subjects that use a tokenizer**. In the following sections we extend our approach, *improving the generation of syntactically valid and diverse inputs for subjects which use a lexer besides the parser*. Our goal is to use the techniques as presented in Chapter 3 even in the presence of a lexer—this was not possible beforehand because we would lose the needed dynamic taints in the lexer and they would never end up in the parser.

Scope In Section 2.2.1 we have already mentioned that more complex input processors have *a lexing and a parsing phase*. As we can see in Figure 4.1, a lexer introduces new indirections in the overall information flow from the input characters to the parser comparisons. Concretely, the lexer breaks the direct data-flow that was used in Chapter 3 to produce tainting information for the input inference phase. Consequently, we would only see lexer comparisons which are, in contrast to parser comparisons, not restricted to the validity of a lexeme at a given position. Parser comparisons are typically done in the context of nonterminal symbol parsing, hence every constant value seen in such a comparison should be a valid replacement for the compared portion of the input. A lexer on the other hand only checks if the given input portion is a valid terminal symbol—only the value returned by the lexer is then checked in the parser to validate correctness at the given position. In this case, our basic approach will only recognize the lexer comparisons and thus, instead of replacing an invalid character with a valid character, it will replace invalid characters with any lexemes from the input space—decreasing the likelihood of having a valid replacement. This leads to a much larger search space as it might happen that we substitute the same input byte location several times with invalid values until a valid continuation is found. Thus, this chapter details a technique to also analyze a parser in the presence of a lexer—introducing lexer analysis and information propagation for input inference.

Approach Overview *In short, we extend our initial input generation loop with additional token information, making it possible to analyze lexer code, look for token generation patterns, and apply our input generation techniques from Chapter 3 also in the presence of a lexer in the subject under test*. The following abstract algorithm shows a very brief overview on our approach:

```
1 def inputInference(subject):
2   inst_sut = instrument(subject)
3   input_queue = [random.nextChar()]
4   token_information = TokenInformation()
5   while True:
6     if token_information.hasTokensToLearn():
```

```
enum lex_token{PLUS, MINUS, MULT, DIV, PAREN_L, PAREN_R,
SIN, COS, NUM, WS, UNDEF, END};
enum lex_token token = UNDEF;

void next_token_non_whitespace() {
  if (input[pos] == '+') {
    pos++;
    token = PLUS;
//...
  } else if (input[pos] == '(') {
    pos++;
    token = PAREN_L;
  } else if (pos + 4 < input_size &&
        !strncmp("sin(", input + pos, 4)) {
    pos += 4;
    token = SIN;
  } else if {
//...
  } else {
    // lexing error
    printf("Undef token at %d: %c", pos, input[pos]);
    token = UNDEF;
  }
}

void next_token() {
  skip_whitespace();
  next_token_non_whitespace();
}

int atom() {
  if (token == PAREN_L || token == SIN || token == COS ) {
    next_token();
    if (expr()) {
      if (token == PAREN_R) {
        next_token();
//...
}
```

Figure 4.1: An arithmetic expression parser with a lexer, partially showing the parsing routine for the *atom* nonterminal symbol from the grammar from Figure 3.2. The full lexer and parser can be found in Appendix A-II.

```
7         inp = token_information.nextTokenToLearn()
8       else:
9         inp = queue.pop()
10      trace = inst_sut.run(inp)
11      taints = taintEngine.analyze(trace)
12      new_inputs = extract_inputs(taints, inp, token_information)
13      input_queue.sortAndAdd(new_inputs)
```

At the beginning of Chapter 3 we already introduced the overall input generation algorithm. In this chapter, we extend our algorithm to also support additional token information. Concretely, we have an overarching container which collects token information throughout the execution loops (created in Line 4). While generating taints (Line 11) we also create a new class of taints, called *token taints*, which are attached to generated tokens during the lexing process.[1] Once values attached with those taints are used in comparisons, the tainting engine reports the resulting token comparisons (Line 11). Besides the new inputs based on character comparisons as before, our input generation now also creates inputs using the new token comparisons (Line 12). In the parser tokens are typically just integer values, thus *token comparisons are integer comparisons*. Hence, we also need to constantly update information about the lexemes that generate the respective integer value—which is done during the input generation (Line 12). Also, to improve the quality of our token information, we *learn tokens* (Line 6) by giving them one by one to the subject under test (Line 7)—besides the constant parallel learning on every input.[2]

4.1 Approach

The idea of *Parser-Directed Fuzzing* is built on the assumption that we see all comparisons in the parsing code. Hence, we need to extend our approach in such a way that during input inference we have the same information, with and without lexer code.[3] We need to solve three problems in order to shift the comparisons from the lexer to the parser if a tokenizer is present and make them usable:

1. *Differentiation between lexer and parser code.*
2. *Attaching taints to lexer tokens.*
3. *Mapping of lexer tokens to their source input characters.*

In the following, we detail for each problem mentioned above why it is relevant and how we solve it. We describe the concepts and reasoning behind the solutions and explain how this can be done in general for the domain of recursive descent parsers.

[1] The token information that is initialized in Line 4 is not used during taint generation.

[2] For clarification: our algorithm is presented as infinitely running (Line 5). Though, in Section 4.1.4 we will see, that we actually can have a stopping criterion to make if possible to combine our algorithm with subsequent tools—Chapter 5 goes into more details about this combination with additional tools.

[3] In Chapter 3 we mentioned that the main conceptualization and implementation of our dynamic tainting engine was implemented for my master thesis [101]. The extensions to detect tokenizing code and taint tokens were not part of the master thesis and are thus a contribution to this thesis.

4.1.1 Differentiating Lexer And Parser Code

As shown in our example code in Figure 4.1, the lexer and parser are two inherently different portions of the code—using a different code style and different values [3]. Aho et al. describe the lexer as follows:

> *"As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program."* [3]

The parser on the other hand is described as follows:

> *"In our compiler model, the parser obtains a string of tokens from the lexical analyzer [...] and verifies that the string of token names can be generated by the grammar for the source language."* [3]

As a first step, we need to differentiate parser and lexer code. Differing those phases brings the benefit that we can easily filter comparisons and categorize them: a lexer should never compare token values and a parser should never access input characters directly. While it is easy to differ between input character taints and token taints (we can attach different kinds of taints to the values, depending on the taint source), we might see some usages of tokens in the lexer itself, which are not part of the parsing process though:

```
1  int concreteAlpha(char c) {
2    // some complex code
3    switch(c) {
4      case 'A': return UP_A;
5      case 'B': return UP_B;
6      //...
7      case 'Z': return UP_Z;
8      case 'a': return LO_A;
9      //...
10     case 'z': return LO_Z;
11   }
12 }
13
14 int concreteNumber(char c) {
15   // some complex code
16   switch(c) {
17     case '0': return NUM_ZERO;
18     //...
19     case '9': return NUM_NINE;
20   }
21 }
22
23 int nextTok(char c) {
```

```
24    int tok = UNDEF;
25    if (isdigit(c)) {
26      tok = NUM;
27    }
28    if (isalpha(c)) {
29      tok = ALPHA;
30    }
31    if (tok == UNDEF) {
32      // some complex code
33    } else {
34      // complex code preparing number/alpha tokenization
35    }
36    if (tok == NUM) {
37      tok = concreteNumber(c);
38    }
39    if (tok == ALPHA) {
40      tok = concreteAlpha(c);
41    }
42    return tok;
43  }
```

As we can see in the artificial example above, in Line 26 and Line 29 a token is generated, in this case to distinguish for alpha and number tokens. Now depending on the token value, either some unrelated complex code is executed or some code preparing the tokenization of an alpha or number is executed (Line 31 and following). Depending on the initially read token a concrete version for tokenizing an alpha or number is called, which in turn contains complex code dependent on the code in Line 31 and following. In other words: it is first checked which complex code needs to be executed and once this is done, the initially used token is again used in the lexer to differentiate how it is further refined. Hence, we have a token comparison in the lexer which we only expect in the parser. Such a comparison though is not an actual parser comparison, hence the differentiation between parser and lexer is not always clear.

We will mainly use three typical implementation details to differ both parts and mark the respective functions of the subject under test accordingly:

1. Lexer code consumes characters (by using comparisons on input characters), while parser code typically does not use any input characters but works on tokens.

2. Parser and lexer code lie in different functions/parts of the code.

3. Lexer code never calls parser code[4].

The lexer is the "glue" module between the parser and the operating system, i.e. it handles the input reading, system details (like different ways to indicate a new line: "`\r\n`" and "`\n`"),

[4] Aho et al. schematically show in Figure 3.1 in the *Dragon Book* [3] how the parser requests tokens from the lexical analyzer and just accepts the returned tokens from the lexer, hence lexer code needs not to call parser code.

and converts the read characters to some system agnostic and program specific tokens. Thus, lexer code should usually not work on lexer tokens, it should only generate them, but never use them in comparisons. Also, parser code should never access input characters directly in comparisons (the parser decides based on the consumed tokens which parsing path to take). With this information we can approximately divide the code into lexing and parsing code.

A lexer is essentially an input provider for a parser, meaning that the parser is typically in charge of the organization of the input validation process. Hence, either the lexer tokenizes the input before the parsing steps start (and the parser then consumes the tokens given by the tokenizer) or the parser calls the lexer whenever needed to request a new token (as we do it in our sample code). In this thesis we assume the lexer to be queried by the parser for each lexeme individually because we assume this to be the most common and intuitive way of lexing inputs. Especially, because this lexing method is more efficient as only one or a few lexemes need to be kept in memory and if a parsing error happens the lexer only needs to work up until the first error. Thus, in a typical parsing setting, the lexer code is always higher in a call stack than the parser code, making any function called from lexer code also a lexer function.

In addition to the division into lexing and parsing code, an encapsulation of lexing and parsing in one or more different functions is essentially needed to produce maintainable and readable code. The *Dragon Book* [3] also mentions this distinction between lexer and parser as the default that should be used, hence we make use of this division of code to make the detection of lexing and parsing code easier.

An under-approximation of the lexer code is favorable in this case as we will later not report token comparisons for comparisons that happen in lexer code, thus we rather do not mark a function as lexing and report more token comparisons than vice versa. During input generation a wrongfully reported lexer comparison just yields additional runs to test the comparison; a missing token comparison though might hide a complete branch of our search space. If a token-comparing function is marked as lexing and not considered as parsing it might happen that a complete nonterminal symbol of the underlying grammar is missing as parser functions and nonterminal symbols are tightly coupled.

Summary Essentially, any function that uses input characters in comparisons is marked as lexing, any function that is called from a lexing function is also marked as lexing and any other function is assumed not to be a lexing function until proven otherwise.

4.1.2 Token Taint Generation

Arguably, the most important part of our approach is the propagation of character taints to token values (in form of token taints). The approach from Chapter 3 is blind to comparisons after the lexing part, but if we are able to transfer the taints from the input characters to the generated token values, they will finally end up in the parser and are then used in the parser comparisons we want to analyze.

One of our core additions to the dynamic tainting backend of *Parser-Directed Fuzzing* (as described in Chapter 3) is the recognition of tokenization patterns, hence we need to go into more detail about how those patterns are crafted and why we believe that they will correctly recognize tokenizing code while ruling out other code constructs. We want to avoid over-tainting, i.e. we want to avoid that every control dependent value is tainted, but also detect as many token generations as possible. In this part we go into detail on the following solutions:

1. *Recognition of token generation code*
2. *Precise attachment of (token specific) taints to lexer tokens*
3. *Reporting token comparisons by the dynamic taint engine*
4. *Limiting the lifetime of information flow taints*

The following code gives a brief overview on the token taint generation cycle:

```
1  def handleTokenTaints(execution_trace):
2    container = TokenTaintContainer()
3    for instr in execution_trace:
4      if isCharacterComparison(instr):
5        container.storeTaints(instr.getTaints())
6        container.used = False
7      if isTokenPattern(instr):
8        instr.addTaints(container.getTokenTaints())
9      if isTokenComparison(instr) and hasTokenTaints(instr):
10       reportTokenComparison(instr)
11       container.used = True
12     if container.used and isDeletionInstr(instr):
13       container.clear()
```

We start our tainting algorithm with an empty container which stores information about tokens as well as the tokens themselves during execution (Line 2). Then, for every instruction (Line 3), we perform analysis steps, which handle the decision which taints are stored (Line 4), attached to tokens (Line 7), reported (Line 9), and when they are deleted (Line 12).

Token Generation And Taint Attachment The token generation itself is split in two parts: *the comparison of input characters* (Line 4) and *the actual generation of a token* (Line 7). Whenever a comparison using a tainted input character is encountered we remember which characters are used in the comparison (Line 4). If we already have one or more input character taints remembered, the new taints are added to this set. Often, we also check if the operands are equal because only for a found and matching lexeme a token is generated in general. This is an approximation to limit the noise of generated token taints later on: a token is a representation of a specific character, string or a set thereof, thus we expect in general the lexer to check for this set membership. And only if a membership is detected, the respective token would be emitted, hence the restriction to successful comparisons. Were necessary, we also filter comparisons which have an operand with a token taint attached, because tokens should not be used in lexing

code. Finally, once a token is actually produced (typically a constant value is assigned to a variable or a memory location), we attach a token taint to this value and link the taint to the input character taints remembered (Line 7).

Token Comparison Reporting For switch statements or 32-bit integer comparisons involving exactly one tainted value (as comparison operand or switch value; the taint must be a token taint) in a non-lexing call stack[5], we report a token comparison (Line 9). Actually, we also report which input characters were involved in the comparison for later inference which lexeme the token tainted constant represents (see Section 4.1.3 for details on how we get this information).

Limiting Token Generation Lifetime One important thing to do besides storing taints for token generation is also deleting them at certain points during the subject execution (Line 12), otherwise we would accumulate all input taints over time. The removal of those taints only happens if the taints were used beforehand as we assume all taints from accessed input characters after the last token usage must belong to the next token—especially because the lexer is in most cases called between parser comparisons to query the next token. Once a new taint is added to the token container (Line 4), the taints in the container are flagged as unused. The taints are flagged as used once a token comparison is done, i.e. once a token taint is reported as defined above. With this usage restriction we make it possible to let the stored taints survive until an actual comparison happens, making it possible to let different token generation patterns use the same stored taints. Even though a token is typically generated at one position, with this we can ensure that we do not miss any token generation sites because of wrongly approximated pattern matchings. Also, in some cases a token is generated from several comparisons, e.g. when lexing a variable or a keyword consisting of several single characters. In such cases it makes sense to add the taints from several characters, which then build the source taints for the final token taint.

Summing Up This section describes the additions to the tainting engine, making it possible to not only track direct data flows through the program execution, but also attaching tainting information to tokens generated while lexing an input. We achieve this by 1. only considering comparisons that are typical during token generation (i.e. lexer comparisons; Line 4), 2. using the taints involved in those comparisons and attaching them to values created by generic token generation patterns (Line 7), 3. reporting comparisons involving those tainted values in controlled circumstances (Line 9), and 4. defining deletion points for temporarily remembered taints for token taint generation (Line 12). Our decisions are mostly based on foundational software engineering concepts (like modularization of code) and compiler design patterns as described in the literature [3] (like the differentiation between lexer and parser) to bring our design choices as close as possible to real world implementations.

The token taint generation in general is an approximation—we might miss some token generating code, we might taint constants with token taints which are not tokens, and we

[5] All functions in the call stack must be non-lexing. For switch only the switch value is checked for taints, all other values are not checked further.

might attach the wrong or not all taints to token values. Later on, in the mutation engine (Section 4.2.3), we use different measures to reduce the impact of the incorrect token values we possibly introduce here.

4.1.3 Mapping Lexer Tokens To Input Values

Now that we detailed how token taints are generated, propagated, and used in the parser, one final information is missing: *how can we use the tokens in our comparisons*? A token is simply a value in the code with no further meaning (as we can see in Figure 4.1, the lexer converts input characters to enumeration values, which in turn are syntactical sugar for integer values). Hence, we need to extract a mapping from a token value to it's respective source characters and vice-versa. Otherwise, once we see comparisons in the parser, we cannot know what substitutions to apply. If we look at the code in Figure 4.1, Line 32, we see three comparisons against enumeration values: *PAREN_L*, *SIN*, and *COS*. Without a mapping we would only know that the token generated from the respective input characters are compared against some integer values. With the mapping though, we know that *PAREN_L* is generated when lexing an opening parenthesis (Line 10 and Line 12), *SIN* comes from the input *"sin("* (Line 14 and Line 16), and *COS* from *"cos("*. With this information, if we want to use such a comparison for substituting parts of an input as we did it in Chapter 3,we can directly look up the token value from the mapping and apply the correct characters at the respective position.

Lifting Of Token Comparisons Our adaptions to the tainting engine make it possible to report so called *token comparisons*, which consist of two integer values, where one is tainted with one or more input character taints. To make use of those comparisons on an input character level, we need to lift the integer comparisons back to actual input value comparisons. For example, suppose we have the input `while(...`, a token comparison would look like `6 == 7` with the value 6 tainted with the input characters at index 0 to 4. Thus, our tainting engine already gives an important information: the value 6 used in the comparison belongs to the input characters `while`. Hence, we need to substitute `while` with the lexeme forming the token with internal value 7. **But:** *which lexeme produces this token*?

Let us have a broader look on the problem: if we just consider one comparison, we see nothing more than two integers and a few taints—we are missing the information what the untainted integer represents. What we get is that the token with value 6, with a high likelihood, is generated from the input characters `while`. We can store this information[6] and in subsequent runs use it to map back the token value 6 to the input characters, i.e. the lexeme, it represents. And if we do this often enough and with all the tokens the subject under test works with, we will get a holistic view regarding lexemes and the tokens they generate.

Summary With the presented long term mapping of all token values to the lexemes they represent we can apply the same methods that we used in Chapter 3. We can lift the token

[6] In the actual code the learning of the token to lexeme mapping is more complex to reduce the number of incorrectly learned mapped values. See Section 4.2.3 for more details.

values of parser comparisons to the original lexeme values, for the tainted and untainted tokens, making it possible to handle the parser comparisons very similar to parser comparisons without a lexing phase. Together with the improved tainting from Section 4.1.1 and 4.1.2 we are able to infer inputs with similar information as in the *"parser without lexer"* case.

4.1.4 Input Generation Loop

Sections 4.1.1 to 4.1.3 describe the measures taken to make it possible to infer inputs in a similar way as described in Chapter 3—this time in the presence of a lexer. This also interferes with the overall generation loop, as we now have to add some additional phases that take care of learning token information (like the token mapping as discussed in Section 4.1.3). Thus, this section gives an overview on the extended generation loop:

```
def input_generation_and_execution():
  inp = random(printable_characters)
  inp_rand = inp
  parent_subst = None
  frontier = PriorityQueue()
  tokens = Set()
  tokens_learned = Set()
  comparisons, is_valid = run_and_trace(inp)
  subst = learn_tokens_and_substitutions(tokens_learned, tokens,
   ↪  comparisons)
  while recently_covered_new_code():
    if is_valid:
      report(inp_rand)
    add_substitutions_for_input(frontier, subst)
    if not tokens.empty():
      inp_rand = tokens.pop()
      tokens_learned.add(inp_rand)
      parent_subst = None
    else:
      parent_subst = frontier.pop()
      inp = mutate(parent_subst)
      inp_rand = inp + add_random_extension()
    if has_rand_extension():
      comparisons, is_valid = run_and_trace(inp)
      if had_return_code_zero():
        frontier.add(parent_subst)
        subst = learn_tokens_and_substitutions(tokens_learned,
         ↪  tokens)
        if covered_new_code():
          inp_rand = inp
          continue
    comparisons, is_valid = run_and_trace(inp_rand)
    subst = learn_tokens_and_substitutions(tokens_learned,
     ↪  tokens)
  # report tool output
```

```
33
34 def learn_tokens_and_substitutions(tokens_learned, tokens,
   ↪  comparisons):
35   for comp in comparisons:
36     if comp.constant not in tokens_learned:
37       tokens.add(comp.constant)
38   learn_token_mapping(comparisons)
39   return learn_substitutions(comparisons)
```

Loop Overview The overall generation loop is similar to the generation loop presented in Section 3.1—though we needed to adapt the representation for our data structures (like the frontier) in this example to make the workflow clearer. We still start with a random character, run the program, analyze the trace, store inputs and possible mutations (called substitutions) in a priority queue (the frontier) based on their heuristic value, extract the most promising substitution, mutate it, and restart the inference loop. In the following we detail the needed adaptions made to improve the input inference in the presence of a lexer in the program.

Token Learning In Section 4.1.3 we discussed the learning of a token to input character mapping. This mapping is learned with the call to `learn_token_mapping()` in Line 38. In order to successfully learn new tokens, it makes sense to analyze the program execution on those tokens in isolation (there will be less noise in the trace if only one lexeme is given to the subject under test). Thus, in Line 6 and Line 7 we introduce two new sets: the *tokens that we still need to learn* and the *tokens we already executed in isolation during the inference loop*. Concretely, whenever we see a constant in a comparison which was not already executed in isolation, we remember it (Line 37).[7] Now, in contrast to the original execution loop, we prefer to gather information about tokens, i.e. we check if there is any token still open to learn and if so we let this value run in isolation on the subject under test (Line 14). This way, we get large portions of the token mapping early in the inference execution, which is important as this information is needed to analyze parser comparisons appropriately (especially once we actually use the frontier and infer syntactically valid inputs).

Input Re-Running In Line 24 we check if the just run input without a random extension caused the subject under test to finish with exit code zero—meaning that it is likely valid. If so, we also add the substitution used to generate this input back to the frontier (Line 25), because in the upcoming lines it may happen that we cut the execution loop short (which may also happen in some other border cases, we will go into detail about this in Section 4.2). To avoid missing the comparisons generated by the random-appended input, we put the just used substitution back in the frontier, hence it can later be run again if no better comparison is found.

New Heuristics We already introduced heuristics in Section 3.2.2 for our basic approach without specific lexer analysis. With our basic approach we missed parser comparisons if a

[7] In fact, we also have some heuristics to add tokens which are generated using more than one comparison. For example, if a token for the greater or equal operator could be generated by first comparing against the greater character and then against the equals character (see Section 4.2.2).

lexer was present, thus the approach had *"blind spots"* in those situations; but then again had almost no noise in the comparisons if no lexer was present, as the reported parser comparisons were determined by direct information flows which involve close to no approximations. The parser comparisons though are just an approximation as we cannot guarantee that the token tainting is always correct in the dynamic tainting engine. Hence, our heuristics are adapted to handle noise in the comparisons—substitutions (as extracted in Line 39) are added to the frontier based on those adapted heuristics (Line 13). In Section 4.2.5 we will detail those new heuristics.

Random Appending In Line 22 we see that the loop flow is dependent on the presence of a random extension to the input. In some border-cases we do not append a random character or token, e.g. when running only on a token value (Line 14 and following). Also, in the presence of a lexer, a random character value is typically not a valid lexeme and would be rejected by the lexer. The random extension though is done to analyze parser comparisons at the very end of the input—everything in front of the extension should be syntactically valid. Thus, if the substitution is based on a parser comparison, we typically append one of the learned lexemes to the input (Line 21), which is likely accepted by the lexer and ends up as token in the parser.

Subsequent Tools After a certain amount of executions without exporting a valid input (e.g. more than one thousand; including all executions, traced or not) the input inference phase is considered to have reached a plateau and is stopped (Line 10). At this point we consider the inference to not be beneficial anymore as our technique is not able to cover new branches in the parser, thus likely not finding new features. Now it makes sense to collect gained information and give it to a subsequent tool like a fuzzer (Line 32), which can then make use of this information for efficient fuzzing (see Chapter 5). This information can include tokens found (i.e. the string contents of the token mapping as defined in Section 4.1.3) as well as found valid inputs. Certainly, any information from the execution could be exported to another tool chain, there might be other interesting data for other tools.

This concludes the token and valid input learning phase and summarizes the resulting output of the techniques described in Chapter 3 and this chapter. In the following we will detail how the token extension is implemented, and give more information about the technicalities involved. As in Section 3.2, the implementation itself must solve practical and technical problems that are not obvious on a theoretical level.

4.2 Tokenizer Analysis Implementation

In Section 4.1 we explained how, on a high level, we want to attach taints to tokens, which in turn let us analyze parser comparisons in the presence of a lexer. Still, bringing those theoretical constructs into a real world application results in additional problems that need to be solved. In this section we will have a look into the implementation details that make it possible to apply our token analysis to real world subjects. We will start with the taint generation done in the dynamic tainting engine, explain how we extract lexemes from the resulting tainting trace, infer the token to input character mapping, and explain how all this information is used

to generate possible mutations for the inputs and how we can put them into a prioritizing order. Finally, we have a look into the extensions needed for the input inference loop to handle tokens properly in the grand scheme, wrapping up and combining the different portions explained in the upcoming sections.

4.2.1 Token Taint Generation

The first step towards token analysis is the tainting of tokens. When solely relying on data flows during dynamic tainting, it is not possible to taint tokens as they are generated indirectly. Typically a token is generated based on a comparison and an assignment of a constant based on the outcome of the comparison. Thus, in the following we explain the different stages:

Lexer/Parser Differentiation We divide the code in lexing and parsing.

Token Taint Preparation We remember the input token taints in lexer comparisons for later token taint generation.

Token Taint Generation The remembered input token taints are attached to a newly generated token taint, which in turn is attached to a generated token.

Token Taint Reporting At some point the tainted tokens are used in comparisons in the parser, which is then reported by the tainting engine.

Token Store Cleaning We cannot store all remembered tokens throughout the whole execution, at some point we need to not consider them anymore to avoid over tainting.

Lexer/Parser Differentiation As already mentioned, if a certain set of LLVM bitcode instructions appears, the respective surrounding function is considered a lexing function. Such comparisons are a subset of the LLVM bitcode *binary instructions*[8] as well as the *SWITCH* instruction, but also function calls to specific functions like *strcmp()*, *strchr()*, and *isdigit()*. For the *binary instructions* we mark the code as lexing if a taint is stored for later token generation (see below). For all other comparisons we only check if a non-token taint is attached to an operand, but no further checks are done, because their semantics are less complex and those comparisons typically only happen in the lexer if a non-token taint is involved.[9] Also we might under-approximate—e.g. for *strchr()* we only mark the function as lexing if the search character is tainted, but not if the searched string is tainted[10]. The approximation just exists for noise reduction and needs not to be perfect.

[8] The LLVM bitcode binary instructions we consider here are: *ICMP*, *FCMP*, *ADD*, *FADD*, *SUB*, *FSUB*, *MUL*, *FMUL*, *UDIV*, *SDIV*, *FDIV*, *UREM*, *SREM*, *FREM*.

[9] Except for *strcmp()*, which also checks for operand equality to reduce noise by only considering successful comparisons which might later cause a token generation. Due to a programming mistake in our prototype, *strcmp()* misses the check whether the second operand is tainted if the first is not, but since in almost all cases the first operand should at least have a string constant taint, this mistake should not have any impact in reality.

[10] As already mentioned in Section 3.2.1, we report `strsearch` comparisons if the first character of the string is tainted with a non-string-constant taint. Those comparisons are used to mark lexing code in the mutation engine later as they are handled like normal character comparisons.

Token Taint Preparation For our approach we defined comparisons that *collect taints for later token generation.* Those comparisons are the start of a token generation pattern (the second part of the pattern is a possible assignment of a constant value to a variable or memory location which we will discuss later). We handle the instructions as follows:

Binary Instruction For the LLVM bitcode binary arithmetic and logical instructions we only consider those instructions that are either an *ICMP* with *i32*-typed operands (as those are likely character comparisons, other types are not used for characters), or contain at least one *i32*-typed constant as operand—this is in worst case an under approximation of all actual character comparisons, but filters out most unwanted comparisons. We always store a taint for *ICMP* instructions if at least one operand is tainted without a token taint[11], as those comparisons are typical for token generation: two characters are compared and a token can be generated.

Furthermore, for other binary instructions we approximate the taint collection for the case that one operand is a constant and the other is a tainted value which does not have a token taint and both values are equal, as there are cases in which an arithmetic expression is used to calculate if a token should be generated (e.g. instead of a comparison for equality one might just use a subtraction).[12]

SWITCH For the LLVM bitcode instruction *SWITCH* we just store the taint of the value used in the switch statement as this is a simple comparison. Also switch statements are quite common to generate tokens—a value is pattern matched to several options (the cases in the switch statement), and if one matches, a token is assigned to a variable or returned. Thus, since this instruction is very typical for token generation, we do not have any requirements for the value used for switching (it only needs to be tainted).

Strcmp() For the function *strcmp()* we filter for comparisons that are successful, i.e. both operands are equal, as we assume that a token would only be generated if the input values match. Now, if at least one operand is tainted and does not contain a string constant taint, we store the taints of the respective operand for later usage in the token taint generation step.[13]

[11] The actual implementation of our approach, lFuzzer, contains a bug not considering commutativity of the operators, we discuss its impact in Section 6.7. In detail, for *ICMP* instructions the prototype would not collect a taint if only the second operand is tainted without a token taint and both are not equal—in this case it would still make sense to collect the taint.

[12] Again, commutativity is not correctly considered in our prototype, the mentioned case is only implemented if the first operand is a constant and the second is tainted without a token taint (and both are equal).

[13] The prototype implementation checks if any operand has a string constant taint, which might result in skipped taint storing for string comparisons. Still, in most cases the characters of the compared string were already compared individually in the lexer (e.g. when checking for the end of a keyword), hence their taints are already stored and we do not miss any taints. Also, as already mentioned in the lexer detection paragraph: the taint check is only done for the first argument, not for the second. But, in most cases both arguments should be tainted (either with string constant taints or input taints), hence generally checking the first argument for taints is sufficient.

Token Taint Generation The taints stored as detailed above, i.e. the pending token taints, are queried whenever a token generation pattern is found—such a generation pattern is defined as follows:

Return Instruction For return instructions the token generation pattern is straightforward: whenever there is a token taint pending for token generation, we check if the returned value is an *i32* constant, and if so we mark the returned value with the pending token taint.

Store Instruction For a store instruction we have two cases to consider. In the most basic case an *i32* constant value is stored and a token taint is pending for token generation: as with the return instruction, we mark the stored value with the respective taint. In the second case, a value with a token taint was stored to a memory location. In this case we overwrite the stored value's token taint with the token taint pending from the last comparison. The reasoning for the overwriting of token taints is that we want to consider the token comparison closest to the token generation. Again, keep in mind that the token generation detection is not exact, but rather an approximation. Hence it might be that a token taint was already attached to the value with taints from other comparisons—the value is already tainted but now used after another comparison. At this point we have to decide which token taint to use for the stored value, hence we consider the latest taint. Apart from that, it could also be that the code actually reuses an already token tainted value. As we already know that this value is used to store a token, we consider it as well when being stored at another location and update the token taint to the currently pending token taint.

Binary Instruction For binary instructions, whenever the instruction is not a compare instruction and at least one *i32* constant is involved, we attach the token taint to the result of the instruction. In some cases a token is not generated from one concrete constant value, but calculated, sometimes based on dynamic values, sometimes based on constants that were created before the first token comparison was done. For example, in a parser we want to generate a token for each number from zero to nine. Now, after reading the number and having a lexeme comparison, the token generation is done by adding the integer value of the number to some constant value representing the base token. The resulting value would be the token belonging to the number which is used further in the parser. To track those cases we consider binary operations as described.

Token Taint Reporting Once we propagated the taints from the input comparisons over the token generation patterns to the respective token values, we also need to track and report when those token values are used—i.e. we want to report the parser comparisons:

SWITCH For switch instructions we just check if the value used for comparison contains a token taint. If so, we call our comparison reporting mechanism, reporting a token comparison for each value of the switch statement. As a switch statement compares a fixed and clearly defined set of values, it is a perfect pattern for token comparisons, hence

we have a very lax definition of when to report token comparisons here, just requiring a token taint in the compared value.

Binary Instruction For binary instructions we check if the instruction is an *ICMP* instruction, comparing two *i32* values, i.e. it checks if two integers are compared. We assume that typically tokens are stored as enums or integer values, hence the restriction to *i32* values. This reduces the noise while reporting token comparisons.

In both cases the reporting mechanism checks for some generic restrictions and ensures that they are fulfilled before reporting them:

Non Lexing Function Whenever a function on the stack is marked as lexing we do not report any comparison as a token comparison.[14] As we already mentioned, it is common to separate lexer and parser [3], hence a token comparison should not happen in the lexer code, but only in parser code.

Token Taint Present At least one of the comparison operands must have a taint that is marked as token taint. The reasoning here is simple: we only want to report comparisons as token comparisons if at least one value is involved that stems from a value generated in a token generation pattern—otherwise we would report many comparisons incorrectly flagged as token comparisons.

Exactly One Tainted For both operands we check if they are tainted. Tainted in this case means that a taint exists, is not empty (i.e. the taint references one or more taint sources), and does not include any *string constant taint*. We will get to string constant taints later, for now it is just important to know that those are taints not stemming from an input but from a global string constant in the program source. Only if exactly one value is tainted as defined above, we report the comparison as a token comparison. The reasoning for this is again noise filtering: a token comparison typically consists of a comparison between a token and a fixed value (a constant or some value calculated program internally). Thus, in our tainting engine the comparison should involve exactly one token tainted value and a value without taints.

If all of the above applies, a token comparison is reported, which means that we report the indices of the input characters involved in the generation process of the token as well as the integer value of both operands (as a token is nothing more than an integer value in our setting). The mutation engine later on uses this information to match the integer values back to the

[14] It might be that a function will be marked as lexing later during the taint analysis. Already seen token comparisons might then be already reported for such functions. As the token comparisons are an approximation anyway we live with this approximation for efficiency reasons in the tainting engine. The parts of the mutation engine (Section 4.2.3) handle lexing and parsing stages for complete runs and even throughout all runs to further reduce the impact of this approximation.

actual input characters, i.e. it builds a bi-directional mapping between token values and lexemes (Section 4.2.3).[15]

Token Store Cleaning As already mentioned in Section 4.1.2, we need to limit the lifetime of taints remembered for token taint generation (i.e. the taints we collected from the lexer comparisons). If not, we would highly over taint, as at some point we would have collected the taints from all input characters. As already said, the removal of those taints only happens if the taints were used beforehand, i.e. once a token comparison is done (once a token taint is reported as defined above). The usage information is reset every time a new taint is remembered for later token taint generation (as explained above). Thus, we define the cleaning points as follows:

Method Entry And Exit Whenever a *method is entered* or a *return instruction* is encountered, we clean the respective stored token. Methods are logical boundaries in the code, hence, once a token was used, it is very likely that the temporarily stored token from the comparison should not be used any further. The scenario is as follows: a token was generated in the lexer, given to the parser by returning from the lexer code. At this point the cleaning would be triggered but as we already explained above, cleaning only happens if the taint was used, i.e. a token comparison happened—hence no cleaning is done at this point. Now the parser performs a token comparison, and then the parsing method is left or another parsing or lexing method is called. This means the lexer finished and the parser either performs additional operations or requests a new token from the lexer. In any case, the temporary token from the lexer comparisons should have been consumed, hence we delete the taint, preparing the next lexing phase. The upcoming two cleaning sites are just fallbacks and are used for faster cleaning of taints to reduce noise. Typically, the method enter and exit locations should be sufficient for an effective token cleaning.

Binary Instruction We clean the taints if at least one of the operands is tainted without a token taint but the other requirements for preparing a token taint as described above are not fulfilled.[16]

Strcmp() For *strcmp()* we clean the taints if the comparison contains *non-equal* values of which at least one is tainted.[17] The reasoning for this is analogous to binary operations: for token

[15] We also have a comparison type called *"tokenstore"* which was intended to report the storage of a token to a memory location for other users of our tainting engine. This is typically not used in our mutation engine later, indeed those comparisons are actively ignored, i.e. in most cases when we talk about comparisons in this chapter we do not consider them—especially not when generating substitutions for new inputs.

[16] Remember the implementation error in the prototype ignoring commutativity for such binary instructions. This causes the cleaning of taints if the first operand is tainted (without a token taint) and the instruction is not an *ICMP* instruction. Also, we clean the taints if the first operand is either not tainted or a token tainted value and the second operand is tainted, but both operands are not equal. Hence, the approximation might not always be correct, but as we guard the cleaning with previous token usage and the cleaning at binary operations is just a fallback, the impact of this imprecision should be close to zero.

[17] In fact, the actual implementation used for our evaluation contains a small deviation from this description: cleaning the taints for *any strcmp()* with different operand values if the first value is not tainted—which should be very rare because of the string constant taints we use. This might cause an over approximation as already mentioned for the binary instruction part.

generation there should be a preceding successful comparison and no other comparison in between, as we assume a new lexer call in those cases.

String Constant Handling A lexer and parser heavily relies on the operation on strings. Thus, string constants from the code are certainly important to consider while analyzing those parts of the program. Therefore, we created a special taint type for string constants to track them and their derivative values throughout the execution, making it possible to analyze them in more detail. Those string constant taints are attached at the very beginning of the program execution, giving every string constant character a unique string constant taint. Those taints are only considered at specific positions in the code, in general they are explicitly ignored:

GetElementPointer In contrast to direct string comparisons, another way of looking up keywords is using hash-tables. Here, several keywords are hashed into an array and during execution this array is directly accessed.[18] Apparently the lexer wants at least one of the values from the array to proceed, so it makes sense to report a series of *strcmp()* operations to the mutation engine. This is especially important because for direct lookups like it is typically done in hash-tables, we would only see the one cell accessed and thus only the taints of this one element, hence we miss options of the lexer.

Now, for the base address of the array we can store the taints of the indices used to calculate the final address with the LLVM GetElementPointer instruction—if they are string constant taints. This gives us the information which string constants are stored in the array. If a value with a taint, which is not a string constant taint, is used as an index, we can assume that an input lexeme was used to calculate the index, hence a hash-table lookup happens. Thus, in this case we report the respective string constant values for the base address. This does only work if the base address of the GetElementPointer instruction is the address used for adding the string constants to the array—which is typically the case when accessing hash-tables.

String Constant Comparison In some cases, instead of comparing input characters with character constants or searching a character in a string with a library function, programmers iterate through a string constant and compare each character of the string with their search value. We report comparisons done this way as *string constant comparisons*, a comparison type that can be used by taint consumers like our mutation engine. Those comparisons are typically handled like normal comparisons in the mutation engine, but there are some exceptions when it comes to filtering those. We will later explain how and what comparisons are filtered before and during substitution extraction (Section 4.2.4).

4.2.2 Lexeme Extraction

In Section 4.2.1 we explained how token taints are created and reported by the dynamic tainting engine. Now before we can make use of those token taints during input inference, we need to extract information about lexemes from the stream of lexer comparisons. The explicit

[18] For cases in which strings are stored in an array sequentially and the array is iterated, we do not need a special handling as those strings will be extracted and used in comparisons.

knowledge about lexemes is important for the later input inference based on token comparisons. For example, in Section 4.2.3 we will explain how we combine lexemes with the token values they generate, which in turn makes it possible to infer input substitutions as explained in Section 4.2.4. In this section we:

1. clarify how we extract lexemes from *standard lexer comparisons*
2. detail how we handle *consecutive lexer comparisons* that read a lexeme character by character

Standard Lexer Comparisons As already described in the approach, the lexer needs to check for valid, known tokens. Hence, if we input a lexeme the lexer does not know, we would already see all the comparisons of our input value against the possible lexemes of the subject under test—the lexer needs to ensure that none of its options match. If, by chance, we input a valid lexeme right at the beginning, chances are high that later during our input inference loop another input will actually contain a value which is not known to the lexer. Thus, at some point we will observe all lexer options. The possible lexemes we collect are mostly the values not yet seen from the input character comparisons that we would also use for substitution when building a new input. For a switch statement we learn all case-values, for *strlen()* comparisons mock values of repeating *'a'*s up to the used string length of the comparison, and for conversions (*strtod()* and *strtoul()*) we add the conversion options. For comparisons with the operands $</\leq$ and $>/\geq$ the tainting engine would recommend to test anything that is smaller (or greater) than the compared value. To reduce load on the token learning phase we do only consider the value *closest but valid* to the compared value, i.e. the value that is smaller (or larger) or equal to the compared value. Typically the lexer would, if this was the only comparison, just accept any of the values from the given range to create a token, hence in most cases trying one is sufficient. For all other comparisons we just learn all options given by the tainting engine.

Consecutive Lexer Comparisons Also, we look for lexer patterns that consecutively lex a single lexeme to build a token. To detect those patterns we take all input character comparisons from one lexer function call and first check if there are more than two comparisons and no two consecutive comparisons are the same instruction in the LLVM bitcode. We also check that the starting index of the input characters used in the comparisons is strictly monotonic increasing, i.e. we actually lex a consecutive portion of the input, one character per comparison. With this, we approximate and handle the consecutive comparisons analogous to direct string comparisons, though here each character or even lexeme is compared individually instead of using a library function like *strcmp()*. For such cases we can still generate a lexeme consisting of one string instead of several single character "lexemes" which is used to improve the respective lexeme token mapping later on (i.e. we can give the combined lexemes as complete inputs to the subject under test instead of using the values from every single comparison). Once such a pattern is detected, we create the resulting possible lexemes as follows:

1. Start with set A containing the empty string.

2. Take the next comparison from the previously collected comparisons, iterate the stored strings from set A and for each append the used comparison values element wise to the stored string; add the new strings to set B.

3. The new string set B becomes set A.

4. Repeat at 2 until no more comparisons exist.

For example, say we have two comparisons, the first with the values *ab* and *cd*, the second with the values *e* and *f*. After the first iteration, we appended the values of comparison one to the empty string, resulting in the intermediate set containing *ab* and *cd*. Now, we append *e* and *f* from comparison two element-wise to those values, resulting in the new set with the values: *abe*, *cde*, *abf*, and *cdf*. This is the resulting set of strings that can be used for later token stack mapping learning (Section 4.2.3).

4.2.3 Token And Stack Mapping

As we have already seen in the introduction of Section 4.1.3, learning token values can be done by mapping the resulting token value back to the input characters it stems from. In our example, the **`while`**-keyword caused the lexer to output the value 6, hence we can learn that 6 maps to **`while`**. If we do this for all lexemes we encounter during our input inference loop, we are able to extract a mostly complete picture of tokens mapped to their originating input characters. Extracting this mapping is possible because even if a lexeme, and hence the respective token, is incorrect as input or as first lexeme in the input, the lexer will just generate the correct token for it (as it is not context aware), and the parser checks all possible options. Hence, the *token mapping inference* can happen without having the lexemes at a syntactically correct position and thus can be done before inferring larger inputs consisting of several lexemes. As a side product of the token mapping inference we also get a mapping from a call stack to comparisons done at this stack level. This information is relevant for our priority heuristic that we will explain in Section 4.2.5. Thus, this section will explain:

1. how we learn elements of the token mapping: *explicitly* and *during input inference*

2. how we *approximate the correct token value* for each lexeme

3. the *resulting mappings* based on the approximations done

4. how we handle the different *lexeme learning situations* that might reduce the token learning completeness and soundness

Lexeme Learning From the lexeme learning (Section 4.2.2) we have a list of possible lexemes. We can start learning their token values explicitly by only giving one possible lexeme to the subject under test instead of trying to learn them from inputs with several lexemes. Learning from inputs with only one lexeme makes it easier to map the generated token to the input characters that generated this token as there should only be one token value.

In general our input generation loop takes an input from the priority queue and runs the input on the subject under test to gain new insights (as described in Section 4.1.4). If we have a lexeme from the lexeme extraction not yet tried alone, we would just take this one lexeme as input. As a lexer not only performs comparisons without context, but also generates token values without context and gives them to the parser, we can just iterate the collected lexemes, give them to the subject under test without further context (i.e. we just input the lexeme and nothing else), and see what token values are resulting from them—in best case there is just one token generated and all input characters are consumed by the lexer. Even though, by design, this token learning is done before the main input inference begins, at any time, if any new lexeme is found, it is tried to be learned in the next iteration of our inference loop.

Furthermore, even though this phase is mostly used for token learning and we do not assume one lexeme to already represent a valid input, all other features of the mutation engine are still active because once the single lexeme is executed on the subject the resulting trace is handled as any other input trace. That means, if a lexeme already causes the parser to finish its execution with exit code zero, we consider this one lexeme as a valid input and report it. And even if the input is not valid, the possible substitutions resulting from the reported taint output are put into the priority queue and queried as soon as there is no single lexeme to learn anymore. This causes the side-effect that once the initial token learning phase is finished, we have a set of possible prefixes in our priority queue to start our input inference.

Token Value Approximation Now that we know how we collect lexemes for token learning and that we run the subject on tokens individually to extract and store token information we go into details on how the actual mapping information is extracted from an execution and how it is used. The procedure for extracting the information from a run with one lexeme is the same as for a run with several lexemes that build an input. We perform the following steps in order of appearance to extract information about generated and used tokens (including filtering techniques to reduce noise when performing the token extraction[19]):

Lexing Functions We already mentioned in Section 4.1.1 that functions that use input characters in comparisons and all transitively called functions of those are marked as lexing—a lexer typically does not call a parser but only vice versa. A similar technique is also used during token mapping learning to filter out all supposed to be parsing comparisons that happen in functions marked as lexing. In contrast to the lexing pruning in the tainting engine stage which is called per subject under test execution, we can combine the information from several executions for filtering lexing and parsing in this stage. Hence, we use a different definition for lexing and parsing functions: a function is marked as lexing if there are more lexing than parsing comparisons. The counting of lexing and parsing comparisons in a function is done only the first time it is encountered for efficiency reasons.

[19] The mentioned filtering techniques are mostly just used for extracting token information, they are in general not actively used during the actual inference of substitutions (Section 4.2.4) for filtering any comparisons—though, in the greater scheme, they might have some influence on the generated substitutions.

A comparison is a lexing comparison if it is not a token comparison and a parsing comparison if it is a token comparison. Now, a function is either marked as lexing, parsing, or did not yet receive a stage definition.

First, we collect for any function that has no stage yet the number of lexing and parsing comparisons. Then, we start with the call-tree root as the initial function and set the initial stage to undefined. We check if the function was already checked and either has no stage assigned or the parent stage (the stage of the caller) is larger or equal to its stage (*whereas the lexer stage is smaller than the parser stage which is smaller than undefined*)—if so we just return; otherwise we do the following. We set the current stage to the parent stage, then we try to assign a stage to the function. For this, we check if there was at least one comparison collected for the function.[20] If so, we mark it as lexing if there are more lexing than parsing comparisons or the parent is marked as lexing—otherwise, we mark it as parsing. Also, we set the current stage to the marked stage. If the function had no collected comparisons, we set the current stage to the already stored marked stage if present. Then we repeat this for all callees of the function. The used call-tree is updated based on the current execution each time a new function with comparisons is encountered during execution and the lexing-parsing mapping is updated.

All token comparisons happening in a lexing function are pruned immediately and not used in the following remaining pruning steps. If a function is marked as lexing in a later run though, the generated, persistent information from this section is not updated—only newer values and overwritten values will be affected by this change in the lexing-parsing status.

Doubled Comparisons We also do not add those token comparisons which appear twice or more directly one after another, i.e. only the first comparison is added if one or more equal comparisons are encountered in a row without any other comparison in between. Two comparisons are equal if they have the *same starting input index*, have the *same tainted value*, the *same comparison value*, the *same call-stack* and the *same comparison id* (i.e. code location). This is done to filter out looped comparisons as those are likely not actual token comparisons as in general there is no reason to compare the same token against the same value several times at the same code and call-stack position. Chances are high that the used value was falsely tainted with a token taint and is now used in some looped comparison. As we will later have a *"majority vote"* on the found comparisons to define for each index the most likely correct token, it makes sense to filter out such looped comparisons beforehand. Leaving them in, even though they are likely incorrect, might cause the majority vote to select the token value from the loop instead of another option that has higher chances to be correct. Hence, we count the looped comparisons as one comparison.

First Index Correction In some cases the index of the first comparison made (after filtering all comparisons in lexing functions) may not be the minimal index seen in this execution

[20] There is one mentionable border case here: if the function solely contains *tokenstore* events (which are typically ignored by our algorithm), it would be considered as if comparisons were collected for it, but the number of lexing and parsing comparisons would be zero.

which should not happen as the parser consumes tokens from left to right. Thus, to have a consistent starting index, the index of the first comparison is always set to the minimal index value seen in this execution, typically the starting index of the input (in most cases it already was this value).

Majority Vote With the majority vote we extract for each index the one token value that appeared most in comparisons in the current trace. We assume this value to be the correct token value generated based on the lexeme starting at the respective index in the input.

For this we create a mapping from each starting index of each comparison to the token values used in the comparisons, giving us for each index the token values used and for each such token value the number of times it appeared for the given index. Hereby, the filtering of sequential equal comparisons as mentioned beforehand reduces the number of found token value appearances for the respective index and token value of those comparisons. Also, if the first index was corrected in the step beforehand this corrected comparison will be added as first and only comparison for the corrected index here. Now we can create a mapping from each index to the token that was used the most (if for two or more token values the number of usages is on par one is taken at random). We delete all comparisons which do not have the majority token value for the index they use. This makes sure that we only have those comparisons that are likely the token comparisons stemming from the token value that was generated from the lexeme starting at that index.

Also we delete all backtracking comparisons, i.e. if there is a token comparison later in the execution which uses an index smaller than a non-filtered token comparison beforehand, it will be deleted. The reasoning for this is simple: the parser works from left to right on the input, hence the index should monotonically increase during parsing while consuming the tokens. Now we have a *reduced list of comparisons* which contains only those comparisons for which the *tainted value equals the majority token value at the respective index*. Also the *indices in the list are monotonically increasing*.

Subsuming Token Ranges After the majority vote we already have a list of comparisons with a monotonic increasing start index value and only one value per index. In this part we try to find token ranges and filter for the largest ones, i.e. we try to *extract those comparisons that belong to a token that will result in the largest part of the input consumed—assuming greedy lexing/parsing*.

The calculation of ranges is started by iterating all comparisons not yet filtered in order of appearance during the run. If the index of the next comparison is higher compared to the current the following information of the current comparison is stored (if not already present, which should not happen because of the monotonically increasing indexes from the majority vote filtering beforehand): *(starting index, difference to next index, comparison value, stack, comparison id)*. We filter for the last comparison of the index position, all comparisons with the same index do have the same token value due to the majority vote and are thus filtered. Due to the first index correction it might happen that at this position the corrected index comparison is added as a first range starting at the corrected index up to the index of the next comparison.

We always take the last comparison that remained after the majority vote and add a range as described above, but take the maximal index plus one found in any "real" comparison (a comparison that is not a token comparison, *strlen()*, or *eof* comparison) as next index value for the difference calculation. We only append this range to the list of ranges if it does not already exist This is done because the last comparison does not have a counterpart, i.e. we only have a starting index but there is no comparison "ending" the token range, hence we take the rest of the input up to the next real comparison as end index. This gives us an additional token-lexeme mapping to one of the last compared characters, likely representing the full token used in the token comparison.

The addition of the value 1 for the length calculation is needed as the last compared character might still be part of the just created lexeme, e.g. if we have an input *"abc"*, the parser lexes up until index 2 which is the character *c*, creates a token and the respective comparison ends up in this range calculation with starting index 0. In this case the length needs to be three, which is exactly $2 - 0 + 1$: maximal index of a lexer comparison minus starting index of the token comparison plus one.[21]

Now, we filter for subsuming ranges. Therefore, we iterate the tuple list in reversed order and compare for each element the starting and ending index with all other ranges. If a range is fully subsumed by our iterator object (start and end index must lie in the range of the iterator object), the respective value is marked for deletion. Once we iterated all values, the marked values are deleted from the list. In general, due to the previous filtering steps it should not happen that any range is pruned in this step (except for the newly generated ranges, i.e. if the maximal index is extended), still if any of the previous filtering steps would be removed this is needed to clean up the token list as it does not make sense to have a token "within" another token, lexemes cannot subsume each other.

Resulting Mappings This final list of ranges is then used to extract information—some for this specific run, some for all runs merged together. In the following we explain which information is stored and sketch how it is used later, details follow in the upcoming sections:

Stack To Comparison Mapping For each stack-tuple we map the *index*, *token value* and *comparison-id* seen for it after filtering. Hence, especially the filtering for subsuming token ranges and the majority vote may drastically reduce the number of added information. In fact, for each index only one tuple is added, i.e. there is only one comparison added per lexeme. The first index correction might cause an additional element to be added for the resulting mapping—i.e. the first comparison seen defines its own range starting at

[21] One might ask if the starting index correction influences this token ranges calculation. The answer is: partially. If there was a starting index correction (the index of the first comparison was higher than the minimal index), and there is more than one comparison after the majority vote, the index corrected token comparison would be the first in the list, all other comparisons come after this one and have a higher index in our range list. Hence the corrected comparison builds its own range. Also, the added last comparison is only influenced by the corrected comparison if the corrected comparison was the only comparison in the token comparison list. When entering the majority vote it would now get the length assigned based on the maximal value instead of using any subsequent token comparison for the calculation, resulting in assigning the range for the minimal and maximal index to this specific token value solely based on calculations rather than actual comparisons.

the minimal index and thus possibly gets its own element mapped to the stack. Also, if there was only one comparison, the index correction shifts the index of the comparison back to the minimal index. **This information is run specific and gets generated for each run individually without storing any information over several runs.** The stack-comparison mapping is later used in heuristics to determine the positioning of newly created input mutants in the priority queue and for filtering some mutations before generating new inputs based on token comparisons.

Token Mapping The token mapping is a mapping from token values (which are integer values) to concrete strings—collected and updated over several runs. In some cases the token value can be generated by different lexemes, e.g. in cases for which the token is generated from a dynamic range of inputs like for variable names or digits. In such cases one token value is mapped to several possible strings, in other cases the token value is very strictly bound to one or a few specific strings, e.g. for keywords.

This mapping is filled with two different confidence levels, depending on where the token value comes from. If (after all filtering above is done) the lexeme which created the token is not the full input string, we mark the string with a low confidence. We assume that due to imprecisions while attaching taints to the token value it could happen that taints from other parts of the input are assigned to this string and the lexeme stored for the token value contains *more* or *less* characters than the actual lexeme that would generate the token. Though, if the full input generated the token value, we assign a high confidence to the string, as the mentioned imprecisions are much less likely to happen here. Remember that we run the subject on single lexemes if we never tested the lexeme on its own beforehand, hence chances are high that we find for most token values one or more high confidence strings. Obviously, the index corrections as mentioned above may influence the stored lexemes here—e.g. we might add a lexeme based on the corrected first starting index and/or based on the corrected size of the last added range. As this mapping is an approximation, it is fine to also include such approximated tokens to lexeme mappings.

We define which values are actually stored in the mapping by preferring values with a high confidence for a given token value. In detail: if for a token value we only have low confidence values stored (including no value at all), and we are about to add a low confidence value, we first check if we find a value in the already stored strings that has a common prefix with the value we want to add. If we find a non-empty common prefix, we replace the stored value with the common prefix, stop, and do not add the low confidence value. With this, we can reduce the number of low confidence values in the mapping to a common ground—this makes it possible to filter out noise introduced by lookaheads and incorrect extensions of actual lexemes with other characters from the input.

If we encounter a high confidence string for a token value, then all low confidence values are removed from the mapping of this token value and the new high confidence value is added. Also, only high confidence values can be added to a token which already maps to high confidence values. **Summarized:** if we found at least one high confidence string during execution for a token value, this token value is mapped to only high confidence

strings. If we never found any high confidence strings for a token value it may also be mapped to low confidence strings.

Lexeme Learning Situations Above we talked about a perfect world in which the token learning phase works as we assume. Still, there might be cases in which this is not true. In the following we talk about problems that may arise while extracting lexemes, but also problems that make it harder to learn the actual token mappings.

Lexer Does Not Check All Lexemes While typically a lexer is not context aware and as such would, for any input, check all options it has up until either one option matches or all options are exhausted and it errors out, this might not always be the case. One example could be a check for the string length of the input, guarding a keyword lexing phase for efficiency reasons. In such a case, instead of performing a *strcmp()* operation on all keywords the parser knows, it would first check the length of the given lexeme and then only compare the input characters to the keywords with same length. Such cases are either caught by just letting the mutator run long enough, eventually generating lexemes in many different lengths, or with some generic lexer and parser specific mutations, solving common problems that arise (e.g. for *strlen()* comparisons we generate inputs with a supposedly correct length, making it possible to overcome such roadblocks).

Lexer Uses Context Similar to the previous point, it might happen that a lexer is not called in a completely context agnostic fashion, but depending on the current parsing stage. While such an implementation is atypical, one could imagine a parser with two different lexers handling different lexemes, e.g. for efficiency reasons. In such a case we might, during explicit token learning, only see one of the lexers, because we only input one lexeme and the parser would be in the initial parsing stage. The mapping for lexemes from the other lexer could be learned in larger inputs which contain the lexemes in their respective context. We do not consider such a case in our approach—in most cases there is just one lexer reading lexemes without further context [3].

The Tainting Engine Wrongly Approximates Tokens As we have already mentioned in Section 4.1.2, attaching taints to tokens without noise is not possible, hence there will be noise in the token comparisons. This noise manifests in different token comparisons in the trace with different token values for the same input characters (or an input character contained in different taint sets). Concretely, it can happen that for our `while` example we not only have a comparison with the token value 6, but also comparisons with 5, 10, ... In the other case we might have a token comparison with taint indices 0 to 3 and a comparison with taint indices 0 to 4—which we assume to be noise, as lexemes should not intersect with each other. To mitigate such noise we use the techniques described in this section as well as Section 4.2.1 and Section 4.2.2 to detect, prevent, and filter such wrongful token taint attachments.

No One To One Lexeme Token Mapping In many cases a token value belongs to exactly one lexeme and vice versa, e.g. in a programming language parser the `while` lexeme would always produce the same token (in our example the token with the value 6), and

this token would only be produced by the respective keyword (in our example `while`). It typically holds that a lexeme produces the same token, but even if the same lexeme produces different token values, there is no need to special handle this for our substitution, we can still map the token value to a string that can generate the value. Much more common is the case that several lexemes produce the same token value. In some cases there is no need to differentiate between two keywords, as they represent the same syntactical meaning and can be handled the same during parsing. More often though there is just a set of values representing the same token, e.g. a variable name can consist of many different characters or a number consists of a string of digits. In such cases the exact value of the lexeme is not interesting, only the fact that a valid variable name or number was read by the lexer. Thus, our mapping does not only allow one to one mappings, but each token value is mapped to all the different actual input values that produce this value.

4.2.4 Substitution Extraction

Once we learned lexemes and their respective token value, we can make use of this information in the substitution extraction. While the general approach from Chapter 3 is still valid—*we replace portions of the input with constant values from comparisons in which they are used*—there are still some implementation details to discuss. In this section we explain:

1. which *comparison filtering* techniques are used
2. how we perform *token comparison de-noising*
3. how we infer *token comparison substitutions*
4. why we need to specifically handle *valid token comparisons* and how we add *random token comparisons*
5. how we *collect the substitutions* based on the extracted comparisons.

Comparison Filtering Not all comparisons of an execution are of interest for us, hence we apply some filtering techniques to reduce the overall search space during input inference to those comparisons that are most relevant. Concretely, we first find the last comparison (in order of execution) which is not approximated or just exists for additional information[22]. Then we extract the first index from the index list reported for this comparison (which is typically the index of the first input character involved in the comparison), which is defined as the maximum index for this run. This gives us a sufficiently precise approximation of the first index of the last consumed portion of the input, so we assume it to be the start of the last consumed lexeme. Now we filter for all comparisons (including approximated and additional information comparisons)

[22]For the approximated types the tainting engine has no direct data-flow but applies some heuristics to produce the comparison. Comparisons for further analysis are not considered as actual character comparisons most of the time but are just used in the mutation engine to improve the substitution generation. For reference, those approximated or additional comparison types are: *eof*, *strlen*, *strconstcmp* (comparisons against string constants as defined in Section 4.2.1), *tokenstore*, *tokencomp* (a token comparison), or *assert*.

that fulfill one of the following conditions—the result is a set of comparisons that either directly use the last input character as operand or could possibly use it:

Standard Comparison This is the most basic case, the comparisons directly use the starting point of the last compared lexeme. This filtering also excludes comparisons that do not explicitly use the last index, including those comparisons that use indices that are higher than the last index. This filters for some backtracking events, like if a keyword is lexed which might involve first checking if several single characters are within a character range and only then comparing the resulting substring for the different keywords. We would filter those single checks and reduce noise.

String Length Comparison We include string length comparisons that would involve the last index, i.e. the expected length of the string starting at the tainted index would have included the character at the last index. This makes sure that even if the length of a smaller portion of the input, not involving the last index, was used for string length calculation, we still consider the comparison as if the last index would have been included. We add those comparisons because the comparison indicates that we need a substitution that replaces the remainder of the original input to fulfill the condition, which we simulate by adding the respective string length comparison.

String Comparison We add string comparisons for which the first reported index of the comparison plus the length of the used comparison value would include the last index. This makes sure that even if the comparison started before the last index and only checked a portion of the input not involving the last index, if the compared string is large enough to reach to the last index, this comparison is included in the substitution generation later on.

Token Comparison We explicitly include token comparisons that come after the last compared character since the token comparison is just an approximation and taint values might be missing (e.g. only the taint of the last used character for the token generation is included in the token comparison).

Token Comparison Denoising Additionally, we have some token specific filters to reduce noise (we do not use any other comparison pruning techniques like the ones described in Section 3.2.2). First, we only consider those token values, which appear the most often for a specific index—we reuse the *majority vote* as explained in Section 4.2.3. Also, we only consider comparisons that are not part of lexing code (as defined in Section 4.2.3). If a function is later marked as lexing, we do not delete the substitutions that are already generated based on comparisons from that function—still, no new comparisons from that function will be considered. Additionally, we filter for **looped comparisons**: this pruning technique checks if a token value appears several times on the same stack for different starting indices or different comparison ID's (after the filtering as described in Section 4.2.3 of the token information extraction is done, i.e. only the comparisons from the stack mapping are taken). If so, the respective comparison is not considered and no substitutions are returned that would be added to the priority queue.

With this, we can filter out recurring token comparisons, i.e. we reduce redundancy in the input and thus make the input inference more efficient.

Due to our token comparison filtering the set of comparisons at each stack level only contains the last comparison done for each index, which contained the most used token value for that index. This is, for valid tokens, in general the comparison that consumed the token value. Thus, if this token value appears several times for a given stack, it must appear for different indices. Hence, with a very high likelihood, the same feature is parsed more than once for the current input, which means we likely created a redundant feature (the same feature appears twice in a similar context). And in such a case we want to keep the priority queue small and prune the resulting mutations, as they were likely already tried in earlier steps and with a shorter input.

For example: say we are parsing an instruction list of a programming language and we already generated a first instruction "`a = 1;`". Now we keep adding lexemes to the input, resulting in a larger input "`a = b; -`". The character "`-`" though creates a token which, at some point, will be checked by the comparison that already consumed the tokens for the lexemes for "a" and "b", resulting in a call to the same parser code that consumes assignments. Now, since "a" and "b" are valid tokens the comparisons that consumed their tokens are in the set of comparisons at the respective stack—the token value for variables is used in more than one comparison at that stack. Hence, we filter out comparisons which would consume a variable token at the same stack as there are already enough of that kind and we want to avoid redundant features—instead we can concentrate on non-filtered comparisons that cover other features (like adding an if-statement which would be handled by another function and thus lies in another stack). It might well be that this pruning technique also filters out needed comparisons, but we assume those cases to be rare and thus our approximation to be beneficial.

Above we talked about "flat" redundancies, i.e. the same comparison (and thus the same feature) appears on the same stack level, but a redundancy could be nested (e.g. *a while loop in a while loop*). While it might be interesting to also filter out those cases it is much harder to define redundant elements over different stack levels. If the stack is different, even if the top of the stack and a comparison appears several times, it might well be that the respective element is needed to close a valid input (at least without backtracking). Say we already have "`if (b < c) {}; while (b`" as prefix, then "b" appears a second time as a boolean operator, possibly being parsed in a method that parses boolean expressions. If we filter this it could happen that we cannot close the while expression, even though we would cover the while feature which is not part of the input yet. Hence, it is often needed to use the same features several times in different contexts—in this case "b" is parsed once in the "`if`" context and once in the "`while`" context, having different functions on the stack. Hence, our filtering algorithm restricts itself to the same parsing context, denoted by the same stack, which reduces the risk to prune needed recurring features.

Token Comparison Substitution Besides the substitutions of lexer comparisons as explained in Section 3.2.2, we now also apply substitutions for token comparisons. For them, we randomly

select at most five lexeme strings as mapped to the token value (as defined at the end of Section 4.2.3)—an arbitrary but fixed number, less lexemes could speed up the input generation process by giving less options, more lexemes have the benefit that more input options are tested and the chances of missing any input feature is lower. As we already have a mapping of token values to their respective lexemes that can produce them, we can do the following:

1. Check which input character taints are used in the comparison, i.e. which characters produced the token used in this comparison—those are the characters which will be replaced.
2. To determine the characters to use for the substitution, we query the token mapping and see which lexemes produce this token. If the number is greater equal 5, we select 5 of those randomly each time we determine the possible substitutions.
3. Then we can construct the possible substitutions as before, though we correct the index of the replacement position to the maximum index as determined by the comparison filtering if the index in the substitution is higher than the maximal index found. The maximum index marks the starting position of the last lexeme lexed during execution, hence any comparison with a higher first index does not make sense. Thus, we use the correction as a safety measure to correct the approximated token comparison indices if they are seemingly wrong.

Special Handling: Valid Token Comparison Substitutions based on token comparisons, like all other substitutions, are added to the priority queue based on their heuristic value and then queried for the next run. Before we get to the calculation of this heuristic value, we need to handle one special case: if we saw a successful token comparison, i.e. a token comparison for which the non-tainted operand is equal to the majority voted token value, we construct a different set of possible mutations.

During processing of the comparisons, we collect all right hand-sides, hence all constant token values seen in any maximum index token comparison. Now if we have a successful comparison we check if there are token values that we have already learned but were not seen in any comparison on the right hand-side—those are parser comparisons that might have been hidden by the successful token comparison. For this, we use the token comparison map as defined in Section 4.2.3. *Or in other words*: the parser likely found a valid token and thus stopped parsing the lexeme we are currently looking at and went on to the next lexeme. In that case we choose a random other token value not yet seen and use one lexeme that builds this token value as substitution for that index based on the selected token value. Also, as we use this new substitution for re-running the input to gather more information about possible substitutions, **we do not append a random character when instantiating a new input with this mutation** (in contrast to other substitutions which are typically accompanied with random additions; see Section 4.1.4). We use the information from the last matching token comparison for later heuristic calculation, but we create a mock call-stack with one dummy element to get a lower stack size and therefore a higher rank in the priority queue as we can later see in Section 4.2.5.

The question is: why do we need to special handle the case of a found valid token and construct a mutation set with a token that was not even seen on the right hand-side during parsing? Would this not end in a parsing error? The answer is: *yes*, we actually do produce a likely invalid new input, but this is the very purpose of this feature. The reasoning is based on a very important difference between parsers with a lexer and those without. If a lexer is involved we have a two stage validation: the subject under test first checks if a lexeme is valid in general, i.e. if a token can be produced, and only if this is the case, the parser verifies if the token is valid at the observed position.

For our approach it is important to see all options the subject under test allows for a certain position in a specific input, i.e. if we have the input `a = b`, what lexemes are allowed after `b`? Without a lexer (as in Chapter 3) we can just append another random character, collect the comparisons done on the character and with a very high chance the character is already invalid and the parser reveals the options it has at this position (and even if we guessed a character correctly, there are likely other values in the priority queue which will end up in a similar situation and we reveal the options then). With a lexer though, if we just append a random character, with a very high likelihood the character is not a valid lexeme, does not produce a token and the parser would not even be called.

In the presence of a lexer, the possible continuations of a prefix are not defined by the character comparisons, but by the token comparisons. Hence by excluding all token values the parser already checked and picking a random token, which was not yet used during parsing, we can construct an input, which gives us more information about the possible substitutions. As we replace the seemingly valid lexeme with a lexeme that produces a token value not seen beforehand, we see those parser comparisons that are done after the initially valid token. Either the new token was incorrect, then we can construct our mutation set as normal, or the new token was correct as well. Then it is either the case that any token is valid at that position, hence we do not have any other token to test and continue as in the incorrect case, or we append another token not yet seen in the comparisons and repeat until one of the other two cases occurs. Without this extension we might miss features in certain subjects under test, especially in cases for which a random identifier and a keyword are valid extensions of a prefix, as chances are high to append a random character which produces an identifier, but we might never produce a string that resembles a valid keyword if the parser first checks for the identifier token and only then for the keyword tokens.

Special Handling: Random Token Values Our mutator is based on the assumption that the lexer checks for certain values and if a value matches, the respective token is given to the parser. The constant values from those comparisons against input values are taken and used as substitutions during input mutation. As already said in Section 3.2.2, some parser comparisons (or in this case token generations), might be implicit. A lexeme or token is accepted either because no other option matches or because the comparison checks the opposite of the wanted values (is the value outside of a range instead of inside a range). In those cases we would never see the actual wanted comparisons. Thus we always add two arbitrary but fixed mutations

(marked as stemming from non-token comparisons), one with a number and one with a letter, replacing everything from the maximum index on. In Section 4.2.5 we will see how those substitutions are ranked compared to other substitutions in our priority queue. Similarly, if we do not see any mutations based on token comparisons, we append the same fixed mutations, but mark them as stemming from token comparisons instead which influences the heuristic value. The reasoning for this is that especially the lexing of such dynamic values like strings (e.g. for variable or function names) and numbers can be implemented in many different ways and it may well be that we simply missed the token generation part for those tokens. The mutator can use those random mutations to try if there is a "hidden" requirement for a number or a letter.

For this to work it is important to know that typically no input will be tried twice: if we already generated a certain input, we will not run the subject again on the same input. Hence, we can just add those additional mutations to fill common blind spots without risking duplicates in the search space that will be executed on the subject under test. This is also the reason why we add arbitrary but **fixed** numbers and letters. If we would add random values here, we would actually generate inputs which often cannot be filtered by our duplicate detection. Furthermore, with those additionally generated substitutions we add some random noise to the overall generation loop, making it possible to detect comparisons and code locations that might have gone missing when only using tainting information and the resulting substitutions.

Substitution Collection Once all of the above is done (and we did not perform the special handling of valid token comparisons), we collect the respective token comparisons as well as all other comparisons containing the last input taint index and store them into a set of possible substitutions. In the next paragraph we detail how we organize those substitutions in such a way that we prioritize those inputs that are most promising in either revealing new code or adding suffixes to already valid prefixes which cover new code, yielding valid inputs that cover new portions of the subject under test's code.

4.2.5 Specific Heuristics

In the previous section we explained how we extract substitutions from a taint stream. In this section we will go into details how we calculate the heuristics for putting the substitutions into a priority queue, such that we can later retrieve the most promising inputs first. Concretely, we will discuss the following:

1. **Valid inputs:** the heuristic relies on information based on *"valid inputs"*, hence we define which inputs we consider as valid.

2. **Coverage:** the heuristic uses *coverage information*, thus we also define how we calculate coverage.

3. **Code path:** we also need to explain our *path definition* for code paths through the subject under test.

4. **Heuristic details:** we give a short *overview on the heuristic* and then explain *each part in more detail.*
5. **Special substitution handling:** at some point in the input inference loop it happens that non-standard substitutions are added. We explain how their heuristics compare to standard inputs.

Valid Input Definition Valid inputs and runtime information from them are a central part of the heuristic, hence we describe what we define as valid and under which conditions we store a valid input. Also, we clarify when the coverage of an executed input is stored and remembered as *"covered by a valid input"*. Internally, during the execution of the input learning, we remember those inputs as *valid inputs* that we assume to be syntactically valid. In general, this definition is fulfilled by inputs that make the program finish with *exit code zero* or *let the program timeout*. In some cases we also check if an assertion was triggered, as we assume assertions to only happen after the syntactical phase. Checking for an assertion is only done for inputs that are executed under tracing, which is not always the case (for efficiency reasons not all inputs are run on an instrumented version of the program; in Section 4.2.6 we detail when an execution is instrumented).

In the following we detail the different storing and exporting cases for inputs that fulfill our definition of syntactically valid:

Valid Input Coverage Addition Any input that causes an *exit code zero*, *a timeout*, *or triggers an assertion* causes the coverage map (as defined below) to be re-calculated *if new code is covered*. The reason for using exit code zero is straightforward, as this typically means that the program exited without any errors, so we assume the input to be valid—*the covered code belongs to a valid path through the parser*. A timeout in most cases means the parser accepted the input but some subsequent program logic did not finish in time—most parsers run fast and typically do not diverge. We believe this approximation to be precise enough to consider such timeout inputs as valid as well—*the covered code likely belongs to a valid path through the parser*. The additional inclusion of inputs that trigger an assertion is due to the fact that such inputs are likely syntactically correct (we expect assertions to happen in the program logic, not in the parsing stage) but there is some uncertainty about their correctness—thus again: *the covered code likely belongs to a valid path through the parser*. This is especially important as inputs that cover new code get a high heuristic ranking, thus for cases in which an input not only covers new code but also triggers an assertion, we might end up building many similar inputs triggering the same assertion if we do not ignore the coverage to the assertion in future runs. Respectively, we store such code as already covered and thus possibly lead the generation loop to other features not yet tested instead of having a possible false focus on the assertion triggering input portion. Also, having a valid input that covers new code also triggers the update of the priority queue, as we have to re-calculate the heuristic value for each input in the queue.

Storing Valid Inputs We use an internal storage of inputs that we consider as valid. It is used at different places for calculations. An input is stored as valid if it either caused the

program to *finish with exit code zero* or *timeout*. The inclusion of inputs that cause an exit code zero and timeouts is reasoned analogous to the coverage addition above. Also, if *no* new code is covered, the input is treated as valid even if an assertion was triggered and the program neither stopped with exit code zero nor ran into a timeout. Here, we can assume the input to be syntactically correct and likely just a semantic check fails, hence we use this approximation to reduce the chance of generating similar inputs and run again in the assertion (similar to the argumentation for coverage storing).

Export An *exit code zero* or *timeout* causes an export of the input as valid input, for the same reasons as we store inputs internally as valid, but when exporting we only export inputs as valid if they cover new code no other valid input beforehand covered (see below for details about the coverage calculation). This is a filtering mechanism to reduce the output of our approach to those inputs that are syntactically interesting as they cover new features, it is not functionally important with regard to the input learning loop. At this point we also reset our counter, which counts the number of executions since no new coverage was found with any valid input. This is especially interesting to decide when to stop our approach and switch to another approach—more on this in Chapter 5.

Coverage Calculation A central part of the heuristics used to prioritize inputs is code coverage, hence we explain how we calculate coverage (which is mostly based on branch coverage). The tainting engine does not only report taints but also coverage events, i.e. whenever a new basic block is entered a tuple with the old and the new basic block identification number is reported in the stream of comparisons and other events—*but only the first time the tuple appears, there are no duplicates.* For coverage calculation we filter this stream of coverage events for those branches that happen in parsing functions at the time this calculation is done (including the current execution). With this information we create two coverage sets.

The first coverage set is the **all covered** set—which is used to calculate the **coverage of valid inputs**. For every branch we remember how many valid inputs covered it (for valid inputs as defined above). For the second coverage set we filter further: we only consider the coverage events up to the last comparison on the second-last lexeme—approximately resulting in the set of covered branches in parsing functions **until the last comparison of the last successfully parsed lexeme** (excluding some comparison types like *strlen()*, those are approximated comparisons and thus might be incorrect). This set not only excludes non-parsing code, but also error handling code as it only approximately covers branches covered up until the lexer comparisons on the last lexeme started, hence only the branches covered by a supposedly correct portion of the input. This additionally filtered set is then used to calculate two numbers: **newly covered branches** and **overall covered branches**, whereas the number of *newly covered branches* is just the sum of branches covered that were never covered by any valid input beforehand. The number of *overall covered branches* is the sum of newly covered branches times two plus, for each already covered branch, one divided by the number of valid inputs that covered that branch—giving a number which ranks new branches high and other branches lower the more often they are hit. The number of newly covered branches is important as inputs that cover new

branches are likely to also cover more features in the code, especially more parsing features. Also, the number of overall covered branches gives a broader estimate on the value of the input, as newly covered branches are ranked high because of their importance and branches that are covered by many inputs are ranked lower, as those are already hot branches in the code which seemingly are traversed by many inputs and need less focus.

Whenever a new, valid input is discovered that covers new branches never seen before, the coverage statistics from above need to be re-calculated. Thus, we store the covered branches until the last comparison as defined above for every input, as this is the baseline for our coverage calculation.

Path Definition Besides coverage, we also calculate and store an approximation of a path for every input, which is defined as follows: we take the list of parser branches covered until the last comparison and, in order of appearance, store for each branch tuple the source basic block id. Hence, if we have a list `[(a, b), (b, c), (c, d)]`, we get a path-tuple `(a, b, c)`. Now, since every branch-tuple is only reported once by our tainting engine, it may well be that there are portions of a path missing (e.g. if a method is traversed twice on the same path, only the first traversal is reflected in the path-tuple), hence this is just an approximation of the actual path taken. Still, for efficiency reasons we believe that this approximation is sufficient enough. Especially since such duplicate branches likely mean that the same parsing feature is traversed twice—thus, there is likely some redundancy in the input. For our input generation it is important that a feature is traversed at least once and the order of features is interesting (an if-statement nested in a while-statement might be different from the opposite case).

Priority Heuristics We already explained heuristics in Chapter 3 for sorting the possible mutations into a priority queue to select the most promising new inputs first and cover new code and therefore new features in the parser of the subject under test faster. The usage of a priority queue is only helpful if the respective heuristics for sorting the values are well designed and match the domain they try to cover. In Chapter 3 we covered subjects that use a parser but no lexer, hence the heuristics were designed for those subjects. In this chapter we extended our idea to subjects that may also have a lexer. Additionally, the heuristic handles different modes: *re-evaluation*, *non-appending* and *appending* mode. In the *re-evaluation mode* the complete priority queue is newly evaluated. This needs some special handling when calculating the heuristic value again for already added values. In the *appending mode* a random character is appended after the mutation was applied. In some cases though we want to apply the mutation *without a random addition*. For this special case we also calculate the heuristic differently.

First, we explain the different calculation steps of our heuristic adaption, then we will go into detail about the different steps. We adapt our heuristic, as follows (a smaller value ranks the input higher in the queue):

1. The **individual value** for the input, defined as follows for the following cases:

a) If the operator is a *token comparison*, the original input *covered more than zero new branches* compared to all valid inputs beforehand, the run was *not a re-evaluation*, and after the mutation a *random value will be appended* this part of the heuristic value is calculated as follows:[23]

   - Create a mapping from stack sizes of the token comparisons of this run to a set of tuples (correction, comparison id); filtering those token comparisons that have the same first input taint index as the current value for which the heuristic is calculated. The underlying mapping used to create this list is the stack-comparison map from Section 4.2.3.
   - Check if the set at the stack size of the current comparison already contains a tuple with the current comparison id, if not add the current (correction, comparison id) tuple to the stack size mapping.
   - Create a list sorted by stack sizes, containing tuples (stack size, - (number of tuples at the stack size)).
   - Add one fallback tuple at the end of the list: (max int, - number of newly covered branches).
   - Use the tuple resulting from this list as priority value by interpreting it as a tuple.

b) If *one of the conditions of item (a) does not hold* and *a random new value will not be appended*:

   - Use a one element tuple containing the tuple with the values (-1, 0).

c) If *none of the above holds*:

   - Apart from the re-calculation case this is a one element tuple containing a tuple with one element: (max int). In the re-calculation case the individual value is inherited if the stack size is larger than 0 and new branches are still covered by the parent input, otherwise it is again the one element tuple with the tuple (max int).

2. The **number of inputs that took the same path** (including this input). A path is defined as above, hence it is not the actual path taken through the program but an approximation. If *no random appending* would be applied, this value is set to 0. Also, the value is set to 0 if the input is smaller than 3 characters or the number of same paths taken is smaller than 6 but larger or equal to 0 (could be manually set to negative values in some border cases to rank an input higher). Gets copied in case of re-calculation of the priority queue, hence this value is only calculated once during creation of the substitution and is then never changed.

3. A **combination of values** that produce a heuristic value for the given input:

[23] Our prototype also requires the stack size to be larger than 0. This serves as a filtering mechanism as the stack size should never be zero except we manually change it—which should not happen for token comparisons. We do not want to rank such mutations high in the priority queue.

- **length of the input minus newly covered branches** (if new branches were covered; else 100)
- + **sum of the length of the same prefixes** in already found valid inputs
- + 1 for every 5 inputs on the **path of the generation tree** (including this input; starting at zero for inputs without a parent)
- if the comparison is a *token comparison*:
  - if the input **covered 0 or less new branches** and the **correction is not part of any already found valid input**: minus 100 times the length of the correction; else 0
  - always: 1 subtracted from the item above
- if the comparison is *not a token comparison*:
  - minus the **length of the correction times 2**

4. The **length of the input**.
5. The **input ID**.

Heuristics Explanation In the following we detail the parts of the heuristic and explain why we had to change some parts of the heuristic, leave out some elements, and add some new elements compared to the heuristics in Section 3.2.2:

Individual Value As the name already suggests, the individual value exists to have a more detailed control over the current value while putting it into the priority queue.

General Case In the most general case, the value is just the maximum integer and sorts the value behind all other values that got an individual correction. The values which got an individual correction are likely more promising and are thus ranked higher than the values that fall into this category. This is the case for option *(c)* in the itemized list above, i.e. if none of the other conditions below hold. More interesting are the other two cases.

Token Comparison The reasoning for case *(a)* is the following: first of all, we want to bring token comparisons to the front of the priority queue, hence we check if the current comparison is a token comparison. But, we only want to consider token comparisons that fulfill additional conditions.

The input from which the comparison originates must have covered more than zero new branches compared to already found valid inputs. With this we only consider those comparisons which were executed for a prefix that found new coverage, i.e. a prefix which, if completed to a closed input, will cover new code and thus likely new features of the subject under test.

Second, the run was not a re-evaluation. In the re-evaluation case we copy the value from the original mutation that is re-added to the queue if the stack size is greater than zero and the input still finds new branches. Thus, when re-evaluating the

queue we do apply the individual value at another position with different conditions and do not consider it again when adding the value to the queue.

And lastly, there will be an appending of a random value after the mutation was applied. For a token comparison there is no random appending if we saw a *valid token comparison* during execution (for details see Section 4.2.4). In that case we will go to option *(b)* which then describes why we need a special handling here.

Once all of those conditions are fulfilled, the individual value is calculated as described in item *(a)*. The calculation essentially creates a large tuple which serves as a sorting order criterion. Each *(correction, comparison id)* is one unique comparison location: the comparison id uniquely identifies the code location of the comparison, the correction is added to diversify the comparisons as it might happen that the same comparison is used with different constant operands (e.g. when comparing against values from a constant array). With this we get a fine grained view on the different comparisons done at each stack size, giving us an insight into the recursive depth the input walked through. Inputs that iterate comparisons at lower stack sizes or have more comparisons at a certain stack size compared to other inputs are ranked higher.

Since we append the tuple *(max int, - number of newly covered branches)*, we favor those inputs that cover comparisons at a higher stack depth if at a lower stack depth they cover the same number of comparisons (i.e. the input covers more), e.g. it holds that `((2, 2), (sys.maxsize, 0)) > ((2, 2), (3, 1), (sys.maxsize, 0))`—both inputs covered two comparisons at stack size two but the input which created the right tuple also covered one comparison at stack size three what we favor. If the number of comparison tuples is equal we sort by number of newly covered branches compared to the branches covered by all valid inputs beforehand.

Overall, we sort the token comparisons of inputs that cover new branches with respect to the number of comparisons and stack sizes, whereas smaller stack sizes and more unique comparisons on smaller stack sizes are more favorable. This keeps the recursion depth low and makes it easier to make prefixes valid, as there is a higher likelihood that we do not need to close any opened features. The reasoning here is that the more unique comparisons happen lower in the stack, the more parser features are covered lower in the stack, hence the covered features are less nested within other features that also need to be closed. For example, closing a simple arithmetic expression like "`1+`" is easier than closing a nested expression like "`1+(`"—in the first case we just need to add some value like "`2`" whereas in the second case we also need to close the already opened parenthesis, resulting in adding at least two characters: "`2)`".

With this, we have a more fine grained stack heuristic designed to the needs of token comparisons, hence *we do not use the size of the difference of the stack to the parent or the average stack size anymore*, which was still used in Section 3.2.2. Especially since we are now mixing the comparisons of different stages, lexing and parsing, it

does not make sense to compare the stack size of a lexer comparison with one of a parser comparison. Also, for non-token comparisons the stack is not of any use anymore as they happen in the lexer which is shallow and typically does not use any recursion.

No Random Appending If random appending of another value after an applied mutation is done, we set the individual value to a tuple containing the tuple `(-1, 0)`. Hence this value will be ranked very high in the priority queue and is likely selected next.[24] We use this because not appending a random value just happens for rare events in which the value to evaluate should be one of the next values to be taken for evaluation. In the token comparison case this happens if we want to try out another token value not yet seen during parsing at the position where the mutation was applied (as explained in Section 4.2.4). In detail: we already executed an input on the subject under test with a token value but the token was valid and the parser stopped at the comparison that matched. Thus, our mutation decision engine selects another token that should be applied at this position and we re-run the new input.

Same Path Taken The reasoning is the same as for the non-lexer case as described in Section 3.2.2—we want to avoid duplicates. The additional filtering for the non-appending mode just exists to deactivate this value for those inputs as they need a special handling and we want to rank them higher (see above).

Calculated Heuristic Value As in Section 3.2.2 we combine a set of values to one part of the heuristic. Again, there is no special order between those values, none of them is more important than the other such that it would be possible to add them to the overall ordering. Let us explain the parts in detail:

Input Length Similar to Section 3.2.2 we consider the input length minus newly covered branches, the reasoning is the same as before. In this version of the heuristic though the average stack sizes are missing (compared to the heuristic in Section 3.2.2), which comes from the fact that the individual value already includes stack sizes. The individual value only considers token comparisons, which is fine though for our case as the stack sizes in the lexer are not of any importance anymore, now that we have token taints and see all parser comparisons.

Same Prefixes As in Section 3.2.2 we want to favor inputs that are more diverse textwise compared to already found valid inputs (including those inputs that ended up in assert calls and timeouts, as often those are just semantically wrong inputs and we want them to be considered here as well).

Number Of Previous Inputs As in Section 3.2.2 we want to favor a breadth first search. As mentioned above, this value is zero for all inputs that do not have a parent input. This, for instance, includes inputs for lexeme learning runs, i.e. inputs that are not taken from the priority queue but generated to learn lexemes individually (see

[24] It might be that there are other values with this individual value added which are then taken first, but this will not happen often. In this case though the other items of the heuristic value play a role for the selection of the next element.

Section 4.2.3). Also, obviously, the first run uses an input without a parent and in some cases it might happen that the priority queue is empty when queried, which would also cause a run without a parent input.

Correction Length As in Section 3.2.2 we want to favor larger substitution values. The non-token comparison case is the same as before (minus the length of the correction, times 2) as we are working on the string level here and longer string constants are more interesting than shorter ones as they likely contain keywords. If the mutation comes from a token comparison though, we consider two cases: in case one the input *covered no new branches and the correction was not yet seen in any valid input*, we rank the substitution very high by multiplying its length with -100 and subtract one. In the other case we just subtract one. If the correction is already used in some valid input it does not make sense to use it again, hence it is not considered in the token comparison case again. If the input covers new branches, we do not want to consider the token length here, as we are already on the token level and tokens are equally important, it is more important to close this prefix that covers new branches. If the input does not cover new branches and the correction representing the token value used for substitution was not yet seen, then we want to rank the input high, hence we multiply the substitution length with -100. The actual length of the substitution is not as important as in the plain string case without a lexer, but it is still interesting to favor keywords, hence we kept the heuristic similar. Still, we want to rank token comparisons higher, hence we use a larger constant for calculating the *token value correction length* part of the heuristic.

Input Length The reasoning is the same as for the non-lexer case described in Section 3.2.2—we want to favor shorter inputs.

ID The reasoning is the same as for the non-lexer case described in Section 3.2.2—this is a tie breaker, inputs that were discovered earlier are favored.

Special Inserted Values During the generation of new inputs it happens that besides the generic generation of possible substitutions we also build custom substitutions which we want to rank at specific positions in the priority queue.

First, the substitutions stemming from runs which found a **matching token comparison** (i.e. the parser successfully parsed the lexeme we are currently observing) are ranked the highest, as the individual value results in the tuple `(-1, 0)`, ranking those elements right at the beginning of the queue. This makes sense as we want those values to be executed as soon as possible again, replacing the correct token with an incorrect one to get more information about possible options. The matching token comparisons are explained in detail in Section 4.2.4 in the special handling of a valid token comparison paragraph.

Second, in many cases we do not only run the input with the substitution, but also append some random value (see Section 4.1.4 for more information). In case the input without the

random appendix already caused the program to finish with exit code zero or causes a timeout, **substitutions are re-added to the queue during the execution loop**, changing the same-path-taken value to -1[25], ranking them right in front values with the same or worse individual value. This is done because at some point we want to also run the input with a random extension as it might reveal a new path, especially since the prefix is already a valid input.

Finally, there are two kinds of **random substitutions**: one simulating a non-token comparison with stack-size zero and one simulating a token-comparison with a maximum integer stack-size. Details about the special random substitutions are explained in Section 4.2.4. The token comparison random values are ranked as if the comparison happened at the maximum integer stack-size on the last-index character and with random appending activated. Hence, they are mixed with all similar token comparisons but because of the maximum integer stack-size their ranking is slightly worse compared to other token comparisons working on the same index. Still, this only holds if new coverage was achieved in the run compared to valid inputs, otherwise there is no difference to other token comparisons of the run (though they are added last, hence their ID is the highest). Also, we cannot say anything about the ranking compared to other values from other executions, as their values are completely different.

Random non-token substitutions are ranked behind most token substitutions that cover new code—like most other non-token substitutions. They are intermixed with other non-token substitutions of that run and ranked rather low as their substitution length is one and they are added last, resulting in a the highest ID-values for a run. They are typically used if there is no token substitution with coverage or new substitutions never seen in valid inputs. In such cases, we reached a local plateau and need to try out different inputs, ranking the random substitutions along the normal substitutions is sufficient in that case.

Queue Recalculation As already mentioned, whenever a valid input is found which also covers new code, we perform a recalculation of the heuristic value for all inputs in the priority queue. This means, for every input we do the following:

- The number of *newly covered branches* (based on the branches stored for each input) is updated.
- The *individual correction* is either inherited if still new branches are covered, or it is set to the maximum integer tuple.
- The *calculated heuristic value* is updated based on the currently available information.

If it ever happens that a function is defined as parsing in a later run the stored branches for the inputs do not change, hence the branches of this function might be missing. As the definition of parsing and lexing functions is an approximation anyway we accept this imprecision but

[25] In some border cases it might happen that this value is reset to zero, e.g. if the complete queue gets recalculated and the input length is less than three. Though, this should typically mean that another interesting input was found and we need to re-evaluate the ordering of values in the queue anyway.

consider it as rare and new inputs (e.g. the one that caused a function to be considered as parsing) will typically cover those new branches and fill the blind spot. All other heuristic values do not change and are inherited from the original value.

Summary The heuristics above mostly concentrate on token comparisons, i.e. if the comparison is not a token comparison some parts of the heuristics calculation are deactivated. The reasoning behind this is that in contrast to the heuristics used in Chapter 3, we now have knowledge about tokens and can differentiate between tokenization and non-tokenization code. The heuristics in Section 3.2.2 partially try to cope with this missing information by combining different information from the execution to infer how the parser behaves if the lexer accepted a character—i.e. the heuristics need to handle the case that the constant character and string values used in comparisons do not directly hint the actual valid characters at a certain position if a lexer-parser setup is involved. Now though, we always extract for each input character the valid characters for its position, hence likely any of the possible characters is correct. Thus, the heuristics need to solve a different task here: *filtering more uncertainty introduced by the token-taint approximation* and *prioritization of token comparisons if a lexing stage is recognized*. Hence, we expect our heuristics to be sufficient for both cases, especially as we expect only simple parsers to not use a lexer-parser splitting.

4.2.6 Input Generation Loop

Sections 4.2.1 to 4.2.5 describe the implementation details for handling tokens in our input generation loop. As already mentioned in Section 4.1.4, this also means that we need to design the overall input inference loop differently. Thus, this section gives an overview on the implementation of the extended generation loop. It remains to be said that the *pruning* and *out of bounds* (maximum index larger than input length) checks are not done anymore when using tokens (see Section 3.2.2). We have much more uncertainty in the traces we generate and thus need to reduce the discarding of possible mutations.

Continuation Set Before we start explaining the overall generation loop in detail, we need to clarify a central part of the input generation: **random extensions**. Oftentimes we need random characters that are appended to input prefixes or even serve as (re-)starting inputs. Those random characters are initially the set `string.printable` of the Python language [144][26]. They cover all values that are typically used in human readable input format—the formats we concentrate on in this thesis. Using another (larger or smaller set) would certainly work as well as those random characters are just placeholder. If the set is too small though, we might end up **not** generating enough invalid extensions, thus possibly missing out on characters that would be lexed in comparisons after the extension is accepted. Remember, with an invalid suffix given to the subject under test the lexer would be forced to check all its internal options before rejecting the suffix. By giving valid suffixes, the lexer would stop once the suffix is matched,

[26] We omit referencing the concrete version used in our evaluation as this is a technical detail that should not influence the overall input generation (if this set ever changed/changes at all).

thus we do not see the remaining options that would be tried after the comparison that matched the suffix—options that are important to know for our input inference.

During execution though this set of continuations changes—for every substitution that is added to the priority queue in the standard way[27] we check if the substitution value is part of the continuation list and if so we discard it from the list. The reasoning for this discarding is that we explicitly want to extend prefixes with invalid new characters to not run into matching comparisons. This increases the likelihood to see all comparisons done on this newly appended character. If during execution all characters from the continuation set are discarded we re-fill the set again with all `string.printable` [144] characters to avoid not having any continuation if needed.

Loop Overview As discussed in Section 4.1.4, we start with a random character, run the program, analyze the trace, store inputs and possible mutations in a priority queue based on their heuristic value, extract the most promising input, mutate it, and restart the inference loop. The new feature compared to the inference loop from Section 3.2.2 is the adaption to token learning and usage during inference. In the following we detail this new feature, i.e. the adaptions made to improve the input inference in the presence of a lexer in the subject under test.

Input Generation During input generation we take an input or a substitution from some input provider source as defined below and generate one input with and possibly one without random continuation which both are executed in the overall execution loop. The inputs are checked for uniqueness with respect to the overall execution, i.e if both inputs were already generated at some point they are discarded and the input sources are queried again.

Token Learning The token learning is preferred over any other input source. As described in Section 4.2.3, we learn a token and stack mapping by running the subject on individual lexemes without random extensions. Whenever a new lexeme is detected, the token learning is used as the preferred input provider.

Empty Queue In rare cases it might happen that the priority queue runs empty, i.e. there is no other substitution that can be tested—most often this happens if the queue consists of substitutions that just generate inputs already executed. In this case we continue the execution by taking a random character. Also, as we generate inputs typically with a random appendix, we create a second input with two random characters. Both of those inputs do not have a parent execution, which influences the heuristics calculated but also avoids re-adding any mutation if the execution is shortcutted (see below).

Substitution The substitution case is the core input generation case, the central part besides the heuristic ranking of possible substitutions. Here we take the most promising substitution from the queue and replace the part of the input starting from the first tainting index used

[27] There is on exception described in this section that adds an input directly to the queue and not through the standard heuristic value calculation and addition path.

in the comparison up to the end of the input with the respective substitution. Also, we add some random extension to the newly generated input as we believe the just generated input to at least be a valid prefix. For parser comparisons we typically add a random lexeme, as we want the random extension to trigger parser comparisons again and a random character extension, which is used for other cases where we append a random value, would likely get rejected in the lexer already. The randomly appended token values come from the token mapping as explained in Section 4.2.3. If no appending is done (which may happen for some mutations), the single input is used as if it is randomly appended and only this input is executed. If the non-appended input was already executed, we only consider the random appended input in the upcoming execution—if it was not yet executed. In the following we detail the different input generation options for different comparison types, explaining how the random continuation is added and if there are any additional details when substituting the original input with the stored correction:

String Comparison The string continuation is added with a whitespace between the input and the random continuation as typically keywords are first read character by character and then compared with the *strcmp()* function and this whitespace oftentimes splits the random continuation away from the string that was just used as substitution.

Token Comparison The same is done as for string comparisons, but additionally there is also a whitespace added between the substitution and the input prefix and both whitespaces are only added if there is not already one, because, analogously to the string comparison case, a whitespace between lexemes is typically allowed and often required, hence we approximate this to reduce the number of incorrectly generated inputs. The random continuation is a random lexeme (from the token mapping as discussed in Section 4.2.3) if the substitution did not end with a whitespace. In this case it is likely that we found a correct lexeme for substitution and we want to see what the parser accepts next, hence we need a valid lexeme. If the substitution did end with a whitespace, we add the substitution again because in this case it is unlikely that the original substitution is correct: lexemes are typically surrounded by whitespaces in an input but do not contain one. Thus, we want to find out what happens if we just append the same value again.

String Length Comparison We never add a random continuation as the actual length of the substitution is important and we just want to see how the parser reacts to a substring of the input with the correct length. The goal is to reveal possible guarded comparisons by string length comparisons, hence we just perform this one replacement.

Other Comparisons The continuation is appended without any whitespaces.

Execution The inputs from the input generation are executed first with and then without the randomly appended character—except for cases in which only one input is executed, this input is considered as randomly appended and the sole input executed for the loop. This distinction between appended and non-appended inputs is important for two cases: starting a traced

execution and abridging the loop if the non-appended input causes the program to exit with exit code zero or a timeout.

Traced Execution Tracing the program execution is rather expensive—the execution time of the subject under test increases significantly when traced. This pays off if the gained information from the trace guides the execution efficiently enough to the goal of generating valid inputs. In the case of inputs without a random appended character though we already assume the whole input to be at least a valid prefix for a valid input—we assumed to have replaced the first invalid portion of the input, e.g. the last parsed lexeme, with a valid one. Thus, the only thing that is to be tested for those inputs is: are they fully valid? Hence, we run those inputs without tracing to reduce used resources and only if we abridge the loop (see next paragraph) or have an input with a randomly appended character we trace the execution. If not abridged, it is sufficient to only trace the randomly appended input as we were either right with our assumption that the substitution is valid and the parser checks the random appendix from which we will then extract the next possible substitutions, or the substitution was wrong, in which case the parser stops at the incorrect substitution resulting in the same extracted information as if the random appendix was never added. If the input was valid we trace again to see if there are any comparisons not yet seen—in those scarce cases it is better to over-approximate and possibly have too much information than to miss any comparison.

Abridging If the non-appended input is executed and either the subject under test returns exit code zero or timeouts, we re-run it under tracing and collect the traced information. To not lose any information from the possibly partially missing run we add the substitution back to the queue with a better heuristic value compared to its original heuristic value as a specially inserted input as described in Section 4.2.5. We check if the just traced execution covers new code, in which case we directly go to the result handling (see next paragraph for details). If no new coverage is found, the random appended input is executed and the loop finishes the iteration as usual.

Result Handling The information from the traced and tainted execution is finally given to the substitution extraction which makes use of the techniques described in Sections 4.2.2 to 4.2.5 to generate new substitutions and add them to the queue. At this point we also check if the last executed input was valid (as defined in Section 4.2.5) and if we need to re-evaluate the queue. If it is invalid (non-zero exit code) and the last instruction is not an assert instruction and the input is below 2000 characters (to avoid infinitely large inputs), the execution is just analyzed and the substitutions are added to the queue. Otherwise, we may store some information about the just executed input (like covered branches) or report it as valid (as described in Section 4.2.5). Once the inputs are added a new input queried from our input sources gets generated, and the loop begins again.

New Heuristics As already mentioned, we include new heuristics as described in Section 4.2.5. Those heuristics take into account that we have an approximation of token comparisons in the parser and are designed to lower the noise introduced by this approximation while making use

of the fact that we have knowledge about parser comparisons even in the presence of a lexer. Thus, even if a program uses a tokenizer, we can efficiently infer syntactically valid inputs.

4.3 Limitations And Assumptions

In Chapter 3 we already mentioned some limitations and assumptions for our approach—most of them are still valid. However, with our extensions as described in this chapter we resolve some of them while introducing new ones. We do not list all limitations and assumptions again in this section, but rather detail the different items by mentioning new and solved limitations. Hence, if a limitation or assumption is not listed here, it is still valid as mentioned in Section 3.3.

Parser comparisons can now be analyzed in presence of a lexer.
Before, in the presence of a lexer, the input generation efficiency of our technique was reduced. Our technique could only detect and use comparisons in the lexer in that case, resulting in an inefficient input generation inference. Now, we are able to track the tokens a lexer generates and make the parser comparisons visible and usable again. Thus, this limitation of Chapter 3 is mainly resolved.

Typical lexer patterns are expected.
If a tool taints every value not only based on direct data dependencies but also based on control-flow dependencies, most values will likely be tainted with almost all taints from the input. Hence, control flow based taints would produce a large amount of noise. This can easily be imagined if we think about a subject that iterates over the input characters until the end-of-file character is found (e.g. `while (input[i] != EOF)...`), having the remaining code in the `while` body. Any instruction in the body would get a control-flow taint from the comparison, i.e. every instruction would be tainted with the taints of the current or already consumed characters. Thus, we need to detect and select very specific lexer patterns in the code and only produce control flow taints for those lexing conditions. In Section 4.2.1 we go into detail what we define as a lexing pattern and how we propagate the resulting taints through the code. Still, any lexing code that does not follow one of our patterns will be missed, we do not generate any token taints for the involved tokens, and in turn will not produce any parser comparisons for them. Chapter 5 will partially solve this limitation.

Parsing and lexing code are expected to be well divided.
We assume that the parser and the lexer are not interleaved in the subject under test, the code of each part lies in independent functions. This requirement is needed to filter out noise stemming from misclassified token taints. While this might look like a strong assumption, a typical programmer would write code like this. Aho et al. also see a lexical analyzer as an independent part of the parser which reports tokens to the syntactic stage (Chapter 3 in the *Dragon Book* [3]). Also, we expect the lexer code to be called token by token in between parser operations—which we assume to be the standard way of calling the lexer during parsing.

4.4 Summary

In this chapter, we extend the ideas of Chapter 3 by improving the handling of tokenizing code in the subject under test. While the basic approach as presented in Chapter 3 is already able to create a diverse set of syntactically valid inputs, it needs specialized heuristics to overcome the missing taints in parser comparisons if a lexer is used in the subject under test. Without the extension presented in this chapter the tainting engine would stop tainting in the lexer, because lexer tokens are typically generated via control-flow, which was not considered by our basic dynamic tainting approach. Now, we detect typical lexer code and are able to attach taints from lexer comparisons to the tokens they generate, advancing the taints to the parser comparisons. Using additional analysis to translate the program specific token values to lexemes and vice versa, *we are able to apply our input generation loop similarly to our original ideas as discussed in Chapter 3—even in the presence of a lexer.*

5 | Input Inference And Fuzzing

In Chapter 3 we laid the foundation for input inference, defining an approach which is able to generate syntactically valid inputs *out-of-thin-air* on subjects with simple parsers excluding lexers. In Chapter 4 we extended this approach to also analyze lexer code and lift the ideas from this basic approach to the world of *lexing **and** parsing*—enabling analyzing even more complex input validators. With this technique we are already able to generate a syntactically feature rich input set from scratch, just by observing the subject under test evaluating carefully crafted inputs from our approach. **One key problem remains though: semantic diversity in syntactically valid inputs—our objective number three.** While our inputs typically cover most if not all of the syntactic features and detect and extract the atomical elements of the underlying grammar (the terminal symbols), our tools likely do not perform well in terms of covering the semantic features. In this chapter we will go into detail on *how we can enhance the input generation process to also cover semantics.*

Scope We found that it does not make sense to run input inference alone as it is designed to generate diverse syntactically valid inputs, which is a time consuming task—resulting in a low amount of inputs. Hence, while our input generation could be considered as a standalone approach, we would rather see it as an information gathering module for other approaches, e.g. fuzzers. Results from different papers support our reasoning: a well chosen seed set is important for fuzzing (besides the fact that our extracted tokens can be used as building blocks for mutations). Ma et al. [98] try to reduce the number of seed inputs with a minimum set solving algorithm. Herrera et al. [62] found out that the seeds used for fuzzing should be neither empty nor too many, they recommend having a large seed set and minimizing it with a seed minimization tool. They tested several different tools and were not able to pin down the one best minimization tool, still using any minimization tool yielded on average better results than using the full set of seed inputs or an empty input. The reduction of inputs has the advantage of shorter iteration times—hence the fuzzer can discard inputs that are not worth exploring faster.[1]

The approach as presented in Chapter 4 only concentrates on syntactic diversity by covering as much of the parser as possible. Hence, syntactically equivalent inputs are typically ranked lower during input generation and we do not optimize for semantic diversity. Still, syntactically equivalent inputs can largely differ during execution. For example, for an arithmetic expression evaluator the inputs `12 / 6` and `12 / 0` might be syntactically equivalent, but semantically very different. The first input will evaluate to 2 while the second input will result in a division by zero error, the program might return an error or even crash. And this semantic phase, the

[1] It has to be mentioned that they are only testing their claim for AFL, the results might be different for other fuzzers and fuzzing approaches.

program logic phase, is implemented for many different subjects. A tool that parses JSON [71] inputs (and is not a parser library or JSON verifier) typically converts the input to internal values, and then runs the program logic on those values. Interpreters like TINYC [78], LISP [76], and MJS [26] not only parse the code but also execute it to produce a result. Hence, there is a need to explore new inputs in the boundaries of syntactic correctness.

Approach Overview *Therefore, we augment our input generation with a fuzzing campaign by providing information from the input generation to the fuzzer in form of a set of seed inputs and token information like lexemes as a dictionary for later mutation and recombination.* The following abstract algorithm shows a very brief overview on our approach:

```
1  def inputInference(subject):
2    inst_sut = instrument(subject)
3    input_queue = [random.nextChar()]
4    token_information = TokenInformation()
5    collected_inputs = set()
6    while didMakeProgress():
7      if token_information.hasTokensToLearn():
8        inp = token_information.nextTokenToLearn()
9      else:
10       inp = queue.pop()
11     trace = inst_sut.run(inp)
12     if wasInteresting(trace):
13       collected_inputs.add(inp)
14     taints = taintEngine.analyze(trace)
15     new_inputs = extract_inputs(taints, inp, token_information)
16     input_queue.sortAndAdd(new_inputs)
17   fuzzSUT(subject, collected_inputs, token_information)
```

In contrast to the techniques presented in Chapter 3 and Chapter 4, we do not run our input generation infinitely (we already talked about this in Section 4.1.4), but use an abortion criterion (Line 6)—we abort after a certain amount of inputs did not cover new branches in the parser code of the subject. At this point we expect the input generation to not discover inputs with syntactically new features. Furthermore, we collect interesting inputs (Line 12)—in our case valid inputs that cover new branches in the subject that no other input covered beforehand. This set is initially empty (Line 5). Once the input generation loop stops, we start another fuzzing session (Line 17)—preferably with a fuzzer that can make use of both, *our collected inputs as well as the token information we extracted.* Currently, we use AFL [160] which is provided with our collected inputs as seeds and our token information in form of a dictionary—the collected lexemes during input generation serve as additional building blocks during fuzzing.

5.1 Approach

The approach itself is straightforward: we take the information gathered throughout the input inference phase and provide this information to the fuzzer or any other subsequent tool.

In this section we explain how to apply our valid inputs as *seeds* and the tokens as *dictionary* to a subsequent fuzzer as an example of one use case for the gathered information. Furthermore, we sketch how the fuzzer can make use of the given information and why we think this improves the overall fuzzing performance.

Seeds In Section 4.1.4 we already explained that some inputs of our inference loop are outputted. We assume those inputs to be syntactically valid based on the behavior of the subject under test when running on those inputs (like exiting with code zero or running into a timeout). In fact, every input covers branches that were not covered by any earlier generated input.[2] Thus, this information can directly be used as seed information for a fuzzer as long as the subsequent tool is not more strict in the selection of seed inputs. For example, we will later see that for the fuzzer used in our evaluation we need to filter out inputs that cause a *timeout*.

Dictionary In Section 4.2.3 we describe how token values are mapped to lexemes that occur in the inputs we generated. Those lexemes can also be used as a dictionary for a fuzzer—i.e. a set of keywords that can be used as building blocks during mutation [157]. A naïve fuzzer might solely use characters as mutation values, replacing one character with another during input mutation. With this dictionary the tool can use complete lexemes as replacements.

Fuzzing Benefits Typically, inputs that are not syntactically valid would be rejected early in the parsing phase—in the lexer or in the parser. Hence, the fuzzer takes a lot of time to first generate valid lexemes to overcome the lexer. As in general longer lexemes are lexed atomically, the fuzzer would not get any feedback on partial correctness and thus needs to guess full lexemes—which might take a long time. With our dictionary this part of the input generation is already solved, the fuzzer has the building blocks for generating valid inputs.

A second problem is the exploration of the input space. Without valid seeds, the fuzzer would need to infer how the different lexemes are combined to build valid inputs. Only those syntactically valid inputs will trigger the semantic phase of a subject under test. With the seeds we generate, we overcome this issue by hinting the fuzzer towards different, larger syntactic building blocks. Those building blocks can be mutated and recombined, making it possible to generate different syntactically valid inputs without the need to first generate a set of diverse inputs.

5.2 Implementation

Export Optimization As described in Chapter 4, our approach already gathers information during input inference which can be directly used in the subsequent fuzzing campaign. The *seeds are generated as syntactically valid inputs* during exploration and *the values for the dictionary are already part of the token-input mapping* that is needed to decipher the token comparisons in the parser.

[2] It could happen that a later generated input fully subsumes the coverage of an earlier input (e.g. if the earlier input is a prefix of the later input).

The export of dictionary values can be customized to reduce the number of choices for the subsequent consumer like AFL. For example, when reporting tokens to AFL, for every token value we iterate the set of learned lexemes (see the token-mapping as defined in Section 4.2.3) and do the following: we add one lexeme after another to the export set, and before adding any lexeme we first check if there is any lexeme in the set with starting characters that are equal to the about to be added lexeme. If so, and if the lexeme to be added is longer than three characters, we delete the already added lexeme from the set, add the current lexeme, and take the next lexeme. For example, if the set already contains *'abcde'*, *'abcd'* would cause the deletion of *'abcde'* and *'abcd'* would be added. In parallel, we check if any lexeme from the set subsumes the about to be added lexeme—for lexemes from the set with more than three characters. If so, we stop the iteration and do not add any value. For example, if the set already contains *'abcd'*, *'abcde'* would not be added. If none of the above happens, we just add the new lexeme. Remaining duplicate lexemes are deleted when merging the resulting sets over all individual token values. Duplicate lexemes are obviously unnecessary to report to AFL as it just uses the reported lexemes as a dictionary—a set of values. The threshold of three characters is arbitrarily defined to keep small lexemes that consist of just a few characters—they might be control characters like parenthesis and we want to avoid filtering them.

Deleting larger lexemes that are subsumed by smaller lexemes makes sense as the token learning is not always precise. In some cases incorrect, random suffixes are learned (e.g. because the lexer accesses one more character after a keyword before the respective token value for the keyword is generated and given to the parser). In such cases the filtering algorithm might filter out wrongly suffixed keywords and report shorter but likely correct keywords. Typically, if the token value belongs to a very specific keyword and the learning works as intended, we only have this one keyword in the set of possible lexemes for the token value, which is then reported without filtering. Similarly, for variables, numbers and other more dynamic syntactical structures we might have many different valid lexemes, thus we filter some of those lexemes to reduce noise. Thus, the filtering only happens for cases in which either the keywords are not learned fully correctly anyway or for token values that represent more dynamic values. In both cases we want to reduce the number of reported lexemes which means reducing noise.

Furthermore, for AFL we filter for those valid inputs that do not timeout. If we would have seed inputs that timeout, AFL would reject the whole seed set as it requires all seed inputs to let the program finish in a certain time frame. If no token or no valid input was found, a whitespace character is printed instead as the only element for the respective set.

Independence Our techniques as described in Chapter 3 and Chapter 4 are run before and completely independent from any consuming tool, which gives us some benefits: 1. the consuming approach can be replaced easily as it is weakly coupled, 2. different tools can be run in parallel using the same inferred values from our approaches, and 3. our approaches only need to be run once on the subject under test to extract and store the wanted values, after that they only need to be restarted if the parser changes. Consequently, adding another tool to the tool chain is often just a matter of collecting and aggregating the output of our tool differently.

We decided to use AFL in this version of our tool chain, but any other fuzzing technique would work as well (as long as it also uses seeds and/or a dictionary; otherwise other data might have to be outputted by our technique). The small seed set our tool reports will be augmented by AFL with the common AFL mutation operators that work well to explore the semantic possibilities within the syntactic features covered by the seeds. AFL [160] changes bits and bytes (and thus characters) in two different stages—deterministic and non-deterministic mutations [42, 159][3], explained in the following.

In the deterministic phase AFL bit-flips all input bytes; adds, subtracts, and replaces bytes with pre-defined constants; and finally takes a user given dictionary (if present) as well as an inferred dictionary of magic values and inserts and replaces input locations with the given values[4]. The non-deterministic phase has a *havoc* phase which applies the deterministic mutations randomly, as well as mutations in which blocks of inputs are overwritten or inserted—all of those are stacked (i.e. several mutations are applied at once in one input; the number of applied mutations is randomly chosen between 2 and 128).

In the second non-deterministic stage, *splicing* (which *"by default is activated only after the fuzzer goes through a full cycle of the entire queue without any new finding"* [42]), AFL takes an input from the test corpus, recombines it with the current input, and applies the havoc mutation phase, generating a completely new input from two parent inputs. All of this might result in either new semantic features (e.g. replacing the divisor from a non-zero value to zero) or in new syntactic combinations (e.g. removing an *else* branch of an *if* statement or adding a *while* loop into another *while* loop). Without this AFL step, we would not uncover such semantic features and combinations, but without our approach step, AFL would take a lot of time to uncover the syntactic features and only then it would cover the semantic features. Hence, one approach cannot perform that well alone, only the combination of both makes it possible to *cover syntactic as well as semantic features.*

5.3 Limitations And Assumptions

As this technique builds up upon the techniques presented in Chapter 3 and Chapter 4, it suffers from similar limitations. Again, we explain the differences to the limitations as discussed in Section 4.3. If we do not list a limitation or assumption here again, it is still valid as defined before. Hence, we still require a recursive descent parser with an underlying context-free grammar, which parses mostly from left to right as well as the assumptions introduced by analyzing the lexer. Also, we still require the parser in the subject under test to be implemented close to the textbook, otherwise our heuristics might not work as intended and mislead the input inference.

[3] For completeness: Fioraldi et al. [42] take a deeper look into the implementation of AFL, explaining the details of AFL in an easily readable form. The original reference though is the source code of AFL itself [159], which should always serve as the technical reference.

[4] The inferred dictionary is constructed *"during the bitflip stage by looking for groups of bits that, when changed, always produce the same coverage, a sign that they might be part of a magic value"* [42].

Semantic correctness is now partially tested by our tool chain.

Our approach performs well in generating syntactically diverse inputs, but since it is targeting the parser only (and specifically the heuristics only incorporate lexer and parser code as far as possible) it is not explicitly generating inputs that cover different features of the code that comes after the parsing step. With the example presented in this chapter, applying a fuzzer consecutively to our input inference technique, we show how to partially solve this limitation of our approach, using the knowledge extracted from the parsing step to boost fuzzers. While this technique produces a diverse set of inputs that are syntactically similar to the seeds (as the fuzzing techniques we are currently using struggle to generate syntactic diversity), the inputs are typically semantically more diverse. There is still a limitation to this: *semantic constraints*. If the variable parts of the input are unconstrained or weakly constrained (e.g. a number can only be replaced by another number or a variable name needs to have letters only), most fuzzers will hit enough different values to also find those that still fulfill the constraints. For strong constraints (e.g. a *use-def dependency*), most (random) fuzzers will break the requirements and the semantic validation of the input will fail early.

The consecutive tool must work on the subject.

A slight addition to the limitations of our approach used standalone: the subject must be analyzable by the tool we use after running our approach alone. While our implementation is still a prototype and we believe that at least mature fuzzers like AFL will not fail in fuzzing the subject if our technique succeeds, it might still happen. Also, if one wants to use a more specific fuzzer as a consecutive fuzzer, the range of subjects that can be tested must fulfill the limitations discussed for our approach as well as for the additional fuzzer.

5.4 Summary

This chapter concludes our approach and combines the different parts presented in Chapters 3 to 5. We combine our method for generating inputs using feedback from a dynamic analysis of the recursive descent parser in a subject under test with generic fuzzing. Therefore, we first apply our approach to extract a diverse set of syntactically diverse and valid inputs as well as a dictionary of lexemes which can be used as building blocks for the subsequent fuzzing stage. Both, the inputs and lexemes are given to the fuzzer as seeds and a dictionary, which serve as a fuzzing base and can be leveraged during input generation as additional information about the subject under test.

6 | Evaluation

In Chapters 3 to 5 we detailed the different stages of our approach, showing how our idea works and how it is supposed to infer and utilize information from a subject under test, which uses a recursive descent parser to parse inputs. This chapter evaluates the different parts and combinations, showing that the analysis of recursive descent parsers during program execution is not only possible but also yields newly detected portions of the input space in form of tokens and inputs compared to the *state-of-the-art* tool AFL.

For this chapter we use the evaluation results from our paper *Learning Input Tokens for Effective Fuzzing* [102] which not only show the effectiveness of the combination of all approaches as discussed in Chapters 3 to 5, but also evaluate the basic approach without lexer detection as discussed in Chapter 3 (*Parser-Directed Fuzzing* [105]). This evaluation will illuminate how the different parts of our approach influence our results, i.e. we show that 1. we can build inputs by tracking input characters as described in Chapter 3 by evaluating the tool pFuzzer, 2. that we can detect tokens as detailed in Chapter 4 by analyzing the results produced by the token extraction stage of lFuzzer, 3. and that the combination of our input inference with well established fuzzing techniques (Chapter 5) can boost the overall fuzzing performance. Our new techniques are compared against AFL [160], with and without a given naïve dictionary of tokens, evaluating if and how dictionaries influence the overall fuzzing process. In this chapter we detail the setup of the evaluation, the used subjects, discuss the different research questions, then describe the experiments used to answer those questions and explain the lessons learned from them. We explain the threats to validity and then summarize the evaluation and wrap this chapter up. **With the asked research questions (Section 6.3) and the resulting evaluations we answer if we solve each objective as discussed in Section 1.1.**

6.1 Setup

Tools Based On Our Approaches Our first tool, pFuzzer (a prototype implementation based on the ideas of Chapter 3 and thus our paper *Parser-Directed Fuzzing* [105]), is used to show that we solved our first objective as defined in Section 1.1: *generating syntactically diverse inputs*. pFuzzer does not contain any token specific extensions as discussed in Chapter 4 (only heuristics as discussed in Chapter 3 that try to overcome the missing information if a lexer is present in the subject under test). This additional analysis power is implemented in our second tool, lFuzzer (a prototype implementation based on the ideas of Chapter 4 and thus our paper *Learning Input Tokens for Effective Fuzzing* [102]), is used to show that we solved our second

objective: *being able to analyze the tokenizer if present in a subject under test.*[1] In our evaluation, we combine AFL with lFuzzer, as discussed in Chapter 5 and our paper *Learning Input Tokens for Effective Fuzzing* [102], showing that we solved our third objective: *generating syntactically correct and semantically diverse inputs.*

State-Of-The-Art Tool We compare our approaches against AFL—one of the *state-of-the-art* greybox coverage-driven mutational fuzzers. It randomly performs different mutations on inputs, collects code coverage information while running those mutated inputs on the subject under test and selects inputs based on this coverage feedback. Hence, AFL works with less code analysis and thus less inferred information about the subject under test compared to pFuzzer or lFuzzer, therefore it is applicable to a more general set of subjects and serves as a baseline. We already discussed AFL and greybox fuzzers in general and in the context of other fuzzing techniques in Section 2.3.

Evaluation Scope The focus of this evaluation lies on showing that the ideas implemented in pFuzzer and lFuzzer actually work as expected, i.e. we want to show that we solve our objectives as defined in Section 1.1. On the other hand, we do not want to perform a direct comparison of fuzzing performance of different fuzzers, especially as the fast fuzzing stage in lFuzzer can be easily substituted with other techniques than AFL. Also, in *Parser-Directed Fuzzing* [105] we have already shown that KLEE [23], a symbolic execution based test input generator, cannot generate inputs for parsers properly—likely because of the path explosion problem. Hence, we omit the results for KLEE in this thesis as we did it in our paper *Learning Input Tokens for Effective Fuzzing* [102]. Furthermore, we specifically do not compare against tools that need a seed corpus (like Glade [13] or AutoGram [65]), as lFuzzer itself works without one—they would only serve as an upper bound for the resulting fuzzing performance.

Technical Details AFL is run with `AFL_SKIP_CPUFREQ` enabled as our test system does not allow changing the CPU scaling policy. Also, AFL requires a valid input to start with. Since we want to find out how well each tool can produce syntactically valid inputs without prior knowledge about the subject, we decided to give it for each subject a file with one space character as content. Every subject in our test set accepts a space character as valid input, still the input is simple enough to not give AFL a huge advantage compared to lFuzzer and pFuzzer—if any advantage at all. The size of the AFL generated inputs is not capped, pFuzzer caps the inputs to 200 characters and lFuzzer to 2000 characters.[2] For all tools we determine if an input is valid by checking the return value of the subject under test—*exit code zero means valid, non-zero is invalid.* Inputs that timeout after 10 seconds are also considered as valid, as

[1] We will later see that lFuzzer is not evaluated as a standalone tool. The reason for this is that we see this underlying approach as a preparational approach which should be used in combination with existing techniques as AFL. Still, we will evaluate some results of lFuzzer on its own, showing that we are actually able to analyze the tokenizer.

[2] We wanted to use the original version of pFuzzer from the paper *Parser-Directed Fuzzing* [105] which allows a smaller number of characters in the input. Sill, both tools report valid inputs with less than 100 characters, hence the difference in the maximum number of characters seems to have no influence.

parsers likely do not diverge, but a following semantic phase or other program logic might get stuck while executing on the already parsed input.

The experiments were run on an *Ubuntu 14.04.5* Docker container with an up to *3.3 GHz Intel Xeon E7-8867* processor for *24 hours*. The experiments were run *four* times to mitigate the non-determinism of all tools, we report the average results as well as the maximum and minimum results per run and point in time. *For example*: say lFuzzer run one covered 5% of code on tinyC after 10 seconds, lFuzzer run two only 4%, then we would print the results from run one for showing the maximum coverage. Now after 20 seconds the results are shifted and run two has 11% and run one only 10%, then we would print at the 20 seconds mark the results from run two for showing the maximum coverage. Technically, for each point in time (with a resolution of ten seconds), we plot the minimal, maximal, and average coverage the respective tool has achieved on that subject at that time. Concretely, this is the last coverage reported before that point in time, i.e. the point at ten seconds in the graph would use the last reported coverage before ten seconds, e.g. from 9.8 seconds.

Tool Setups AFL is run with and without a dictionary (a naïve extraction of strings from the LLVM bitcode file), pFuzzer is run once for *24 hours* standalone and once with the same threshold as lFuzzer to switch to AFL (using the seeds pFuzzer produced). lFuzzer is only run in combination with AFL, the threshold for switching to AFL is 1000 iterations without an input that covered new code. We count every execution of the subject under test as one iteration, even if it is executed on the same input first without and then with tracing activated. In Section 4.2.5 we defined when the execution counter is reset.

For pFuzzer with AFL and lFuzzer we filter out inputs that cause a timeout on the subject under test before giving them to AFL, as those would not be accepted as valid inputs for AFL. They are still included as valid inputs overall, since timeouts typically happen in code after the parsing stage (e.g. for some subjects like tinyC it is easy to generate inputs that cause infinite while loops, causing the interpreter to diverge). If no valid inputs (and for lFuzzer no tokens) were found during seed generation, one test (and one dictionary file) containing one whitespace character each is given to AFL (again, because AFL needs at least one valid test). Apart from the above mentioned configurations we run all tools without specific command line options, including special options for compiling the subjects (except for libraries that need to be added during compilation and dictionaries given to AFL in the tool variations AFL_Dict and lFuzzer). This means every tool uses its internal optimizations and configurations as generically defined by the developers.

6.2 Subjects

To determine how well our approach performs in the real world, we compare pFuzzer and lFuzzer against AFL on different subjects parsing a variety of input formats. These subject are listed in Table 6.1 including the date of access and the lines of code (hinting the complexity of the test subjects involved). The lines of code were counted with the tool *cloc* [5]; using the code

Name	Accessed	Lines of Code
INIH [14]	2018-10-25	293
CSVPARSER [74]	2018-10-25	297
CJSON [34]	2018-10-25	2,483
TINYC [78]	2018-10-25	191
MJS [26]	2018-06-21	10,920
LISP [76]	2019-03-19	2,741

Table 6.1: The subjects used in our evaluation to test the performance of LFUZZER and PFUZZER in comparison with AFL.

and header files as root for the counting algorithm. Concretely, we use the files for counting which are used for evaluating our subjects, because those also include our wrapper for reading input from the command line if it was added (see below). These subjects are randomly selected from GITHUB (`www.github.com`) with the main technical criterion that they consist of only one C file (and possibly a header file)[3].

For some subjects we wrote a wrapper that makes it possible to read inputs from *stdin* (with size up to 999 characters; if the input is larger, the subject under test exits with *code one*), giving all programs the same interface to accept inputs.[4] This does not change the subject semantics much but makes it easier for our evaluation setup, as we do not need to handle special cases. Additionally, some subjects are manually changed by us to report parsing failures as early as possible, typically on the first erroneous character or token that is encountered while parsing (details on this are mentioned in the respective subject description below)—this is a limitation of PFUZZER and LFUZZER as they need early failure reporting and stopping of the parsing step. Finally, we disabled failure reporting on semantically invalid inputs if the subject supports this. For example, if the input was successfully parsed and then rejected in the semantic checker (e.g. because of a missing definition of a used variable), the input would still be reported as (syntactically) valid with return code zero.

Subject Details In the following we present a list of the programs and the formats they parse with some brief description on the perceived complexity of the parsed language. For each

[3] The compilation scripts of LFUZZER and PFUZZER are not able to handle multi-file programs. Both implementations of our approach are designed as a research prototype, hence some design decisions are based on implementation efficiency and complexity rather than completeness. This restriction is not a restriction of the approach but only restricted by the implementation itself.

[4] While checking the scripts and results in depth for this thesis we found that the wrapper might cause a buffer overflow by one byte if the input is exactly 999 characters long. Even though this might in theory cause a problem for the LFUZZER and AFL generated inputs that could reach this size (including the inputs from the AFL phases of PFUZZER and LFUZZER), we believe that chances are low that this influences the overall results. Such large inputs are not needed for our subjects and are not beneficial to generate, they would typically contain redundancies from smaller, already generated inputs and are likely not syntactically valid. Also, all tools run with the same wrapper, hence, this coding error can be seen as any other bug in the subjects.

subject we also describe the manual changes done, which are partially needed for pFuzzer and lFuzzer to function correctly. Details on the changes can be produced from the replication package in most cases [103]:

inih [14] parses INI files—the readme of the tool gives a vague description about the features inih implements, but no explicit feature list. The INI format is simple: it stores a set of key-value pairs. The key-value pairs are separated by a previously determined character—typically by an equal sign. The INI standard also allows sections which are started with an opening bracket and closed with a closing bracket, e.g. `[section]`.

Manual Changes: We added a main method including some information for inih to make it run and added a method to read input from the command line (up to 999 characters; for more the program exits with return code one). We also change one configuration in the header file to let the program stop on the first error.

csvparser [74] parses CSV files [70], another rather simple format—though the tool itself does not list which features of CSV are actually implemented. CSV represents a table, each row is also a row in the file, ending with a line break. Each row consists of an undefined number of columns, separated with a predetermined separator character (by default a comma, hence the name CSV: ***C**omma **S**eparated **V**alues*).

Manual Changes: We added the same standard input reader as for inih as well as a main method which parses the input, requires a header (otherwise almost all inputs would be valid), and prints all rows with their columns from the just read input.

cJSON [34] parses JSON [71] values. JSON, the *Javascript Object Notation* is one of the most commonly used formats for data transfer between programs and in the web. It has the simple base data elements *number*, *string*, *bool*, and *null* as well as the more complex elements *object* and *array*. An *array* is a list of JSON elements, separated with a comma and put into brackets (e.g. `[1,2,3]`). An *object* is a list of key-value pairs, separated with a comma (e.g. `{"A": 1, "B": [1,2,3]}`).

Manual Changes: Again, we added the standard input reader as for inih and a main function which calls the parsing code and checks if it returns a successfully parsed JSON value.

tinyC [78] parses a small subset of C [73]—though the syntax and semantics are only inspired by C. It contains code constructs like *variable assignments* (`a=b+5;`), *if-statements* (`if (a < b) {a=5;} else {a=6;}`), and two kinds of while constructs: *while-loops* (`while (a < 5) {a=a+1;}`) as well as *do-while-loops* (`do a=a+1; while (a < 5);`). A large difference compared to a real C program is that tinyC programs do not have variable definitions but the set of variables is pre-defined. In particular, every tinyC program has variables *a* to *z* pre-initialized with the value 0 at program start. Thus, every syntactically valid input is also a semantically valid input and the resulting program can be executed. Also, this subject is not only a parser, it is also an interpreter of the language, hence syntactically equivalent inputs can lead to different code coverage. It

may also happen that a fuzzer produces an input that contains an infinite loop, resulting in a divergence of the TINYC interpreter.

Manual Changes: We added the missing standard arguments to the main function (`int argc` and `char* argv[]`); they are not further used.

MJS [26] is a parser and interpreter for a subset of JAVASCRIPT (which has no specification on its own but is based on ECMASCRIPT [39, 111]). We will not list all features here as MJS is very feature rich and this will go beyond the scope of this document. In general, it supports the basic features of JAVASCRIPT like *loops*, *branching statements*, *a subset of JavaScript boolean and arithmetic operators*, *let expressions*, and a small *built-in API* containing functions that can be called from the MJS code. This means that the parser contains lots of features that can be explored by the testing tools, but it also results in a complex interpreter which supports a wide variety of semantical features for syntactically similar code. We will later see how this influences our results.

Manual Changes: Again, we added the standard input reader as for INIH and changed the main function to use the input from this reading method. We also added code to exit immediately with an error code if a parse error would be reported (instead of keeping parsing). This change was just done by adding program exiting to the *error report function for parse errors*, a typical point where one could add similar code for other subjects as well to make them easier to analyze for PFUZZER and LFUZZER. Also, we disabled error reports for errors that occurred during code interpretation, as those are semantic errors. All those changes were done to the best of our knowledge and understanding of the original code.

LISP *"An embeddable lisp interepreter [sic] written in C. [...] Scheme-like[[108]] (but not confined to) syntax."* [76]. It contains the core language features and data structures like *" if, let, and, or, etc."* [76]—we will not list all features here. The LISP syntax is mostly based on parenthesized expressions, every instruction is parenthesized and nested instructions are in nested parenthesis. It uses a prefix notation, hence the operator comes first and then the operands follow. Being one of the oldest languages, it got instantiated in different dialects, Scheme being one of them [108].

Manual Changes: We took a sample main function from a file from the repository (`lisp_i.c`) and adapted it slightly to report a parse error with exit code one, execution errors with exit code zero, and return with exit code zero in any other case (the original code always returns with exit code one which we would interpret as error).

Further Tool Adaptions For the evaluation, we also had to adapt LFUZZER and PFUZZER towards LISP: our prototypes might generate inputs for LISP that start with "`(#`", "`( '#`", "`( "#`" (and variations with additional whitespaces) causing the AFL instrumented version of LISP to crash [102]. Since AFL requires a valid seed input set (the program must return with exit code zero), it would not start if those inputs are given as seeds. Hence, we filter those seeds (for all subjects) if they are generated by LFUZZER or PFUZZER before giving them to AFL.[5]

[5]The seed input generation phase of LFUZZER never generates any LISP input that starts with "`( '#`", "`( "#`" (ignoring whitespace), hence we only added code to filter out inputs explicitly starting with "`(  #`".

Benchmarks One question might remain: why did we not choose any typical benchmarking set like FUZZBENCH [107] or MAGMA [61] which would also enable us to measure *bugs found*. First of all: the two mentioned benchmark sets were not released by the time *Learning Input Tokens for Effective Fuzzing* was created—i.e. when this evaluation was done. But more importantly, we built a research tool completely from scratch and it should be considered as a prototype with less compatibility as mature fuzzing projects built by many developers.[6] Our main goal of this work is to show that using comparisons in recursive descent parsers is beneficial for generating syntactically valid inputs from scratch. Thus, we put much effort in researching how this can be done and less effort into enabling programs written in different languages. Those benchmark sets typically contain subjects written in C++ and many subjects that do not contain a recursive descent parser, making it hard to enable LFUZZER and PFUZZER on those benchmarks (e.g. MAGMA [61] and FUZZBENCH [107] contain such subjects). Hence, we could only analyze a portion of the benchmark sets. Also, many fuzzers are based on AFL and as such can use the AFL eco-system [2, 99], making it easier for them to be integrated into the benchmark sets which are already supported by AFL (e.g. FUZZBENCH [107]). Using bug seeding tools like LAVA [35] would enable us to add bugs to the subjects we chose, but then again, our main goal is not maximizing the number of bugs found but showing that we can generate valid inputs for recursive descent parsers using dynamic tainting and generic heuristics. Hence, we created our own set even though it does not have a ground truth of known bugs.[7]

6.3 Research Questions

In this overarching evaluation we want to quantify how well the different stages of our approach work—i.e. *comparison usage for input synthesis*, *token extraction* for targeting parser with a lexing step, and the *combination of state-of-the art fuzzing with our token extraction and input inference*. The research questions are formulated along the lines of the objectives and contributions made in this thesis. Still, it is hard to draw a sharp line between the different objectives (e.g. syntactically diverse inputs already are semantically diverse to some extent). Hence, we decided to order the research questions not in the order of our objectives, but such that we add more and more depth to the analysis of the raw results. Thus, we propose the following research questions answering if we reached our objectives:

RQ1: Tokens Extracted analyzes the capabilities of the LFUZZER token detection to extract tokens from the subject under test in comparison to a simple search for string constants. We compare how many actual program tokens can be extracted if we extract all string constants from the LLVM bitcode in comparison to the tokens extracted with our dynamic token detection approach. With this we want to evaluate if the token detection of LFUZZER misses any tokens and also if it misclassifies values as tokens that are none. **This answers if we (partially) solved our Objective 2: can we extract tokens from the subject under test?**

[6] The GITHUB page of AFL ++ lists 131 contributors at the time of writing [2].

[7] With the techniques as described in our paper *"Systematic Assessment of Fuzzers using Mutation Analysis"* [58] it might be possible to solve such shortcomings in the future and augment a set of subjects with bugs.

RQ2: Coverage Achieved gives us a detailed look on how well the different tools perform in covering code. We want to specifically have a look at the benefits that result from combining the different techniques. Thus, we compare AFL with and without extracted strings, pFuzzer with and without a subsequent AFL run (we stop pFuzzer after it failed to find new coverage for more than 1000 generated inputs in a row[8]), and lFuzzer with AFL. As we already discussed, lFuzzer should not be considered as a standalone tool but rather as a preprocessing step, a preparation tool for other fuzzers (in this evaluation AFL), hence lFuzzer is not evaluated alone. The achieved coverage gives a hint on how much of the actual program code, and therefore also the semantic part of the program, is covered. **This answers if we solved our Objective 3: can we generate syntactically valid inputs with semantic diversity?**

RQ3: Tokens Used analyzes the capabilities of each tool (and tool combination) to generate inputs that actually use the tokens that are valid for the specific subjects. This analysis especially shows the syntactical diversity of the generated inputs. It is important to note that the used tokens only represent the different syntactical features of the underlying grammar, there might also be interesting feature combinations, resulting in semantic diversity, which cannot be analyzed with this number. **This answers if we solved our Objective 1 and the remainder of Objective 2: can we generate syntactically diverse inputs by analyzing the subject under test and is this also possible for subjects with a tokenization phase?**

Result Collection In general, for calculating the results we either used the output of pFuzzer or the seed input generation phase of lFuzzer (whenever only the generation phase was evaluated) or the inputs from the AFL queue (*"test cases for every distinctive execution path, plus all the starting files given by the user"* [158]) as well as the inputs from the AFL *hangs* folder for the AFL evaluations (including the AFL parts of lFuzzer and pFuzzer with subsequent AFL runs).[9] pFuzzer and the seed generation of lFuzzer output a valid input if it covers branches in the subject under test that were not covered by any value beforehand. In the different sections of the research questions we give more details on how exactly we used the information for generating the evaluation results.

As we already mentioned, for Lisp we had to filter some valid inputs before giving them to AFL as test inputs. We still consider those inputs when calculating the coverage for the lFuzzer and pFuzzer part of the pipeline. Hence, because the final coverage is solely based on the AFL generated inputs, we might put lFuzzer and pFuzzer in a slight disadvantage here—but therefore we have a fairer comparison to the other AFL runs which might not handle those inputs properly as well; only for pFuzzer alone we do not use the AFL results as baseline.

[8] We try to be as precise in counting the number of inputs for pFuzzer as for lFuzzer. Still, it might be that in some border cases a run might not be counted correctly. The overall impact of this incorrect counting is negligible.

[9] The crashes folder contains *"unique test cases that cause the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT)"* [158]—hence the respective inputs are not accepted on the subject under test and we need not to consider this folder for our evaluations on syntactically valid inputs. For pFuzzer and lFuzzer we also do not consider any inputs that they consider crashing, hence we have a fair comparison for all tools.

6.4 RQ1: Tokens Extracted

First, we evaluate the token extraction precision of the token detection phase of lFuzzer in comparison to the naïve approach of extracting strings from the subject under test—**partially answering if we solved our Objective 2 and can extract tokens from the subject under test**. We compare how many tokens are reported by lFuzzer and how many tokens can be extracted from the subject under test by searching for strings. Therefore, we read the code and documentation of the subjects, manually define for each subject a set of tokens as ground truth, and check how many of those are found by each approach and how many non-token values are found—strings reported as tokens which would not be recognized as such by the subject under test.[10] For this research question and research question three we use a handwritten lexer which uses the tokens we manually extracted. We expect the string extraction to be more complete (find most tokens of the underlying subject) as it has full visibility over the source code, but we also expect it to be less precise, as it might collect I/O strings like error messages. For this research question we ignore empty strings reported by the tools.

Static String Extraction For AFL For the string extraction the setup is straight forward: we compile the subject to one LLVM bitcode file in human readable form and then iterate over all string constants defined in this artifact. We chose LLVM bitcode as this is the common ground we are also using for our instrumentation and the bitcode contains all known string constants in individual global registers. LLVM also stores names of the subjects' C functions from the subject under test in string constants for mjs and Lisp. This is because the `__func__` feature of C is used to print the name of surrounding function as string [73]. To avoid using all function names from the source code which would result in a high noise ratio, as those are no tokens, our string extraction only uses global values that start with `@.str`, as those contain, as far as we understood the subjects' implementation and the generation of LLVM bitcode, the actual tokens plus some other non-token strings that cannot be easily filtered out.[11] In fact, the names are used in mjs for logging and printing of assertion locations, in Lisp it is also used for assertion location printing. For the extracted strings we ignore the `zeroinitializer`, which marks the string as empty [88] and we remove the trailing zeros at the end of the string constants, as they are needed for the subjects internally but not when using them as tokens. Furthermore, AFL requires the strings to be escaped when being used in a dictionary file [159], which is done by us by converting the escaped bitcode hexcode values to AFL hexcode values.

[10] For this manual extraction of tokens we tried to be as sound and complete as possible (extracting keywords, operators, internal callable functions, …), still this extraction is a threat to validity as we might miss some tokens or add a value which is not an actual token. Some subjects (like mjs) also include sub-parser which are used to parse strings for later interpreter runs (like JSON strings to create a json object). We ignore such sub-parsers that are not part of the main language.

[11] For completeness, we need to mention that the string extraction was done on another machine than the actual execution of the overall evaluation (simply because the string extraction can be done offline). This causes, in some rare cases, the extraction of a different string value for the aforementioned non-token strings that are compilation dependent, e.g. the strings generated for *assert* calls which, among others, contain the code which is used in the call. For example, on one system the value NULL is compiled to the string NULL while on the other system it is stored as `((void*)0)`. Those are not actual token values but noise that we would extract, hence the concrete value of the noise should have little to no influence.

Dynamic String Extraction With lFuzzer The string detection and extraction with lFuzzer is, in contrast to the aforementioned method, dynamic. Thus, we need to run the program with a diverse set of inputs that cover the lexical space of the subject under test, i.e. that covers all possible lexemes and extracts them. The input inference phase of lFuzzer does exactly this, it tries to build syntactically correct inputs from scratch by observing the program behavior, specifically the comparisons done during parsing. Hence, we get the set of possible token values that we have seen during the program probing phase *as a side product* from lFuzzer, as it needs to extract them anyway. We let lFuzzer run until it tested 1000 inputs without finding a new input that yields more coverage than the already found inputs before. This threshold is selected randomly as a rule of thumb as the search space of lFuzzer gets too large at such a point during fuzzing and has too many plateaus to be useful anymore. In fact, chances are very high that at this point most of the syntactical features are already explored and any remaining features are easier found by using a *state-of-the-art* fuzzer with less program analysis but higher execution speed.

For this evaluation we use the tokens reported by our approach as described in Section 5.2. In the presence of a lexical phase lFuzzer might not have generated inputs that cover all tokens, but it will, with a very high probability, have seen all valid tokens and put them into a dictionary for further use in a faster fuzzer with less program knowledge. This is due to the fact that if a lexer is present it generally checks an input token against all known tokens before rejecting it—that is the reason for the token learning phase (see Section 4.2.3) and the usage of those tokens for this part of the evaluation. pFuzzer does not explicitly extract tokens, as such we cannot evaluate its token extraction capabilities on its own, but in research question three (Section 6.6) we evaluate how many different tokens are used by each tool in the generated valid inputs, including the ones from pFuzzer.

Completeness In the first part we show how many valid tokens were found by the naïve string extraction and by lFuzzer. We define a token as valid if it would be parsable by the subject under test[12]. In fact, we created some generic lexer for this, containing a set of regular expressions that match the respective token values. For keywords we check that no invalid character is in front or behind the given keyword; for numbers, identifiers and strings we also map them to one token, such that only the presence of the token is counted, not every instance on its own.

Figure 6.1 shows how many valid tokens lFuzzer and naïve string extraction find in the different subjects. We can see that the data exchange formats CSV, INI, and JSON only have a small amount of tokens overall. This is because they are mostly used to pack data and, hence they only need a small amount of control characters to structure the data. For JSON there are also a few special values like `true`, `false`, and `null`. The programming languages on the other hand do not only need control characters, but also fixed keywords to expose pre-defined methods and semantics that are later used in a compilation or interpretation step. Hence, the amount of tokens is much higher for them.

[12] As found out by our manual analysis of code and documentation.

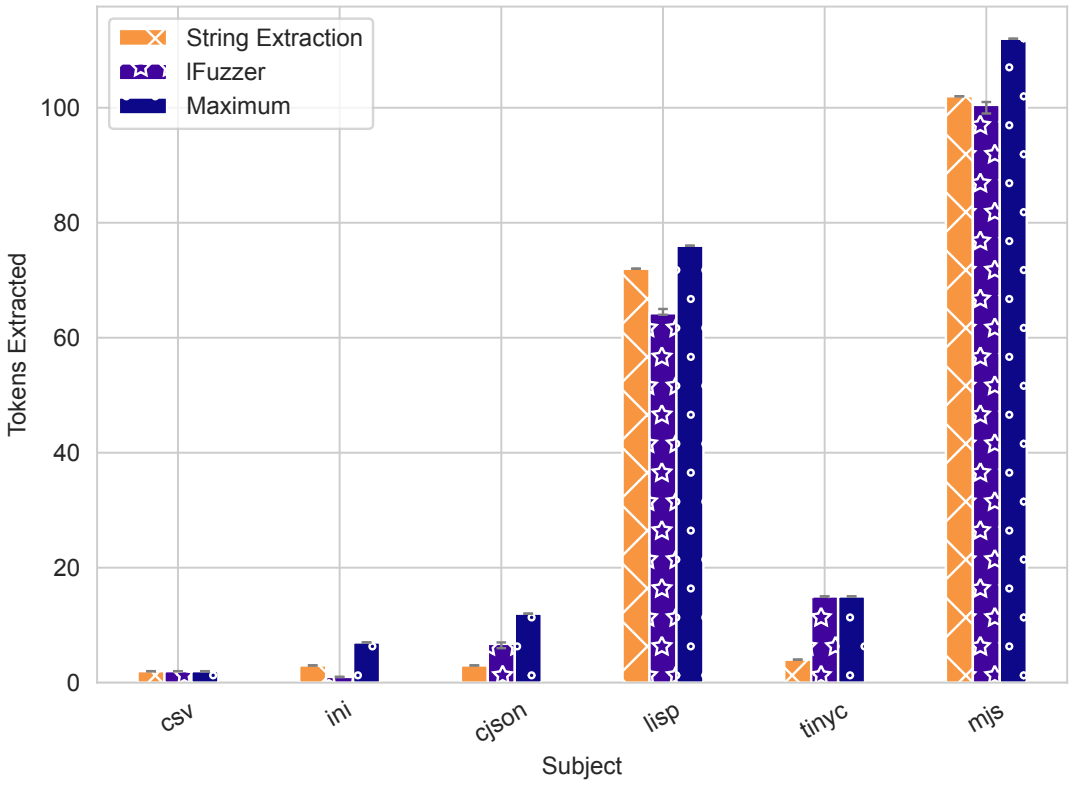


Figure 6.1: Valid tokens found by naïve string extraction as well as lFuzzer including maximal number of unique tokens that can be found.

For most subjects the number of extracted tokens is comparable between lFuzzer and naïve string extraction, which is expected, as the string extraction should over-approximate and find most if not all tokens the subject accepts. For cJSON and tinyC though, lFuzzer finds actually more tokens than the string extraction—a result that seems surprising at first. When looking at the missing tokens and combining this information with the extraction method, one immediately detects the reason for this outcome: *single character tokens*. While lFuzzer successfully detects them during program execution based on constants used in character comparisons done on input characters, naïve string extraction misses them because they are not encoded as string constants. In fact, such characters are compiled to integer constants, as a char is nothing more than a numeric constant, hence a comparison in C code like "`'(' == inp`" with `inp` being a variable containing an input character, would be compiled to a comparison of `inp` against the constant numeric value 40.

On the other hand, for INI, mjs, and especially Lisp lFuzzer finds less tokens. The reasons can be manifold, most likely though it is because of the missing lexer for INI and more complex lexers for Lisp and mjs. While typically tokenization is done with almost no context information

and close to the textbook in the lexer, some tokens only appear later in the execution—for MJS and LISP those tokens are only evaluated in the interpreter phase of the subject. Due to the dynamic token detection method of LFUZZER, our approach needs to *"see"* a lexeme during the learning phase to report it. Hence, if the token generation is not correctly detected, the tokens are also not correctly learned and thus not reported. Finally, the error bars indicate that the LFUZZER string detection is stable—over all 4 runs the number of extracted strings is close to the average. The naïve string extraction is deterministic and as such does not have fluctuating results by design.

Interestingly, for most subjects the number of extracted tokens is already close to the maximum number of tokens we found. This means, especially for the token extraction of LFUZZER that our analysis is indeed able to extract and learn most of the tokens the program uses. As such, the later phases of the approach should profit from from this high number of found tokens and respectively generate diverse inputs.

> LFUZZER can successfully extract about 84.6% of all tokens from the subjects under test.

Precision Besides the number of correctly found tokens, it is important to also keep the number of non-tokens low, because those tokens should later be used as a dictionary in fuzzing or in combination with other approaches. Hence, if a fuzzer uses the extracted tokens, it might need more trials to find the correct values to inject. Therefore, we also checked for each method how many strings are reported that would not be detected as a token by the subject under test.

In Figure 6.2 we present the number of invalid tokens reported for each subject. With increasing code size and complexity the number of wrongly extracted strings for naïve string extraction likely increases as well. This is because more complex parsers and subjects also tend to have better error handling, resulting in a more diverse set of error messages. Another factor are format strings that are used all over the code and also end up as string constants. Hence, for CJSON a low but noticeable amount of invalid strings is extracted, for LISP and MJS the number of invalid tokens is also much higher than for LFUZZER.

For CSV, CJSON and TINYC LFUZZER reports a large amount of non-token strings. For CSV and CJSON the reason is likely a misclassification of random code as tokenization code, because those subjects do not have a tokenization phase. LFUZZER always tries to detect tokenizer code with its heuristics, and typically if tokenization code is present the heuristics are strong enough to filter out any code that is not tokenizing. If no tokenizer is implemented though, this noise cannot be filtered and some of the code is still assumed to be tokenizing, yielding random strings as tokens.[13] For TINYC the lookaheads needed to parse numbers and identifiers cause our token detection heuristics to combine too many characters to one token. In detail, our algorithm is

[13] Subjects without a tokenizer typically have a less complex underlying language, in Section 6.5 and Section 6.6 we will see how this extracted noise influences the fuzzing performance down the pipeline.

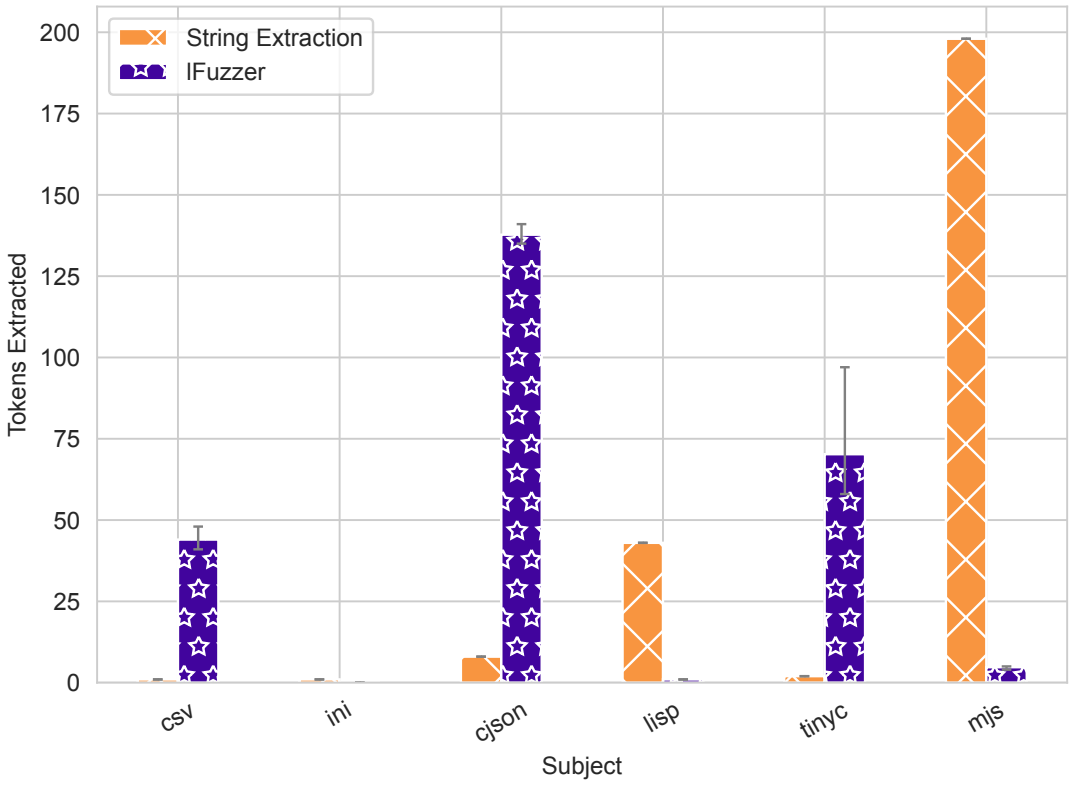


Figure 6.2: Invalid tokens found by naïve string extraction and lFuzzer.

designed to combine all characters accessed between two parsing steps to be combined to one token. While we try to filter out such lookaheads as much as possible (see Section 5.2), it may happen that the additionally accessed character is appended to the token, resulting in noise.

The number of misclassified strings deviates more compared to the valid token results between different runs for lFuzzer, especially for tinyC. The reason is simple: while for valid tokens any instance once extracted results in a token hit, we count every non-token string as one invalid token. Still, lFuzzer is able to keep the number of misclassifications low if there is a tokenization phase present, resulting in a 28 percentage points higher precision compared to naïve string extraction for those subjects under test.

> lFuzzer has a 28 percentage points higher precision compared to naïve string extraction when looking at subjects with a tokenization phase.

Summary In Table 6.2 we can see that if the string extraction has, per subject, a higher precision it is much higher, but for higher recall values it is almost on par with lFuzzer. The

reason for this is likely that if lFuzzer runs into imprecisions during token extraction, chances are high that many different wrong alternatives for a token value are found. The fact that for some subjects the precision is very low and for others it is very high supports this claim: if the token extraction works as expected we get a precise set of tokens. Regarding the recall, we can see that lFuzzer performs better on cJSON and tinyC, whereas the string extraction performs better on INI, Lisp, and mjs. Considering, that lFuzzer must detect the tokens during execution while the other approach statically extracts them, it is surprising that lFuzzer often finds at least as many tokens as the naïve string extraction.

Subject	Precision		Recall	
	lFuzzer	String Extraction	lFuzzer	String Extraction
CSV	4.35%	66.67%	100.0%	100.0%
INI	100.0%	75.0%	14.29%	42.86%
cJSON	4.67%	27.27%	56.25%	25.0%
Lisp	98.47%	62.61%	84.54%	94.74%
tinyC	17.6%	66.67%	100.0%	26.67%
mjs	95.49%	34.0%	89.73%	91.07%

Table 6.2: Precision and recall on extracted strings regarding their token validity per subject.

In Table 6.3 we see the aggregated precision and recall for the naïve string extraction and lFuzzer for *tokenizing subjects*. The results in this table are surprising: while we expected the precision of lFuzzer to be higher, as it is more focussed on the tokens in the subject under test, we expected the recall of lFuzzer to be lower than for string extraction as it might miss some features of the lexer due to its dynamic exploration. Still, both approaches are on par when it comes to recall, meaning they find a similar amount of actual tokens in the subject. With lFuzzer though, the noise, as indicated by the precision value, is much lower, hence on average we produce a more focussed token set which can be used as a dictionary for fuzzing. Thus, *we solved one part of our second objective*: **we are able to analyze the tokenizer of a subject under test and extract a precise set of lexemes**. In research question two (Section 6.5) we see how this influences (in combination with the generated seed inputs) the achieved coverage while fuzzing and in research question three (Section 6.6) we also quantify how well our dictionary and seed input generation works when it comes to using the tokens of the language, i.e. we evaluate how well the syntactic features of the underlying language can be covered.

Tool	Precision	Recall
String Extraction	42.3%	87.7%
lFuzzer	70.3%	88.5%

Table 6.3: Precision and recall on extracted strings regarding their token validity for subjects with a tokenization phase (tinyC, mjs, Lisp).

6.5 RQ2: Coverage Achieved

Setup One of the main strategies for evaluating fuzzers is determining achieved coverage on real world subjects. While we think that especially in the context of fuzzing systems with grammar-based inputs coverage may not be the best option to evaluate fuzzer performance, it is still a valid proxy to determine bug finding capabilities. At least the number of found bugs generally correlates with the achieved coverage of a fuzzer, even though the agreement in ranking and superiority between the fuzzers might be more complex [20]. Thus, we evaluate the **branch coverage** over time achieved, **answering if we solve Objective 3 and build not only syntactically valid inputs but also semantically diverse inputs—using code coverage as a proxy metric for this part of the evaluation.**

We evaluate AFL with and without a dictionary, pFuzzer standalone and with a subsequent AFL phase, and lFuzzer which is by design a combination of the original pFuzzer strategy including tokenizer optimizations and a subsequent fuzzing phase represented by AFL in this case. pFuzzer and lFuzzer switch to AFL after 1000 iterations without a new input that covered code never covered before. We only count coverage for inputs that are valid (the subject under test has a return code of zero for those inputs or timeouts[14]), as we want to find out how well the tools produce semantically diverse but syntactically valid inputs.

The inputs themselves are taken from the AFL *queue* and *hangs* folders (for the AFL experiments and the AFL phases when evaluating pFuzzer and lFuzzer) and for the seed input generation phases from the valid inputs reported by pFuzzer/lFuzzer. For the results of our techniques in combination with AFL, we first collect the coverage for the pFuzzer/lFuzzer phase alone and then iterate the AFL inputs in order of time and count the coverage once the combined coverage of the AFL inputs is larger than the coverage of the pFuzzer/lFuzzer inputs, adding the reported runtime of pFuzzer/lFuzzer to the input generation time of the AFL input. The timestamps for the coverage evaluation are given for AFL inputs by the generation time of the file (the first generated file marks the starting time) and for the lFuzzer/pFuzzer inputs by the tool itself (the time when the input was written to the output file for valid inputs). We use `gcc -fprofile-arcs -ftest-coverage` to compile the subjects for coverage collection[15] and gcovr [141] to extract the generated coverage information for further usage in our evaluation pipeline. The presented values are rounded to one decimal; rounding half up. Hence, for $x \in [0.55, 0.65[$, x is rounded to 0.6; for $x \in [0.45, 0.55[$, x is rounded to 0.5.[16] Due to rounding errors it might happen that different machines yield slightly different results which may cause different rounding results. For this evaluation, all results that are directly compared were also generated on the same machine.

[14] For technical reasons we cannot measure the coverage for inputs that timeout. We run them anyway under instrumentation for completeness.

[15] For completeness, we mention that the coverage collection compilation for the non-AFL phases of pFuzzer and lFuzzer additionally used the option -DCOVERAGE which is an artifact in the compilation script which, to the best of our knowledge, has no further meaning to our compiler.

[16] Specifically, even if a result of 0.549999999 is printed by our evaluation scripts, we use the value 0.5 in this document.

CSV And INI In Figure 6.3 we show the coverage over time for CSV and INI subjects for all tools, the x-axis is logarithmic to balance out the fact that the fuzzers tend to flat-line the longer they are into the fuzzing run. For CSV and INI we can see that all tools perform well, the differences are small and only visible in the first few minutes of the run. This is expected as both subjects only have a very rudimentary underlying format, as such AFL can easily cover the syntactic features which are guarded by one-character tokens. lFuzzer, which tries to analyze the subjects first, is in a disadvantage, the simple trial and error of the vanilla version of AFL is already sufficient and the missing analysis phase makes AFL more efficient. For CSV a simple input consisting of one character is already sufficient to achieve some initial coverage as we see it in the graph for lFuzzer. For INI lFuzzer cannot generate any valid input, likely because it tries to identify a lexer and is misguided during input generation. Still, since both formats are comparably easy, AFL can generate sufficient inputs in its fuzzing phase, bringing all tools to comparable results. pFuzzer on its own is also not able to find all features of INI, having less coverage than all tools that use AFL.

cJSON A more interesting format is JSON, implemented by the cJSON subject, which does not only contain single character control values but also actual keywords. From the graph in Figure 6.3 we can see that the coverage over time increases much more diverse, with AFL_Dict having the most coverage (20.2%, compared to pFuzzer + AFL having 20.1%, lFuzzer having 19.9%, AFL having 18.3%, and pFuzzer having 14.6%). AFL_Dict is aware of the existing keywords and coverage is a sufficiently strong indicator for JSON subjects to show the progress towards valid inputs, thus it performs best on cJSON. pFuzzer is able to cover the code of cJSON very fast, as it is designed to work on token comparisons directly and cJSON does not have a tokenization phase. But only with the help of AFL it can cover more code beyond the basic syntactic features of cJSON (like triggering a portion of the code that handles the conversion of *UTF-16* to *UTF-8* literals) and result in a high coverage. AFL alone performs similar but worse than AFL_Dict, likely because AFL_Dict already knows the keywords needed to generate valid inputs. lFuzzer on the other hand is not able to extract tokens properly due to the missing tokenization phase, which results in a slow increase of coverage at the beginning but once AFL is started with the found seeds, the coverage increases much faster compared to AFL, still resulting in slightly less overall coverage in the end.

Lisp For Lisp we can immediately see in Figure 6.4 how a missing optimization for tokenizing subjects lowers the fuzzing performance. pFuzzer, not having any code to handle tokenizers performs worst, likely because it can only guess how to combine the different lexemes it detects in the lexer. It does not have any information from the parser and needs to fall back to using heuristics during input inference. AFL needs to guess the different lexemes correctly, likely spending a large amount of time in the lexer, trying to find the next valid token. pFuzzer + AFL can combine their strengths to some extend: pFuzzer finds simple but seemingly sufficient seed inputs that can then be used by AFL to fuzz better. lFuzzer is able to extract most of the tokens from Lisp, giving AFL a beneficial dictionary to use during fuzzing. This takes time, resulting in lFuzzer + AFL to be slower than AFL_Dict in achieving coverage—the final coverage is similar though. For this subject, the lexing and parsing phase is much more involved

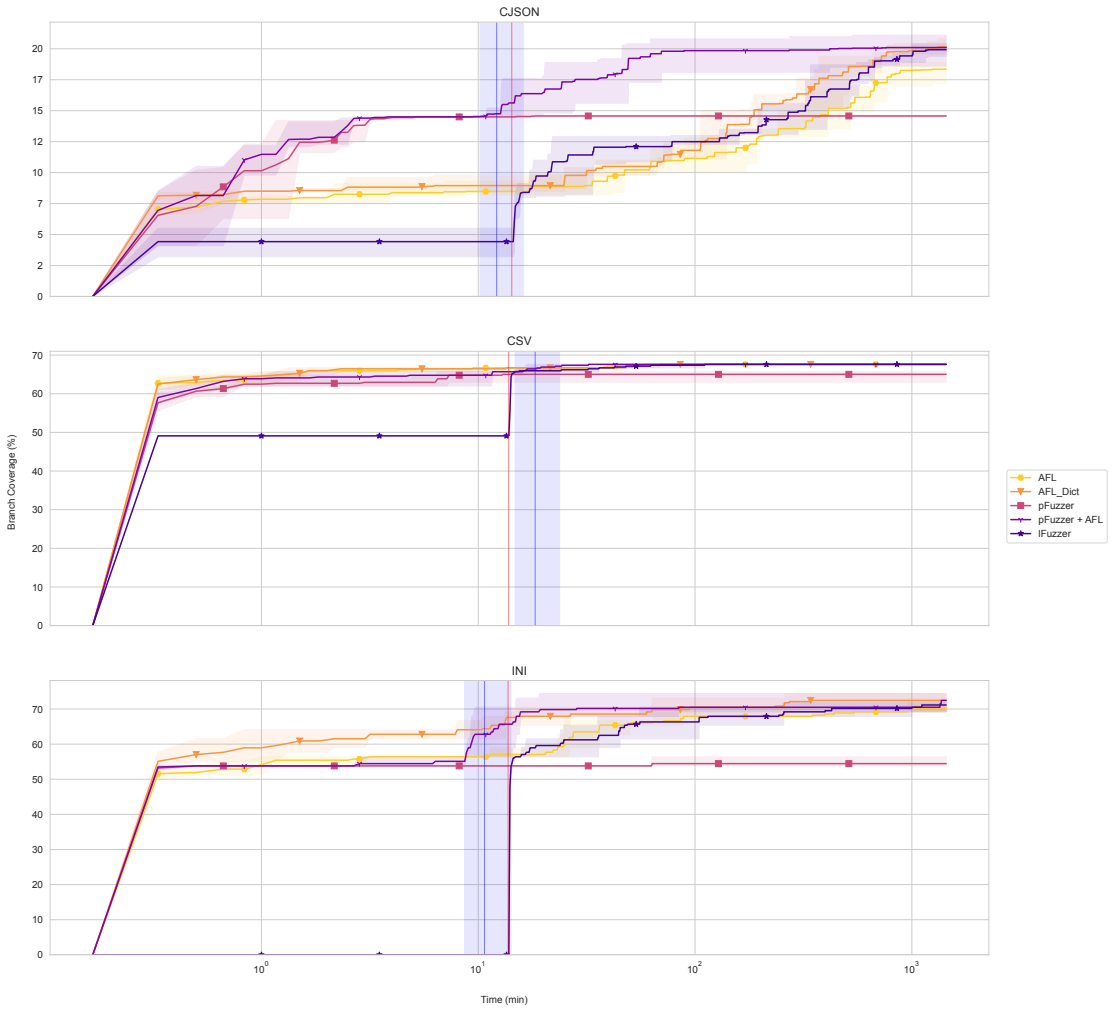


Figure 6.3: Coverage over time for valid inputs for the subjects cJSON, CSV, and INI. The vertical lines indicate the point in time when lFuzzer (red) and pFuzzer (blue) switched to AFL—represented as a range with an average point in time as a solid vertical line.

and not as close to the textbook as for the other subjects, resulting in only a small amount of extracted seed inputs. Especially the parsing step involves additional steps beyond simple token matching, breaking the link between lexeme and token for our approach. Still, this proves the advantage of our symbiotic approach: even if one part has a low performance, the other part might still perform well, making the combination of both approaches much better than each tool individually. AFL_Dict seemingly profits from the already extracted tokens over time and is able to constantly increase coverage by combining the different tokens in new ways.

tinyC lFuzzer is designed to work well on parsers that are implemented close to the textbook as we assume that apart from some details most parsers are designed this way. tinyC, being an

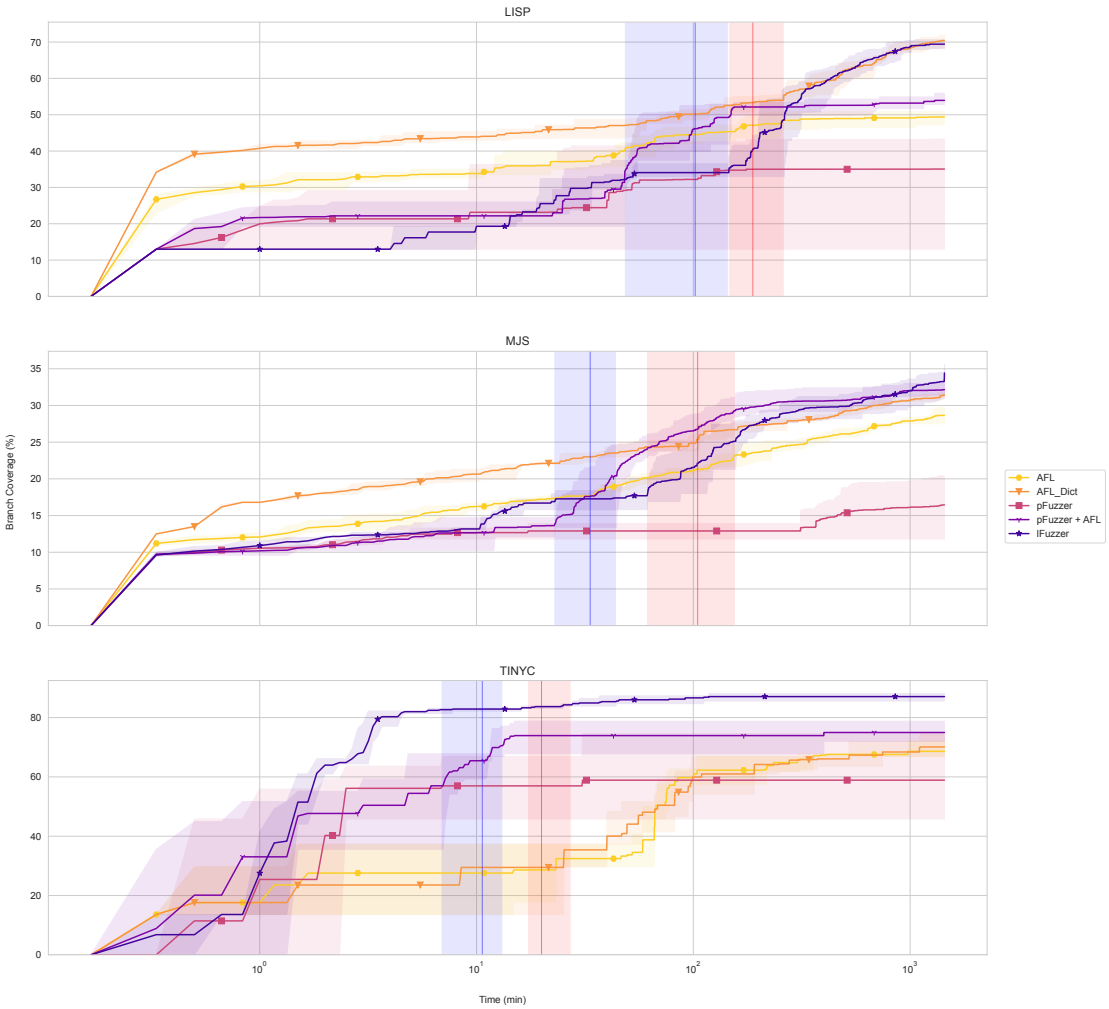


Figure 6.4: Coverage over time for valid inputs for the subjects Lisp, mjs, and tinyC. The vertical lines indicate the point in time when lFuzzer (red) and pFuzzer (blue) switched to AFL—represented as a range with an average point in time as a solid vertical line.

educational sample implementation, follows this coding style very strictly, which lets lFuzzer analyze it in depth, resulting in a proper dictionary and a well selected set of seed inputs which already covers most features of tinyC. In Figure 6.4 we see that the missing parts are easily filled by the subsequent run of AFL, resulting in 17 percentage points more coverage than AFL_Dict, which performs third best. pFuzzer is able to use its heuristics to overcome some of the missing information induced by the tokenization phase (again, the order of lexemes needs to be guessed and approximated with the heuristics and coverage, similar to AFL_Dict). In combination with a subsequent AFL run that builds up upon the extracted seeds, the pFuzzer + AFL combination performs better than AFL_Dict. Interestingly, AFL and AFL_Dict perform similarly well. The reason for this might be that the known keywords are of no use for AFL_Dict

as also the combination and order of those keywords needs to be correct, a task that is hard to solve by using path coverage alone. Finally, pFuzzer alone has a high deviation in the results: the minimal coverage achieved is way below AFL while the maximal coverage achieved is on par with AFL_Dict. The reason for this is the reliance on the heuristics instead of informed decisions as in subjects without a tokenization phase. Depending on the order of generated inputs, the search space forms differently, resulting in early plateaus for some runs which are not escaped anymore.

mjs For our last subject, Figure 6.4 shows how the *combination of token extraction and seed input generation with subsequent fuzzing* can improve the overall fuzzing performance beyond what is possible for each approach individually. pFuzzer again struggles with the tokenization phase, resulting in the lowest overall coverage. AFL alone cannot build all needed keywords, hence it can only cover some portion of the subject under test, but would need much more time to also guess the missing keywords to cover more code (from Section 6.6 we can see that AFL does only cover a little portion of larger tokens in its generated valid inputs for mjs). Likely, because AFL_Dict has knowledge about the keywords in the subject (plus some noise), but has no seed inputs, it performs better than AFL alone, but still misses some coverage the two best performing tools achieve. pFuzzer + AFL perform second best, pFuzzer extracts a set of seed inputs which only covers a small portion of the actual program features but already gives a great starting point for further mutations and recombinations, leading to a high code coverage. Finally, lFuzzer is able to do both: extract a diverse set of tokens and generate a diverse set of seed inputs, resulting in the most coverage during the overall fuzzing sessions—revealing the power of the combination of token extraction, seed generation, and fast fuzzing.

> On all subjects, lFuzzer achieves on average 2.9 percentage points (55.4% vs. 58.3%) and up to 17 percentage points (on average for tinyC) more coverage than our baseline AFL_Dict.

Summary The results for CSV and INI show that for subjects with simple formats a fast approach like AFL implements it is already sufficient. For more complex formats like JSON domain knowledge is helpful to achieve coverage faster, but naïve approaches can still achieve a similar coverage. When it comes to subjects that are syntactically simple but semantically rich, like Lisp, it is especially important to know the building blocks of the language: *the lexemes*. For subjects that are syntactically rich but have a low amount of keywords on the other hand it is much more important to use domain knowledge for generating the complex syntactic features, lexeme knowledge is not sufficiently helpful (as we can see when looking at the tinyC results). Finally, for subjects that combine a rich set of keywords, syntactic features as well as semantic features, the *combination of token extraction, seed generation and fast fuzzing is the most promising in terms of coverage achievement.* **Thus, we also solved our third objective: in combination with a *state-of-the-art* fuzzer we can generate syntactically valid inputs that are also semantically diverse—which is shown by the high coverage on real world subjects.**

6.6 RQ3: Tokens Used

With our coverage analysis we have shown how much code the different fuzzers can reach. While this is a proxy for bug finding capabilities, it leaves out an important aspect in the domain of subjects with complex input formats: *language coverage*. In this research question we want to find out how well the different tools use the lexemes of the underlying language of the subject under test—**answering not only if we solve Objective 1 (generating syntactically diverse inputs), but also portions of Objective 2 (analyzing the tokenizer for on par input generation compared to subjects under test without a lexer).** Only if a lexeme is used in an input, the belonging language feature can be covered and tested, thus the more different lexemes are used in the valid inputs, the more diverse is the feature set that can be tested. Token coverage and code coverage may correlate to some extend—for every token there is code to handle the token—but it may well be possible to cover a lot of code with just a few language features. Some syntactic features can have complex underlying semantics, hence they could activate a large portion of the code. Thus, to ensure that the fuzzer is able to pass one of the first roadblocks, the syntactic parsing stage, we also need to check the language coverage.

Setup In theory it would be best to **parse the generated valid inputs** with their respective grammars. Unfortunately, most of our subjects do not come with a formal grammar. If programs would typically come with a machine-readable grammar we would not conduct this research but just rely on input models. Thus, we built a generic lexer which *greedily* tries to match token after token in the inputs.[17] This lexer gives an approximation of the actual lexemes used.

We use the same regular expressions as in research question one (Section 6.4) to lex the inputs. Each lexeme is only counted once per subject and as before: strings, identifiers, and any other value that maps to one token is also only counted as one. Strings are defined as having length two, identifiers and numbers have a defined length of one[18]. We only count lexemes of length greater three (and all that have variable length like strings and identifiers), as lexemes of smaller sizes are easy to guess for any fuzzing approach. While it is certainly important to also have knowledge about smaller tokens, many of them serve as control characters and do not actively enable semantic features of the language (e.g. a parenthesis controls how different arithmetic or boolean information belongs together, but a keyword like `while` might actually activate a feature in an interpreter to run code in a loop). Thus, they are typically important to form valid inputs (which we already have at this point), but less important to trigger semantic features.

CSV And INI Figure 6.5 shows the number of lexemes with length greater three used in the valid inputs each tool generated for each subject. For CSV and INI there are no lexemes larger than 3 characters, hence the results for those are empty.

[17] For inputs generated by AFL we chose `latin-1` as encoding when reading files, because with `utf-8` we experienced decoding issues in our Python evaluation scripts. This approximation should be sufficient, because the characters used in valid tokens for our subjects should be correctly decodable with this method. The files containing the inputs generated by the seed generation phase of lFuzzer and pFuzzer are encoded in `utf-8`, hence we stick with `utf-8` for them.

[18] For valid CSV inputs we typically count values that are not a comma as identifier (even quoted CSV values).

cJSON AFL_Dict, pFuzzer, and the combination of pFuzzer and AFL perform equally well on cJSON in terms of lexemes used and use all possible lexemes. The reason for this is that using those larger lexemes is easy if they are known to the fuzzer, they need not be embedded in complex syntactic structures. AFL_Dict gets the keywords from the string extraction, pFuzzer recognizes them while analyzing the subject under test. AFL though is not able to generate such large keywords *out-of-thin-air*, as they have at least four characters which would need to be guessed correctly at once (they are often checked via *strcmp()*; hence AFL cannot detect a gradual improvement in coverage). Furthermore, since lFuzzer expects a tokenization phase, often it does not recognize the lexemes as tokens and the subsequent AFL phase cannot generate them as discussed above.

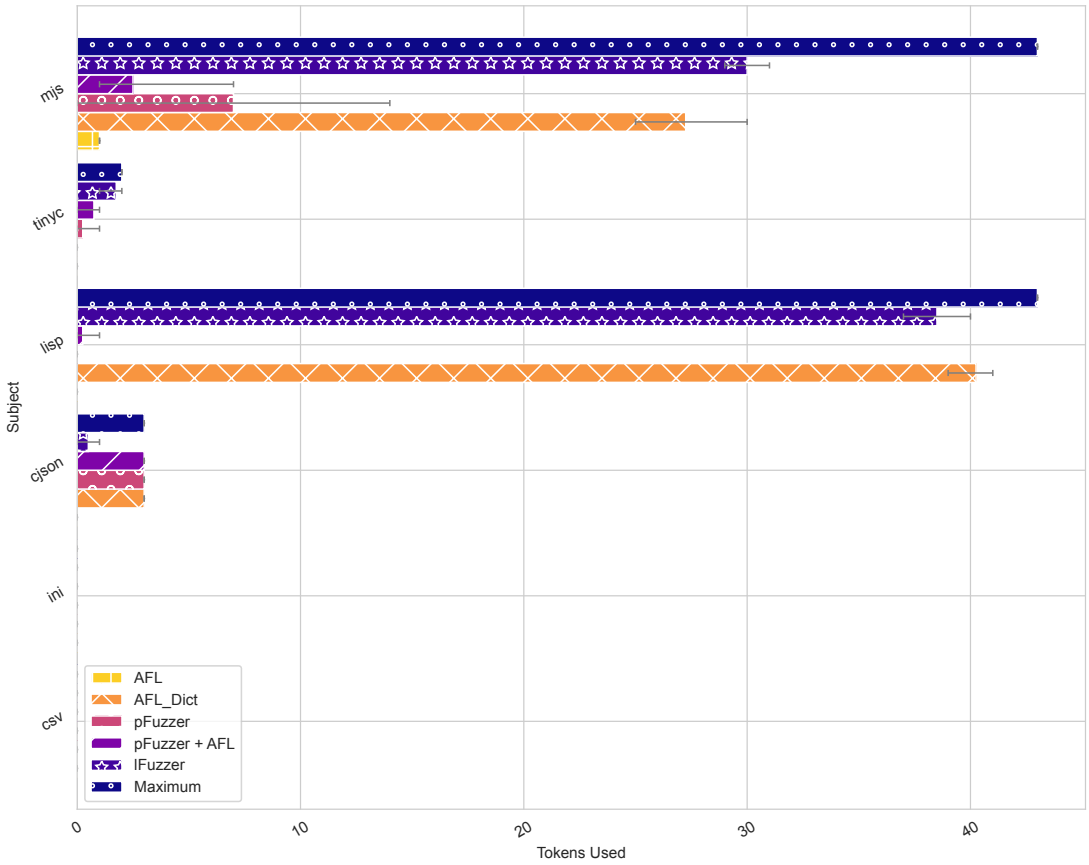


Figure 6.5: Amount of valid tokens with length greater three in the inputs generated by each tool including maximal achievable number of found tokens with length greater three.

Lisp Here, the picture changes slightly: while AFL is still not able to generate larger keywords, AFL_Dict and lFuzzer use a similar amount of lexemes in valid inputs, close to the maximal amount of tokens that could be used. The reason for this is that lFuzzer, as discussed before,

cannot generate a useful seed set for AFL to use, but it is able to extract the tokens from Lisp, hence once the AFL phase starts it has similar information as AFL_Dict. Since the syntactic properties of Lisp are less complex, AFL performs well in using most lexemes in valid inputs. For pFuzzer, also in combination with AFL, the picture changes: pFuzzer cannot generate a diverse set of inputs, it misses many keywords. Hence, for Lisp for the subsequent fuzzing session with AFL there are neither useful seed inputs nor keywords, thus it performs similar to AFL alone (which is also reflected in the coverage as we can see in Figure 6.4).

tinyC AFL and AFL_Dict are not able to generate any input with larger keywords for tinyC, likely because those larger keywords can only be used in complex syntactic structures (e.g. a `while` loop also needs a condition and a body). pFuzzer on the other hand is able to use its heuristics to generate such complex inputs at least in some runs even though it is mostly *"blind"* beyond the tokenization phase as most heuristics and the dynamic tainting do only work up until the tokenizer. pFuzzer in combination with AFL performs on average better than pFuzzer alone, mostly because of the non-determinism of pFuzzer, which results in more runs having a lexeme with more than three characters. lFuzzer can show it's full potential here: most of the time it is able to generate a diverse set of valid inputs using all tokens with more than three characters of the tinyC language.

mjs All tools are able to use at least some lexemes of mjs, but the distribution is similar to the ones in the other subjects. AFL alone uses the least number of keywords, primarily because it typically has to generate them without gradual improvement, hence it is blind for a lexeme until it found one and possibly placed it at the correct position. pFuzzer alone uses more tokens than pFuzzer with AFL, the reason might be that pFuzzer alone runs for 24 hours and thus has a better chance to use more lexemes it already saw in the lexer because it can use them together with its heuristics also later in the run while AFL is not able to generate those keywords if they are not already known. AFL_Dict has information about the existing lexemes and is able to put them into valid inputs successfully, as in many cases the syntactical complexity is low and AFL can make gradual progress while generating valid inputs. lFuzzer extracts a large portion of the existing keywords and generates a set of seed inputs that already provide some basic structure. In the subsequent fuzzing phase, AFL can use the given dictionary and seeds to replace some already used keywords with others, recombine existing inputs, and in some cases also generate new syntactic features. Hence, the combination of our initial program analysis with fuzzing produces the best results, better than AFL_Dict on average.

> lFuzzer and AFL_Dict are both able to use a large number of tokens (weighted average of 77.7% and 77.5% of all tokens with more than three characters) in valid inputs.

Summary When it comes to tokens used in valid inputs, it is crucial to know which tokens can possibly be used in the subject. While often those tokens are not part of complex structures, and as such simple valid inputs with a diverse set of tokens can be generated well enough by

fast trial and error approaches, in some cases domain knowledge is of use. For Lisp and mjs the number of used tokens is very similar for AFL_Dict and lFuzzer, mainly because most tokens can be embedded in simple syntactic structures. When it comes to complex structures and just a few tokens, as in tinyC, we can see that AFL_Dict actually struggles with generating those values, even though the lexemes are known to it. We conclude: if syntax as well as token knowledge is needed, only lFuzzer can provide the needed information for successful fuzzing. **Thus, we solve Objective 1 and Objective 2: we can generate a diverse set of syntactically valid inputs—no matter if the subject under test parses without or with a lexer.**

6.7 Threats To Validity

In this section we list the combined threats to validity coming from the concepts introduced in Chapters 3 to 5 as well as the evaluation conducted in this chapter.

Restriction to left to right recursive descent parsing.

First and foremost, we are bound to a relatively strict left to right parsing technique, not allowing larger backtracking features as they might slow down the input generation process too much. This obviously opens the question if this holds for recursive descent parsers, the parsing technique we are targeting. Let us go one step back here and look at the grammars those recursive descent parsers represent: context-free grammars. One feature when parsing context-free grammars is: they do not need context, each and every part of the grammar is independent from every other part. In terms of parsing this means every input character can be parsed out of context of any previous or upcoming parsing steps—only the current parsing step is important.

Hence, in theory no recursive descent parser needs to backtrack: once a character was consumed (compared last), there is no need to ever access it again, it could have already been determined as valid or invalid.[19] While this is true in theory, real world implementations of recursive descent parsers do not necessarily follow this theory: sometimes it is easier to just read later characters first and then go back to verify the suffix, sometimes they actually do not represent a context-free grammar fully but are slightly context sensitive, creating a need for backtracking.

We did not see this behavior much in our test subjects and if it happened the impact apparently was small. Thus, we believe that this restriction can be ignored for now, especially as other problems cause larger fuzzing roadblocks, but acknowledge that our approach and also our evaluation certainly do not generalize to the implementation style of all real world parsers.

Generality of input inference and code analysis heuristics.

Another question is how much impact do the heuristics of pFuzzer and lFuzzer have on the results and how generalizable are they? This question can be answered as follows: we

[19] In Section 2.1.1 we detail some restrictions to the underlying context-free grammar that are needed to avoid backtracking while parsing.

present a prototype implementation and a very new idea that was never tested before in the field of parsers. Hence, we are sure that there will be better techniques in the future and better heuristics to generate syntactically valid inputs from scratch—for now we just wanted to give a proof-of-concept. For each value and each part of the heuristic we gave a general explanation based on typical recursive descent parser implementation styles to clarify and substantiate the reasoning behind the decision of choosing it. We believe (and have shown in our evaluation) that the heuristics we chose are a good starting point to generate valid inputs from scratch for non-trivial input languages and parsers.

Apart from the heuristics used to generate valid inputs lexeme by lexeme, our pattern matching algorithm to detect tokenizer code adds further heuristic based decision making. We cannot claim that we will cover all different tokenizer implementations one can imagine, but we try to incorporate the most common ones: a lexer needs to read input characters, it needs to compare the characters against the token alternatives it has, and it needs to provide some information about the token to the parser. Thus, we believe that the patterns we designed already follow the typical design concepts. Furthermore, the *Dragon Book* [3] gives some design patterns which are partially covered by our patterns, hence we believe that our detection algorithm covers the main portion of lexers.

Close to the textbook coding style is beneficial for input and lexeme inference.

Furthermore, we rely on good coding style patterns, and even though they are very basic concepts, not all programs might follow them. For parsers without a lexer we mostly depend on comparisons in the parser which are inevitable when writing a recursive descent parser and thus in general exist as expected. When analyzing a parser with a lexer we need a clean code that well divides parsing and lexing to reduce noise in the detected token comparisons. This clear division and close to the textbook implementation of lexers and parsers is not always present in subjects. Hence, we cannot generalize our ideas to all subjects but believe that any programmer trained with the basic concepts of software design who applies modularization to the program parts writes sufficiently clean code that is analyzable with our approach. Additionally, the *Dragon Book* [3] recommends the clear division of lexer and parser.

Restriction to subjects written in C.

We limit the selection of subjects to programs that are written in C. While this is a restriction only introduced by the proof-of-concept implementation of our tool and not a general restriction of the approach, it needs to be quantified how much impact this decision has. We are requiring an instrumentation which makes it possible to taint track input characters through the program execution and report comparisons on characters as well as on strings. While there might be programming languages in which this is not possible, dynamic taint tracking is feasible in most cases. AutoGram [65] uses a taint tracker for Java and the tainting engine Pirate [151] is based on QEMU [122] and as such runs on a virtual machine.

Once such a taint tracking engine is implemented, querying taints on comparisons as well as querying standard library function calls is possible. In theory, one could also

implement the whole pFuzzer or lFuzzer pipeline for binaries as well, especially the byte level comparisons of characters should be detectable. Calls to standard library functions though might not be easy to detect on binary level, creating a finer grained comparison trace (as we would need to taint track the standard library as well) and thus increase runtime. Hence, we are convinced that our approach is generic enough to be applicable to subjects written in other languages than C and also in a limited way to languages at a *"lower level"* than LLVM bitcode.

Manual test subject selection.

We selected the test subjects ourselves based on the criteria detailed in Section 6.2 and decided against using already prepared fuzzing benchmarks for the reasons listed there. Even though we tried to pick a diverse set of subjects implementing different features of context-free grammars and different complexity levels, we cannot ensure that our approach generalizes to all programs that parse an input with a recursive descent parser. While we tried to use heuristics and methods that we expect to generalize to all recursive descent parsers, we cannot guarantee that no overfitting happened. Still, as explained above, we justify the different parts of our tools and thus expect them to be valid for a broad set of programs.

Furthermore, we acknowledge that this is just a prototype implementation and the first idea of its kind to infer syntactically valid inputs *out-of-thin-air*. Established fuzzers like AFL still struggle generating such inputs without prior knowledge (e.g from dictionaries or sample inputs that cover the different features). Hence, we are still far away from generating a full C or Java program with fuzzing *out-of-thin-air*, but research not only exists to answer questions, it should also open new questions and show alternatives to the known paths. Thus, we believe that our work sets a foundation for future research in the direction of fuzzing programs with complex input formats.

The results from our evaluation already indicate that our approach is promising and able to generate inputs for parsers that were hardly fuzzable for the *state-of-the-art*. To sum it up: we cannot claim full generality with our test subjects, but we show that it is possible to use comparisons from program executions, analyze tokenizer code, and propagate taints to token comparisons in the parser to build syntactically valid and diverse inputs from scratch for recursive descent parsers parsing languages as complex as subsets of JavaScript.

Usage of a self-written lexer for token extraction and analysis in the evaluation.

The subjects we chose for evaluation did not always come with a formal specification like a grammar or a formal lexer, hence we wrote our own lexing algorithm and added lexemes from the code and documentation of the subjects. The lexer we used generically applies pre-defined regular expressions to the inputs to greedily lex and match the different lexemes (skipping whitespaces). If it cannot match a character it tries to skip the respective value and continues with the remaining input. Neither is the lexer perfect nor can we guarantee that we found all lexemes—not as input to the lexer in form of tokens defined by us and also not in the inputs we analyzed with the lexer. Still, we tried our best to

include all values, and used the same lexer and lexemes for evaluating all tools. Hence, we believe that even if there are slight errors in the lexer, the evaluation is still fair, as the evaluation of all tools should suffer the same problems. Also, we made the original publication data and metrics public (as far as possible) [103].

Early errors for parsing failures required.

Another threat is the restriction to early errors and early exits on parsing failures. If the program keeps on parsing after the first character was determined as invalid, we cannot distinguish properly between valid suffixes and the first invalid character or lexeme. Parsers typically stop parsing on the first error or are configurable as such and even for the remaining parsers this can often be solved by adding a program exit to the error reporting function.

Overlapping error codes might misguide the input generation.

Also, we are prone to misguiding error handling by the subject under test, i.e. if the program reports parsing errors the same way as other errors, our input generation pipeline will fail to distinguish parsing errors from other errors (like semantic mistakes). Hence, the generation of syntactically valid inputs will take significantly more time. While we cannot do much about this (if a program does not distinguish errors, we will run into this problem), we believe that this can be handled with a proper fuzzing wrapper or slight program changes (like we did in our evaluation).

Typically, many programs have a clear distinction between the parsing and the semantic phase, hence it should be possible to sufficiently distinguish between parsing and semantic errors. Instead of relying on the return value of a program, one could check the actual program output to distinguish parsing errors from other errors [57]. Also, instead of running the full program for input inference, one could just run the parser alone, stopping the program after parsing—then every error is a parsing error. It is also possible to only allow non-zero exit codes in the parsing step. Hence, we acknowledge this threat but believe that this roadblock can be solved with one of the techniques above if someone wants to apply our approach on their subjects.

Limitation to AFL as subsequent fuzzer in the evaluation.

We acknowledge that only using AFL, even though it is one of the industry standard fuzzers and the baseline for many more fuzzers [2, 99], means that we cannot generalize our results without restrictions. Still, we believe that our intention to evaluate the capabilities of our approaches to generate diverse seeds and tokens is fulfilled. Research on seed selection found that the right set of seeds is crucial for a proper fuzzing campaign and often influences the outcome [62, 79]. The evaluation with AFL shows the influence of our generated seeds and dictionaries on its fuzzing performance.

Nonetheless, it is obvious that any fuzzer that does not use seeds or dictionaries will not profit at all from the results produced by *our approach*. Hence, we can only claim a performance improvement for those techniques, but not for approaches that, for example, rely on a model of the subject under test, like a grammar. For such techniques we would

need to use additional steps, e.g. combining *our technique* with MIMID [54] or AutoGram [65] and see if the grammar inference performed by those approaches works well with the inputs generated by lFuzzer.[20]

Limitations of a prototype implementation.

For some cases we had to choose implementation complexity over completeness in our prototypes pFuzzer and lFuzzer. One prominent example is the wrapping of standard library functions. Be it in the tainting engine to have a summary for taints flowing in and out of a function or taint replacements for specific functions like *strtod*. In both cases its a one time generic effort and in a commercial tool or larger project one would implement wrappers for all different standard library functions. In our prototype we just implemented those functions that appeared in the subjects under test as a proof-of-concept, the replacements were randomly (but manually) selected and the taint summaries were generically defined. It may be that our choices were influenced while implementing those summaries, but we believe the chosen replacements are generically valid and diverse enough (except specific values are required) and the taint summaries are defined by the implementation of the library function and as such leave no room for subjective interpretation. Hence, we believe that this tradeoff between implementation effort and completeness did not influence the results and only needs to be accounted for when comparing the implementation complexity of pFuzzer and lFuzzer to others.

Deviations in the prototype.

Even though we tried our best to implement the tools pFuzzer and lFuzzer as well as the evaluation scripts as correct as possible, it would be utopian to believe that there are no implementation deviations from our approach. Our prototype already contains thousands of lines of code, mostly written by one person. Hence, there might be known bugs arising after evaluation and publication (see Section 4.2.1) and unknown bugs. For this dissertation we went through the code again to ensure that the details noted in this thesis are in line with the original publications—especially the code itself. If we encountered mistakes that we deemed to be impactful enough, we tried our best to note them down and quantify their influence in this thesis.

Though, since we believe that the bugs (known and unknown) in pFuzzer and lFuzzer only give a disadvantage to the respective tools and there will always remain hidden coding errors, we refrained from evaluating everything again. For example, the missing commutativity we found in Section 4.2.1 when generating token taints likely leads to less or incomplete generated token comparisons, hence the search tree when generating valid inputs might be incomplete. We checked the code in depth and believe that the evaluation is not impacted significantly, hence it is still showing what was planned to be proven: *our approach works well enough on real world subjects and is able to generate a diverse set of valid inputs which can be used for fuzzing.*

[20] We believe that the combination with grammar inference approaches is beneficial, as those need inputs that cover many different syntactic features, which are the inputs we generate—but this hypothesis is not proven yet.

6.8 Evaluation Summary

All results and evaluation scripts as well as a replication package of the original publication are available at: `https://dl.acm.org/do/10.1145/3406885/abs/`. For this thesis we used the same data as in the replication package, but improved the visualization of the graphs.[21] As we can see from the results, a certain amount of domain knowledge is crucial when fuzzing systems with grammar based formats, even if it is just a dictionary. AFL without any form of a dictionary cannot generate a large set of syntactically diverse inputs, it gets blocked by guessing the correct keywords which takes a lot of time, especially the longer the keywords get. The results for pFuzzer show that a program analysis via dynamic tainting is already helpful for subjects with a simple parser that do not have a tokenization phase. Using pFuzzer in combination with AFL can already boost this approach, but still lacks the information about keywords and the seed inputs pFuzzer can extract are also not sufficient for in depth fuzzing. Running AFL in combination with a dictionary of extracted strings already yields good results as long as syntactic complexity is not needed, but as we can see on the tinyC subject, generating syntactically complex inputs is still hard. Only lFuzzer can combine both dictionary and syntactically complex seed generation, which in turn makes it possible for subsequent approaches, like a greybox fuzzer as AFL, to generate a wide variety of inputs that do not only cover a large portion of code, but also a large amount of language features of the underlying format—**solving all our objectives as defined in Section 1.1**.

> In general it holds: the more restricted the input language, the greater are the benefits of automatic dictionary extraction and seed input generation as done by lFuzzer.

[21] We also corrected some mistakes in the evaluation scripts, e.g. bugs and missing tokens in the calculation of found tokens.

7 | Related Work

In Chapter 2 we explained the foundations our work builds upon and relies on. In this chapter we want to bring our work in context of existing approaches, explaining the *state-of-the-art* and how our technique compares—and even more important differentiates—from other ideas. We will first bring our approach in the context of input generation for input parsing programs work in general, then we go into detail on different input generation and fuzzing techniques that can be used in the context of parsers and are closely related to our work.

7.1 Input Generation And Input Formats

This section gives an overview on the different research directions of input generation in the context of testing programs with a parsing stage. We will often follow the generation techniques as described in the Fuzzingbook, as it already gives a broad overview on the typical input generation techniques [161]. We will not cover all tools or types that exist as the field is too large for the scope of this thesis; still we try to cover the most relevant techniques. To have a running example, imagine we have a simple compiler for some C-like language and we wait for the input generator to produce a `while` loop (`while () {;}`).

7.1.1 Code Analysis Depth

Let us have a look at the typical levels of code analysis during input generation: *blackbox*, *greybox*, and *whitebox*. Typically, all input generators fall in one of those categories, as they define how much information the technique uses. The lines between the different types are not sharp—some might consider a technique to be one type, others would categorize it differently.

Blackbox Blackbox generators mostly work very similar to the work of Miller et al. [109]: they randomly generate inputs, give them to the subject under test, and check for crashes—they might use some form of specification for deriving tests though [161]. Typically, no other feedback is involved; in fact a blackbox input generator does **not** use any information about the program code at all. Tools like Radamsa [4] are more involved as they use sample inputs and mutate them instead of running completely randomly from scratch. They also apply mutations not fully at random, but pick certain possible lexical elements (like numbers) and apply targeted mutations on them. Approaches like Glade [13] and the blackbox adaption of pFuzzer from Gopinath et al. [57] make use of the program feedback and possibly seeds to infer the underlying input structure with a large amount of samples.[1]

[1] Glade actually infers the underlying grammar to some extent while Gopinath et al. primarily try to efficiently generate syntactically diverse inputs, but do not learn the underlying grammar.

Basic blackbox input generators have the main disadvantage that they have virtually no feedback from the subject under test, hence in the context of complex input formats they randomly generate inputs and only by chance create a valid input, in all other cases they just see a rejection response (e.g. a non-zero exit code). Hence, in the case of a `while` loop the generator would first need to generate the `while` keyword. If our search space is the space of ASCII characters, the generator has, for each character of `while`, 128 possible options; only one is correct. Hence the chance to generate a correct `while` keyword is 128^{-5}—and this is just the keyword, the rest of the statement is still missing. Thus, naïve random blackbox input generation will not generate valid inputs efficiently. Adding seeds like in Radamsa [4] would result in more valid inputs, but we will likely not see new features (i.e. new lexemes). In best case the approach just shuffles tokens present in the seeds. A program model will obviously solve most of the issues (like in the work of Havrikov et al. [60]), but we will come to this later.

Greybox Greybox input generation goes one step beyond blackbox input generation and uses weak signals from the underlying code. Some of the best known greybox input generators are AFL [160] and libFuzzer [93], relying on coverage that is tracked for each input given to the subject under test. A larger portion of greybox input generators actually uses AFL as their baseline and build on this codebase [99]. In industry, tools like ClusterFuzz [138] (which is built and used by Google) make use of AFL [160] and libFuzzer [93]. The interesting thing about greybox input generators is that they are mostly language agnostic and independent from the subject under test. In contrast to blackbox input generators they need at least some sort of instrumentation on the code, but such instrumentation should be lightweight and in best case easily applicable to other domains (like other programming languages).

Greybox input generators have more information about the subject under test—especially they know which parts of the code were covered during execution. This information can also be used to infer valid inputs from program executions. In our evaluation based on our publications [102, 105] we have already shown that a basic greybox input generator like AFL [160] is in fact able to produce some valid inputs, but due to its generality those inputs are rather simple and are missing some input tokens (an input dictionary or a well selected seed set will help improve input diversity). More specialized greybox input generators like Grimoire make a more targeted use of their instrumentation feedback and try to infer a partial input structure to create more domain specific and higher level mutations [17]. Such techniques that are more focussed on a specific domain, in this case parsers, might be able to overcome the limitations of greybox input generation in the area of complex input validators in exchange for generalizability. Approaches like SmartFuzz [118] and Superion [150], which use some form of input model, can combine the information from the model with the instrumentation feedback, thus they do not need to infer the underlying model of the subject but can rely on the given model. Hence, they do not suffer from insufficient inference methods and instrumentation feedback.

Whitebox The last type of input generators (when speaking about *"code analysis depth"*) are whitebox generators. Here the input generator assumes full knowledge about the source code (or possibly some intermediate representation like LLVM bitcode) which is used for deciding which

inputs to generate—either statically like KLEE [23], statically mixed with dynamic information like Driller [135], or completely dynamically. Our tools pFuzzer [105] and lFuzzer [102] also fall in this category as they are using source code information dynamically during runtime to decide which inputs to generate next and which mutations to apply. Other tools like Weizz [41] or SLF [156] also use dynamic runtime information—in this case comparisons.

In whitebox input generation the input generator makes use of any information it can retrieve from the subject under test's code—making it the most promising approach for generating syntactically complex and valid inputs. Our approaches pFuzzer and lFuzzer [102, 105], as discussed in this thesis, are whitebox techniques and show that it is indeed possible to infer syntactically valid inputs. The approach by Sochor et al. [132] and MIMID [54] also show that not only inputs can be inferred with whitebox analysis, but also context-free grammars.

Taint-based approaches check the information flow during fuzzing to improve the inference of new inputs. Liang et al. [84] use dynamic tainting to extract interesting parts of the input that end up in *"unsafe functions"* [84] that *"are often used without strictness that may lead to memory leakage"* [84][2]. Ganesh et al. [47] use dynamic tainting in combination with pre-defined and user-defined attack points (specific library and system calls) to select input bytes that affect values at the defined attack points. Only the selected bytes are mutated—*"according to the types of these attack point values"* [47]. Aschermann et al. specifically avoid any tainting or symbolic execution with their tool Redqueen [9]. Among others, they use an *"input-to-state correspondence"* [9] because *"in many cases, parts of the input directly correspond to the memory or registers at run time"* [9]. With this, they can approximate data flows without taint-tracking while still being *"able to control these values by changing the corresponding input bytes"* [9].

7.1.2 Input Mutation Techniques

Apart from the depth of code analysis, it is also important how the input generator alters already generated inputs, i.e. which input mutations are used. In the following we discuss the different mutation techniques and explain how they would generate syntactically valid inputs.

Random Mutations First and foremost, an input generator generates inputs with a certain level of randomness, some portion of the input generation process is randomly decided. In the most basic case inputs are generated completely at random [109], sometimes by using sample inputs and very generic higher order mutations (like altering numbers to other numbers instead of randomly changing bytes in inputs) [4]. AFL goes one step further and not only incorporates a set of higher-order mutations, it also selects seeds to mutate based on their previous performance of discovering new code [160].

The reasoning why random mutations are typically insufficient to generate syntactically valid inputs is analogous to blackbox input generators: the chance to produce a `while` keyword from

[2] Liang et al. run their analysis on the binary level and categorize their approach as black box. Based on their usage of dynamic tainting, we rather categorize it into whitebox fuzzing.

the basic ASCII characters (128 different characters) is 128^{-5}. A promising way to increase the number of valid inputs during input generation is using seeds and trying to apply syntactically non-destructive semi-random mutations (like replacing one number with another—this likely keeps the input valid; done for example in RADAMSA [4]).

Static Information For Mutations Some tools use static information extraction and code transformations to improve input generation performance: they try to either extract valuable information for an input generator [131] or instrument the code to reduce the number of input generation roadblocks [82]. Valuable information could be a well extracted dictionary like Shastry et al. extract it statically from a subject under test [131]. Ebrahim et al. use CODEQL [48] to extract *"valuable information,[...] i.e., commonly occurring keywords, strings and constants"* from the subject that are then further filtered to reduce noise (e.g. introduced by error and warning messages) with user customizable filtering methods. The results can then be given to a fuzzer as a dictionary [38]. In DIFUZE the authors use a *"static analysis to compose correctly-structured input in the userspace to explore kernel drivers"* [32]. With this information they can *"automatically generate valid inputs and trigger the execution of the kernel drivers"* [32]. In this thesis (and our paper on *Learning Input Tokens for Effective Fuzzing* [102]) we have shown that even a simple extraction technique like string extraction is useful for dictionary creation and input generation. While static lexeme extraction might be useful to improve our techniques, we get those lexemes anyway while extracting seed inputs as our approach needs to approximate tokens and their usage during program execution. In Section 4.1 we detailed how we extract such information dynamically.

One of the most prevalent input generation roadblocks for greybox input generators are magic values in comparisons, as those need to be guessed, especially if they are atomically compared. For example, if a part of the code is guarded by a *strcmp()* call and the compared token is the keyword **while**, a greybox input generator needs to blindly guess this word using the 128 ASCII characters, having a chance of 128^{-5} of guessing correctly. Tools like the LAF LLVM PASSES reduce this effort by making such comparisons more greybox input generation friendly, i.e. comparisons against constants are split into sequential character comparisons. The example from the LAF LLVM PASSES webpage shows the idea [82]: the code

```
if(!strcmp(directive, "crash")) {
  programbug()
}
```

would be converted to

```
if(directive[0] == 'c') {
  if(directive[1] == 'r') {
    if(directive[2] == 'a') {
      if(directive[3] == 's') {
        if(directive[4] == 'h') {
          if(directive[5] == 0) {
            programbug()
//...
```

presenting the greybox input generator for each correctly guessed character a newly found branch, reducing the search plateau. Hence, the input generator does not need to guess the `while` keyword correctly in one pass (in worst case 128^5 guesses based on the 128 basic ASCII characters), but rather gets feedback for every correct character, resulting in $128 * 5$ guesses at most (if always the last guessed character is the correct one and the input generator uses the valid prefix for further input generation).

While this is a valid technique to improve input generation performance, a dynamic observation of such comparisons is more promising if the code does not use constants directly in a comparison, e.g. if the code iterates over an array of string constants to check if any of the stored values matches. In this case, a dynamic approach like ours would still be able to detect and extract the used string constants in the comparisons as they are present during runtime.

Search Based Mutations A search based approach uses a fitness function (either given by the user or a more generic one like the *distance in number of branches* to a location in the code) to reach a goal [106]. The approach optimizes towards this function, e.g. by using a hill climber algorithm: *start with a random input, evaluate it's neighbors based on the fitness function, choose the neighbor with the best fitness value and repeat with another neighborhood evaluation.* A lot of work was done in the past to improve search based approaches [106]. ANGORA [27] uses such a technique to solve input generation roadblocks for standard greybox input generation, applying a gradient descent algorithm on hard to solve comparisons in the code.[3]

A search based input generator works very targeted towards a goal, using a fitness function which guides the way towards specific code locations or other properties [106]. Hence, if a fitness function is available which efficiently guides the input generator through a parser, such a search based technique would work well. Still, it is not only important to generate one syntactically valid input, but many syntactically diverse ones that cover different paths through the target—often one input cannot cover all syntactical features of the input grammar.

Technically, this search based approach is similar to what we are doing in the approaches in this thesis—we search in the direct neighborhood of generated inputs by applying parser specific mutations and evaluate them on a recursive descent parser specific fitness function. Valid input prefixes that cover parser features never covered by any input beforehand get a favorable heuristic value; if no such prefix exists our approach tries to generate one. Once we found a valid input the set of already covered parser features changes (and thus the data used in the fitness function), hence we re-evaluate already known input candidates as we now have a smaller set of parser features that we still want to cover.

Model Based Mutations Input generators may not only use program feedback but also external input about the subject under test given in form of a human readable model. One

[3] It needs to be said that there are some doubts on the results presented in the ANGORA paper as discussed in a blogpost by Zeller et al. [7].

option is to provide input structure information, making it possible to either generate inputs from a grammar [60] or parse them [118] to apply mutations not only on single bytes or random substrings, but to keep the borders between the building blocks of the inputs intact (the terminal symbols for context-free grammar parsing subjects, i.e. the lexemes) and only perform targeted mutations on those building blocks. The main improvement over simple random mutations is, if the program input is structured, the basic structure of the inputs is known, depending on the quality of the model it is even possible to distinguish valid and invalid inputs. Hence, the input generator can decide if an input should be able to pass the input validation or if it should be slightly wrong, making the search space smaller and more targeted to different input features.

As already explained above, model based mutations can be used to generate syntactically valid inputs. The model defines the structure of the inputs, the input generator then needs to create inputs based on this structural description [60]. As such, when it comes to complex input formats, model based techniques are certainly in a high advantage, as long as a model is present and the model is complete and sound. In many cases though, the model might either not exist, not fully exist[4], is incomplete (specific features are missing), or contains more features than the subject. All of this is not uncommon, as the program model needs not only to be written but also kept in sync with the program itself and be in a machine readable form. Especially if a handwritten parser for a program specific new format is designed and implemented, it is likely that this format is constantly adapted throughout the development cycle and lifecycle of the subject under test—making it hard to keep the formal grammar updated. Thus, inferring inputs or a grammar from the subject under test is worthwhile as the input generator can then specifically target the actual code instead of the (possibly incorrect or outdated) model.

Concolic And Symbolic Mutations Concolic and symbolic testing can be directly related to whitebox input generation—the testing tool has full knowledge about the subject under test and uses this information in combination with an SMT solver to build inputs that cover the defined paths. One of the best known symbolic execution engines is KLEE [23]. Essentially, in symbolic execution a path is chosen through the program, then an SMT solver is used to solve the conditions collected along the path which results in an input that would exercise the respective path through the subject under test—an input that renders each condition to be *true* or *false*, depending on the direction the path should take. As such, symbolic execution is not dependent on any sample inputs and makes it possible to create inputs very targeted—making them traverse the selected paths (if the solver finds a solution). Concolic execution is the hybrid of concrete execution and symbolic execution, one example would be the tool Cute [129]. Here, the subject under test is run with a test input and the comparisons on the path are collected including their symbolic values. Now the tool selects the last comparison along the path and decides to negate it, knowing from the collected symbolic values which parts of the input need to be kept to not alter the path up until the negated comparison. CSEFuzz [153] uses symbolic execution together with test-case selection based on coverage criteria to generate a set of seed inputs, which can then be used in a subsequent fuzzing campaign.

[4] For example, the subject under test accepts JSON inputs but also requires specific key-value pairs—a JSON grammar is easy to find online, the specific values though may not be found easily.

Both symbolic as well as concolic execution are typically used very generically and especially symbolic execution suffers from the path explosion problem [24]. As such, even though theoretically those approaches would be able to cover all possibilities of a parser, they will likely end up traversing many error paths and generating inputs for those (especially when focussing on breadth-first search, as the parser error paths end early in the execution). With depth-first search they might end up generating syntactically similar but semantically diverse inputs when creating inputs for the program logic which comes after the parser.

Domain/Target Specific Input Generation In some cases input generators are not as generic as the above mentioned methods but very targeted to either a certain domain (e.g. programs that consume inputs that are backed by a grammar [49]) or to very specific targets (e.g. compilers for a certain language like C [154]). For domains and targets that are complex to test (e.g. because they have a very restricted input format or a very unique environment like embedded systems software) it is very beneficial to build an input generator that is targeted towards such restrictions and uses specific knowledge for more efficient input generation. Even though more knowledge given to the input generator will almost always improve input generation, for those restricted domains it is the only chance to have sufficiently efficient input generators that can properly test the subjects.

Obviously, domain and target specific input generators are the gold standard when it comes to input generation systems with complex inputs. The more specific an input generator is, the more targeted are the generated inputs (plus the input generator can be fast in generating new inputs because it is optimized for the given domain or target). Hence, tools like Csmith [154] can *actively* produce inputs that are either valid or invalid—also the input generator can produce values that include border-cases of the specification. For example, Holler et al. [63] used mined bug reports for JavaScript engines and included those code snippets in their newly produced inputs. As with model based testing, only features that are implemented can be tested and if the input generator implements more features than the subject under test, it might generate inputs that are rejected.

7.2 Symbolic And Concolic Execution

For input generation, typically most of the resources available for testing are put into program execution and observation: the input generator generates an input, gives it to the program and observes the execution and its output. The faster this is done, the more often a program can be executed, and the more diverse those execution paths are, the better are the input generation results in general. Symbolic execution and concolic execution have a different approach: here most of the resources are put into solving path constraints and generating inputs that follow those very paths. Both approaches are similar to our ideas, hence we want to go into more detail about them and bring them in direct context to our techniques.

One of the most popular tools for symbolic execution is KLEE [23], a testing tool that builds a symbolic representation of the subject under test, selects paths through the program, and

generates inputs (with the help of an SMT solver) that follow those paths. KLEE does not randomly select those paths, but, according to the original paper, *"KLEE has two goals: (1) hit every line of executable code in the program and (2) detect at each dangerous operation (e.g., dereference, assertion) if any input value exists that could cause an error"* [23]. Thus, KLEE is designed to specifically concentrate on code regions that could be buggy. In order to be efficient, KLEE also uses several optimizations like replacing implied value replacement ($x + 1 = 10$ gets replaced to $x = 9$) and constraint independence—those constraints can be put in different sets. The authors of KLEE itself as well the community added many features throughout the years—a list of publications is available at the KLEE main website[5] [142].

Even though there are many additions and publications in the field of symbolic execution, making it an interesting candidate for testing input validators, one of the main roadblocks is still the path explosion problem. Cadar et al. mention this problem in their overview article: *"Path explosion represents one of the biggest challenges facing symbolic execution"* [24]. Also, there is some work done to reduce the path explosion problem [66, 67]. Input validators like parsers are no exemption from this problem. Many context-free grammars contain recursive elements in the underlying grammar, leading to an infinite amount of possible paths, many of them resulting in errors and program exits if the input cannot be parsed completely—only valid inputs can go beyond the parser. Hence, if the symbolic execution engine does not differentiate between parser and program logic, many generated inputs will end up in some error path of the parser, a result that we have also seen in our experiments in the *Parser-Directed Fuzzing* paper [105]. Our technique, which uses domain knowledge about parsers, specifically recursive descent parsers, focusses on the fast generation of inputs that solve the parser constraints and go beyond the parsing code.

Concolic execution, similar to symbolic execution, takes a path through the program and solves the path constraints to generate an input that will follow this exact path. In contrast to symbolic execution though, this path is taken based on an actual execution of the subject under test, hence concolic execution tools like Driller [135] typically start with one or more concrete inputs, and then select one or more conditions on the path the inputs trigger that would, if switched, cover new branches. Then, concolic execution approaches try to generate an input that will follow the switched branch for each selected condition (similar to what we try to achieve by replacing incorrect substrings of the input with values they were compared to, trying to make them valid). Concolic execution is well researched and applied in different contexts (e.g. Cute [129] working on C code including the handling of pointers in the subject under test during test generation and jCute [127, 128] working on Java code and in the domain of concurrent programs). Still, depending on its input generation style, concolic execution might also suffer from the path explosion problem, if it selects and solves paths similar to symbolic execution (i.e. puts more weight into the symbolic execution part of the overall approach), resulting in many dead ends and only a few valid inputs. And even if depth-first search is used, symbolic execution and concolic execution might concentrate on very specific syntactical

[5] We will not go into detail here because this is out-of-scope for this thesis as this dissertation is not focussed on symbolic execution but on testing systems with complex input formats in general.

features and create semantically diverse inputs for those features. For example, they might find a path through the parser into the program logic and then would possibly generate inputs that change the end of the path, which likely lies in the program logic—thus altering the semantic, but not the syntactic features of an input.

Context Concolic execution is very similar to the approaches presented in our publications [102, 105] and this thesis: the subject is run under instrumentation, runtime information is collected, and then a solver decides which path to take next based on previously taken paths. The main difference is the domain knowledge included in our approach, especially the usage of comparisons on input bytes as separated events to solve—besides other domain specific optimizations that reduce the *path explosion* problem. Concolic execution solves a path, which guarantees the next input to take the path at least to the solved point, with our technique this is not guaranteed—it might happen that replacing input characters based on a comparison causes a different path to be taken. Thus, our approach spends more time with testing the subject (as it is used as an oracle) instead of calculating the next input. For example, let us assume we first saw the comparison "`input[2] > 'B'`" on our execution path and then "`input[2] < 'H'`". As our approach does not use context, if it solves the second comparison, it might select `'A'` as a substitution for `input[2]`, which would alter the result of the first comparison [6]—with concolic execution this would not happen as it solves the whole path.

In general, our approach is more efficient than concolic execution for the domain of recursive descent parsers, because we make use of domain knowledge to reduce the effort put into solving path constraints. Comparisons as mentioned above do not happen often in parsing, except for range checks, but those are mostly handled by our heuristics. A parser typically checks for different valid options at the respective positions. Hence, in general we have a sequence of equality comparisons—solving any of them would only alter the respective path at this comparison. For example, we might collect the comparison sequence "`input[2] == 'A'`", "`input[2] == 'B'`", and "`input[2] == 'C'`" with the input `"DEF"`. Obviously, all three comparisons failed, which is expected as the parser tried to match `'F'` with the different options at this parsing stage, which are the characters `'A'`, `'B'`, and `'C'`. Now replacing `'F'` with any of those options will lead to the behavior we want: *all options except the one we altered are still invalid.* That said, one might also be able to specialize a symbolic or concolic execution tool for input parsing programs to generate syntactically valid and diverse inputs (e.g. once a feature/token of the parser is present in an input, the concolic/symbolic engine should favor unseen features/tokens), but we are not aware of any tool that does this.

Summary While our approaches are close to concolic execution, we found a middle ground between precision and speed, resulting in a more efficient input generation technique for the domain of systems with complex input formats.

[6] In fact, we implemented heuristics in pFuzzer and lFuzzer for code constructs that check for character ranges like `input[2] > 'B' && input[2] < 'H'`. We assume that directly consecutive comparisons on the same input character using the operators *less* or *greater* belong to one single range comparison, giving the solver engine only the option to choose a character in the respective range (in this case between `'C'` and `'G'`).

7.3 Model-Based Input Generation

Besides whitebox input generation (especially symbolic and concolic approaches), model-based input generation is one other domain that is highly related to our techniques. While we are not working with a program model itself, it is very common to have or infer some sort of input model when generating inputs for systems with complex input formats. The reason for this is simple: generating inputs for such systems without any knowledge, i.e. mostly blindly, often ends in insufficient input generation results as we have seen in our evaluation. In order to overcome this issue, the input generation community came up with approaches that incorporate domain knowledge in different ways. In the following we go into detail how this can be achieved.

Context Model based input generation is, when it comes to input generation efficiency, code coverage, and bugs found, in general superior to our approach. If the model does not miss crucial features from the input domain of the subject, it can be considered as an upper bound for our techniques. That said, there are still drawbacks while using those techniques. First of all, those tools suffer from human bias, they will only test the features a developer decided to test (those domain features that are given to the tool).[7] Second, the input domain specification must be implemented, hence there is a huge effort to specify what the input generator should do. Typically, one major roadblock for implementing input generation as a part of the CI/CD pipeline is the effort to set up an input generator. Even though simple greybox techniques require no more than using a specific compiler that performs the input generation instrumentation in the subject under test, practitioners (users from the industry) still see usability as one of the most important areas to improve in todays input generation research [18]. Hence, we can assume that practitioners will tend to not maintain a domain specification in parallel to their code base, not even speaking of writing an initial one. Especially in the case of non-generic formats, this will make the application of subject specific input generation unlikely in many cases. With our technique, this specialized information is not needed: as long as the program contains any recursive descent parser (which is one of the most popular parsing techniques), we can run our domain specific technique and test the subject under test.

7.3.1 Generator-Based Approaches

As one might expect: giving input generators domain knowledge highly increases their efficiency, the better the knowledge and the more advanced the usage of this knowledge is, the more precise an input generator can create inputs. Hence, it is efficient in solving the specific task it was designed for. For instance, an input generator like Ijon [10] lets the user decide which features of the subjects are interesting to explore, i.e. the user can add different annotations to the code to guide the tool, giving it additional information about the specific target program. One could also think about giving domain knowledge to such a input generator. Such knowledge could then be statically or dynamically applied to concrete subjects without the need to manually annotate each subject individually.

[7] Some tools provide so called “out of specification” mutations, but those will only search in the near surroundings of the model, they will likely not detect new features, e.g. features that require completely new lexemes.

For program models, the input generator decides based on the given model how and which inputs to generate, with a generator approach the user writes code to define how inputs should be generated. There are different forms of input generators some like CSMITH [154] and JSFUNFUZZ [124] are focussed on a specific target (in this case the languages C and JAVASCRIPT). Generator approaches have the advantage that the respective generator code can be freely implemented, hence with such an approach it is possible to build any input. For a tool like CSMITH, which is very targeted to one specific language, this means that not only the syntactical and semantical features of the underlying input format can be added, but also interesting peculiarities and common error sources. For instance, Yang et al. designed CSMITH to output programs that are free from undefined behavior [155], a feature that requires a high context awareness, e.g. to avoid accessing uninitialized memory even in complex generated programs. Nonetheless, this technique, while possibly being highly effective on its respective targets, has the drawback of needing large manual effort and maintenance. A specialized tool like CSMITH has more than *25,000* lines of C, C++, and C/C++ Header code[8] [33]. Furthermore, even if the first version of the generator is finished, it will likely need debugging and ongoing maintenance for targets that are still under development.

Context In contrast to that, our techniques work directly on the code and analyze it with generic domain knowledge instead of manually written or user defined information, generating inputs for the input format that is encoded in the recursive descent parser of the subject under test using automated inference techniques. As such, our approaches are neither dependent on any manual work, nor are they prone to failures done during the implementation of the input specification or the generator. On the other hand, our techniques can only be applied to subjects with a recursive descent parser, while generator techniques have virtually no restriction to any input format, as long as the input generation can be encoded in program code, they can generate inputs. Hence, our approaches build a middle ground between simple greybox testing which struggles with generating inputs that are highly structured and generator based approaches which are very targeted to the subject under test or at least the input format.

Summary If a generator is available, this should be the first choice to test a program. If the target is still under heavy development and input format changes happen frequently, it might be beneficial to use our approaches first (or any approach with small to none manual effort for application) and only once the format is mostly set a generator should be implemented. Also, our approaches could be used in addition to generators, they might find inputs that are differing between the underlying format of the generator and the subject under test itself.

7.3.2 Input Model Based Approaches

A grammar can be seen as a model describing the input space. In contrast to a generator approach though, a grammar is typically written in a description format that is specifically designed to describe an input language. Grammars often already exist for a variety of input

[8] Counted with `github.com/AlDanial/cloc` version 1.90 using the command *cloc ./src* from the CSMITH root folder, using the code column of the output.

formats (the website Grammar Zoo [147] lists more than *1000* grammars at the time of writing—covering a wide variety of input formats) and can be used for different applications, e.g. to generate parsers instead of writing them by hand. Tools like ANTLR [136] and Bison [44] take a grammar in their respective description format as an input and generate parser code in the specified output language (e.g. C or Java code).

Besides generating input parsers, another application for grammars is input generation. With a grammar a test input generator can build inputs very targeted, similar to other model-based approaches, but developers often can reuse an existing grammar or adapt an existing one to their needs, making the tradeoff between input generation performance and manual work attractive. Especially, since a grammar can be used for more than just testing. Burkhardt [22] explored the generation of test programs from a model to test software systems very early—in 1967. For his work *"an example language was constructed in a subset of a FORTRAN-like language; and therefore the output format is consistent with the requirements of FORTRAN"* [22]. Based on the given example language, he automatically generated a set of programs.

There are many different approaches researched to improve the quality of grammar based input generation. Holler et al. used a JavaScript grammar to not only generate random programs to test JavaScript parsers, but they also crawled bug reports of JavaScript engines, extracted the respective bug inducing inputs, and embedded them in newly generated contexts [63]. This technique revealed numerous bugs, many of them caused by insufficient fixes of the respective engines and shown again by searching the direct neighborhood of the fixed code location. Havrikov et al. improved the grammar coverage, i.e. they presented an approach which systematically covers tuples of terminal symbols and nonterminal symbols of a grammar to guarantee that every (so called) *k-tuple* is covered [60]. Their reasoning is that every input feature (every nonterminal symbol) belongs to a code feature (a function or code location that handles the respective nonterminal symbol during parsing but also in the program logic), hence by covering k-tuples, they test combinations of code features. Pham et al. extend the greybox fuzzer AFL to incorporate information from a grammar for better fuzzing results [118]. Wang et al. present Skyfire [149], an approach to learn a probabilistic context-sensitive grammar from a set of inputs and a context-free grammar which can be used to parse those inputs. This grammar is then used to augment the set of initial inputs to cover more code—the initial and generated inputs are reduced to have less redundancy and used as seed inputs for further fuzzing. And even though the basic version of AFL does not supports full-fledged grammars, dictionaries are already a part of the tool. With them, AFL has larger building blocks with which the inputs can be generated and altered [158].

Since writing a grammar is not always necessary, but if so, a cumbersome and possibly also error prone task, approaches for learning and inferring an input format from samples have been proposed. Glade [13] starts with a very specific grammar containing the seed inputs given to the tool, from there on it tries to generalize the grammar first on the regular expression level (adding repetitions and alternations), then on the context-free grammar level (by adding

recursive productions).[9] All of this is done by generating targeted alternatives to probe the subject under test and use a given oracle to decide if an input was parsable. In LEARN&FUZZ [49], Goidefroid et al. use a neural network in combination with a large seed corpus of files having the same underlying grammar to infer the input model (technically, not a grammar but a neural network containing the grammar is extracted—we still count it under grammar extraction). Angluin proposes the learning algorithm L* to learn *"regular sets from queries and counterexamples"* [8]. Gopinath et al. use a whitebox technique for grammar inference [54], an advancement of the technique presented by Höschele et al. [65] that has a stronger focus on the *dynamic control flow* of the parser code instead of the *dynamic data flow*. Schröder et al. envision a technique to infer grammars from static program analysis [126], Schröder also went into more details about portions of this planned research track [125]—though, to the best of our knowledge, there are still many open questions they plan to solve in the future to make their technique complete.

Another type of input models are *schema files*. Especially the web domain often uses schema files like PROTOBUF [51] and JSON schema [112] to describe an API—the format of the data is already known but the containing data is described with such a schema file. Those schemas are machine readable, and in some cases like for PROTOBUF they can also be used to generate respective parsers like described in the Google tutorials for protocol buffers [50]. Hence, it makes sense to use them also for input generation, like the JSON SCHEMA FAKER [6][10] or the LIBPROTOBUF-MUTATOR which *"is a library to randomly mutate protobuffers"* [52] and *"could be used together with guided fuzzing engines, such as libFuzzer"* [52].

Context As we have seen, the field of model based input generation is large, many researchers worked on building better and faster input generators that are backed by an input model in one form or another. Going over all the different approaches covered in the recent years would be out of scope for this thesis, hence we gave a broad overview and will now explain how our techniques could help model based input generation—*concretely in the domain of grammar based input generators*.

All of the existing approaches either assume an input format specification or a sufficient set of sample inputs from which the respective format can be learned. While, as with generators, having an input specification is great for efficient input generation, it may not always be available, needs some amount of maintenance to be up-to-date with the actual implementation, and might miss features or have more features than the actual code contains. Similar problems hold for sample inputs: the techniques can only learn features that exist in the sample inputs, anything else will not be reflected in the extracted format specification. Hence, the sample inputs should cover as many grammar features as possible, possibly even in different variations

[9] In fact, Bendrissou et al. found that the *"effectiveness score (F1)"* [15] as claimed in the original paper [13] cannot be replicated with a reimplementation based on the description of the original paper [15].

[10] The JSON SCHEMA FAKER is actually a tool for just generating fake JSON inputs that are valid with respect to JSON schema files, but used as a backend for generating several different JSON files, it would function as an input generator as well.

and depending on the learning technique (e.g. if new inputs are generated to augment the initial set) the subject under test must also signal if an input is valid or not. Thus, we developed our techniques pFuzzer [105] and lFuzzer [102] as presented in this thesis—both do not need any input formats or sample inputs but infer inputs directly from the subject under test's code and observed runtime behavior.

Still, having an input specification is in most cases much more efficient than using pFuzzer or lFuzzer, the input generation can be very targeted instead of having to use several trial and error runs to observe the subject under test. Nonetheless, with the techniques presented here as well as our techniques we are able to build a tool pipeline that (at least for some subjects), may combine the best of all worlds. We could first start with pFuzzer or lFuzzer to extract a set of sample inputs (and with lFuzzer also a dictionary of tokens). This step does not need any manual effort or input samples, we will get inputs that, in best case, cover all features of the underlying input format and nothing more. A tool like Grimoire [17] can take the sample inputs as starting set, start fuzzing with them as those inputs already cover the parser and try to infer a partial input structure based on its instrumentation to create more domain specific and higher level mutations. Also, with a learning technique like MIMID [54], Glade [13], or Learn&Fuzz [49] we can extract the underlying input format using the seeds (and tokens) from pFuzzer or lFuzzer, hence we do not need to manually make sure that the seed set is 1. feature complete in general and 2. maintained to be valid and complete with the current program version. Now that we have an input model we can choose an input generator that accepts the generated model [60, 118] and use it for fast and efficient input generation—again without manually maintaining the format.[11]

Summary Input model based input generation is a great technique for generating inputs with complex formats, but it involves manual effort and might miss features that are forgotten to be incorporated into the input model or the sample seeds. With our techniques we make it possible to build a set of syntactically feature rich sample seeds that can be used for model learning and those models in turn can be used for input generation, enabling input model based input generation of subjects *out-of-thin-air*—without the need to involve any developer knowledge. While such a tool chain does not yet deliver the same quality of input models as a developer would do in all cases, our approach likely works best as a complement to model based input generators or during development of projects while the subject under test is still a moving target and the input model and seeds might often be out of sync with the code base.

7.3.3 Property-Based Approaches

Similar to generator- and input-model-based approaches, property-based testing relies on properties about the program that describe how a program should behave [40]. Those properties can then be used, in combination with other techniques and program inputs, to extract information about the subject under test—like code that is related to the properties or executions that

[11] All of this assumes a perfect world in which all tools are sound and complete, which is obviously not true. Still, even an incomplete pipeline would likely be able to infer a grammar which can be used in input generation.

are outside of the intended program behavior (with regard to the given properties). The benefit of such properties is that they avoid specifying the whole program behavior, but they only focus on the specific parts of the program that are about to be tested. This makes it possible to use the properties, besides others, as oracles for the program specifications they are reflecting.

Padhye et al. make use of property-based testing in JQF to combine it with coverage-guided fuzzing [115]. In JQF the developer writes property-based tests for the subject under test which are used, in combination with some input generator, to create tests targeted towards the given properties—also using them as an oracle to verify the correct handling of the input during program execution. Löscher et al. improve property-based testing by extending the properties to be targets for a search based testing strategy [96]. With the help of the search strategy they feed the input generator, trying to build inputs that falsify one or more of the given properties. Typically, this approach needs additional manual guidance: *"(1) the strategy that is used to explore the input space, (2) the component that supports writing targeted generators, and (3) UVs that we want to maximize or minimize"* [96] ("UVs" are the *"utility values [...] that specify how close input [sic] came to falsifying a property"* [96]). In a follow-up paper Löscher et al. reduce this issue by trying to automate the generation of a neighborhood function which serves as the targeted input generator. Thus, *"a user effectively now only needs to extract the utility values and specify whether to maximize or minimize them"* [97] (besides the setup that needs to be done for random property-based testing techniques).

Context Property-based testing can also be used to generate syntactically (and even semantically) valid inputs. For instance, Zest is such a tool for testing the *"semantic analysis stage of test programs"* [116] using a technique to convert untyped parameter sequences to syntactically valid inputs. Such parameter sequences can be bit sequences generated with the help of a coverage guided, *"feedback-directed parameter search"* [116] approach which is integrated in Zest. New inputs are generated with mutations on the bit sequences and the input conversion phase in Zest converts those to syntactically valid inputs. The goal is to create semantically diverse inputs with the help of those mutations that are already guaranteed to be syntactically valid inputs due to the bit sequence converter. In other words: instead of letting the input mutation phase work on producing syntactically valid inputs, the complete parsing stage is handled by the generator and the input mutation can concentrate on the semantic phase of the subject under test.

Padhye et al. performed an evaluation on input parsing subjects with Zest [116] and showed that they can successfully cover code in the semantic analysis classes of their test subjects—outperforming tools like AFL and indirectly also showing that they can produce syntactically valid inputs (typically only syntactically valid inputs end up in the semantic phase of a program). Thus, we can derive that property-based testing is also applicable for testing input parsing programs. Also, search based approaches like the one by Löscher et al. [96, 97] might be able to help guiding the input generation towards syntactically valid inputs (depending on the quality of the given search strategy and the given properties). Still, those approaches, in contrast to our techniques, require guidance and adaption for each subject. With manual work one can

define how a syntactically valid input looks like and describe properties over such an input, which in turn can then be used to determine if a subject under test consumes the input properly. Our techniques might serve, possibly with some adaptions, as another input generator for input parsing subjects, which can then be combined with property-based testing to verify the execution of the subjects on the generated inputs in more detail.

Summary While property-based testing is certainly a great technique to analyze a program as a whole, it still involves manual work which again might result in missed or outdated features of the subject under test. The properties for the subject under test need to be written and maintained, which makes it possible to target those properties in depth, but then again does not target the program in breadth. The combination of our techniques and property-based testing though might be fruitful. One can test the program in breadth with our techniques, but certain parts that are more important and need more in depth testing and oracles, can be tested with property-based methods—e.g. security relevant properties [40].

8 | Conclusion And Future Work

This chapter concludes the dissertation. Thus, here we summarize the solved challenges, key takeaways, aggregate how our research improved over the *state-of-the-art*, and have a look into the future of *Fuzzing Systems with Complex Input Formats*.

8.1 Solved Challenges

In this section we want to give a very condensed overview on the achievements that we reached with our work, we summarize where we are in terms of *Fuzzing Systems with Complex Input Formats* with and without tokenizer code. Hence, nothing new will come here, this is only a condensed version of the main contributions without too much details.

Objective 1: Complex Comparisons With Magic Values First and foremost, we need to mention our fundamental idea and approach for *Parser-Directed Fuzzing*: using input character comparisons done during parsing with a recursive descent parser for input inference as presented in Chapter 3. Our idea is based on the observation that any recursive descent parser at some point needs to compare the input characters it currently parses with the valid options it has at that position (be it directly or after lexing). With this information we can create valid inputs efficiently step-by-step, replacing randomly guessed characters that are appended on already generated valid prefixes with the constant values they were compared to during the execution of the subject under test.

Additionally, beyond the usage of comparisons, we also leverage other domain knowledge regarding recursive descent parsers. For instance, we make use of stack depth to avoid nesting of features (represented as called parsing functions on the stack) and thus unnecessary complexity in inputs during generation, a metric which directly correlates with the typical implementation of a recursive descent parser. We also take into account the new coverage achieved by each input (compared to the code covered by valid inputs beforehand), but only the coverage in the parser, i.e. we detect (and under-approximate) which code still belongs to the parser and then only use this coverage for our search heuristics. This can only be done because we know that the subject contains a parsing step and we know the typical implementation style of a recursive descent parser.

Objective 2: Tokenzation While the first research contribution makes it possible to infer inputs from parsers in general, it requires knowledge about **parser** comparisons. Those comparisons not only give information about the valid tokens of the underlying language, but also about the validity of a token at the respective position in the input (i.e. the time in

parsing). In the presence of a lexer, this is not an easy task anymore: in this case the lexer contains the character comparisons, the parser only the token comparisons. We extended our dynamic tainting technique in Chapter 4 to also handle input-character to token conversions and propagate taints from the input characters to the lexer tokens (which finally end up in the parser and reveal the parser comparisons we are looking for).

Objective 3: Syntactic Correctness And Semantic Diversity Finally, during the design of our approach we found out that building syntactically valid inputs is a necessary requirement to generate inputs that reach code beyond the parsing step, but not sufficient for reaching all lines of code. Thus, we combined our technique with a fuzzing technique in Chapter 5: *greybox fuzzing*. The idea is that our technique produces a syntactically diverse set of seed inputs and a dictionary to give consecutive tools (like fuzzers) starting information. In our evaluation we show how this information can be used in combination with AFL, a *state-of-the-art* greybox fuzzer, effectively. While such fuzzers struggle with generating syntactically valid inputs from scratch, they are very good in altering existing inputs in a fast pace, leading to a high variety of existing values close to their originals. Thus, altering the inputs generated by lFuzzer in combination with the extracted lexical elements, yields a powerful combination of tools that is able to produce inputs that cover code of subjects under test with complex input formats efficiently and effectively.

8.2 Key Takeaways

We collect the key takeaways from Chapters 3 to 6 to summarize the whole approach before talking about possible future work and the conclusion of this thesis.

Dynamic tainting is a promising way to collect information about a parser for input inference.

There are *"limits on the efficiency of automated systematic testing beyond which random testing is certainly 'superior"'* [19]. Those limits are dependent on the *"number of error-based partitions and the fractional size of the 'smallest' error-revealing partition"* [19]. Still, our results indicate that input parsing programs can benefit from our systematic testing techniques which may spend more time in the input creation phase—it seems that overall our approach is still in the mentioned limits to be more efficient than random testing. We are able to generate inputs that are targeted towards the parser in the subject under test and cover a large variety of input features.

Comparison traces can be used to generate syntactically valid inputs *out-of-thin-air*.

While most parsers rely on general metrics like coverage, which is great for fuzzing general programs, we can make use of very specific information tailored to recursive descent parsers. This makes it possible to create a technique that is very efficient for this specific set of programs, which cannot be fuzzed efficiently with general purpose tools. We believe that in future the question is not: *"How well does my fuzzer perform on the most general set of programs?"*, but: *"how well does my fuzzer perform in this domain?"*, i.e.

is the approach both specialized and broad enough to perform well on subjects from a specific domain. Böhme et al. also recognize in their overview paper that fuzzing needs to be broadened to new types of software (like parsers or protocol based systems), and that benchmark sets need to acknowledge the existence of specialized fuzzers [18], hence we believe that domain specific fuzzing is in the minds of the broader fuzzing community. We presented an approach for the field of parsing, a field that could only be fuzzed efficiently with many sample inputs and knowledge beforehand (either through learning an input format with a large set of samples [13, 65] or by using a manually defined grammar [60]).

Dynamic taints can be extended to token generation.
Producing taints from unfiltered control flow is not feasible if one wants to have a meaningful set of taints—most runtime values would be tainted by the majority of input byte taints, because most of the code and thus most of the internally produced data is control dependent on the input. Hence, we present a technique that filters for token generation patterns and produces taints which are attached to the respective token values. With this technique we have less noise compared to tainting all bytes based on control flow. This noise can be further filtered with different heuristics to reduce the overhead and make the taints precise enough for observing parser comparisons on the token level.

Lexer code can be detected, separated from the parser, and parser code can be analyzed.
With the token taints produced by our tokenizer adapted tainting technique and the detection and separation of parser and lexer code, our approach is able to produce a stream of token comparisons. Including our token mapping, both presented in Chapter 4, this comparison stream can be analyzed with an adaption of the parser analysis backend from Chapter 3 to generate syntactically valid inputs *out-of-thin-air*.

Seed and dictionary extraction is beneficial for efficient fuzzing.
While AFL as well as lFuzzer already perform well as standalone tools, they have their weaknesses: AFL struggles with generating syntactically correct inputs but produces many different inputs which in turn often cover the semantic features of the subject while lFuzzer performs vice-versa. Thus, it is obvious to combine both approaches—lFuzzer extracts syntactically feature rich seeds and a dictionary, AFL then covers the semantic features of those seeds by recombining the extracted information. With this method we leverage the individual strenghts of each technique while mitigating the weaknesses. Furthermore, this approach is highly modular: *we can use any other fuzzer, as long as it accepts seeds and in the best case a dictionary.*

The semantic part of a subject under test (and especially the program logic) can be tested *out-of-thin-air* on subjects with complex input formats.
Not only does the parser analysis and AFL perform better in combination in terms of input generation, they also achieve something that was not possible before: covering a diverse set of semantic features. lFuzzer covers the parser broadly and AFL specific (syntactically simple) features deeply. Their combination enables the tool chain to first

broadly cover the parser and then let the subsequent fuzzing campaign go into depth and beyond the parser code—and all of this from scratch without the need to handcraft a seed and dictionary set. This essentially concludes our journey for now: we started with a technique that can analyze parsers that work directly on the input characters [105], went over an approach that can analyze more complex input validators that have a lexing and parsing phase, and ended up with a full fledged tool chain that can also test the semantics of a subject under test [102]. While all of this is now possible, we just scratched the surface with our prototype technique, there is more to explore in future.

This is just the beginning ...
Analyzing and testing systems with complex input formats is hard and even though our technique made it easier, the field is still open and waits for exploration. We believe that the future of fuzzing lies in specialized fuzzers that make use of domain knowledge (either manually provided for specific targets or generic domain knowledge). With this approach we set the foundation for *Parser-Directed Fuzzing* without the need for manually crafted input models—the future of parser fuzzing has just begun.

8.3 Future Work

In the previous sections we have shown the different achievements produced by our research in detail. This section though, has a look into the future of *Fuzzing Systems with Complex Input Formats*. We want to share our experiences in this area and give an outlook on what might be possible, what might be worthwhile to explore, and what we imagine should the the ultimate goal of *Fuzzing Systems with Complex Input Formats* and domain centered testing in general. Our journey is at a crossroad, there are different paths to explore, different areas to have a look into and we try to list them here. But not only that, we also made decisions in the past which may or may not have been the best, so we also take a look back and see if there are paths that might be interesting to explore deeper.

While our techniques for parser analysis already laid a great foundation and are capable to cover a broad set of grammars and their respective features, there are still many open tasks for future research. We see our approach as a prototype and one of the first of its kind to introduce efficient parser analysis and fuzzing *out-of-thin-air*. As such, we believe that all parts of this technique can be improved. In the following we go into detail about possible research directions.

Analyzing The Semantics One key problem of our technique, is the insufficient testing of the semantical level of the subject under test. While greybox fuzzers like AFL indeed are able to re-arrange and replace input characters to achieve a wider diversity of input features, also and especially on the semantical level, this still just scratches the surface. Tools like Zest [116] try to cover this semantical level, but typically they either have blind spots (like our technique) or need manual help (like Zest). Thus, we believe that our techniques need to be refined in order to test beyond the parsing phase in the subject under test. In the following we also give some ideas for extensions of our technique which could also help in that matter.

Context-Sensitive Grammars A sub-problem of the general testing of the semantic level are context-sensitive grammars. If one thinks about a programming language, the problem gets apparent very quickly with one of the most basic features: *use-def dependencies*. Variables and functions need to be defined before using them (be it syntactically earlier in the code or just semantically somewhere in the scope when calling them). Such concrete semantic dependencies might be solvable to some extent: one could think of using some specific algorithm that infers how variables or functions are defined and then use this technique as a kind of pattern matcher if a variable use is semantically checked. The program internal def-use checker would try to find the variable name in its scope, i.e. it might iterate over the defined variables and compare the known names with the one it has until it is either found or there are no more values in the scope. Such a pattern in the execution could be matched and then used to create an additional variable or function definition to satisfy the missing definition for a variable or function usage. Still, we lack proper techniques that are actually able to generally solve such semantic dependencies like the one in the example above for context-sensitive grammars.

Grammar Learning Another problem our approach has in its current form is the lack of structural knowledge. In fact, for each input that is tried to be inferred we start more or less from scratch, using collected information about already covered inputs and code and feed this information to heuristics, but no knowledge about the underlying structure of the grammar is extracted. Approaches like AutoGram [65], MIMID [54], and Glade [13] already try to use a set of sample inputs to infer such structural information. To the best of our knowledge though, no one combined input inference with grammar learning in depth—a combination of both would likely be beneficial though. Hence, while grammar learning techniques already exist, there is room to improve them further. With the combination of both techniques it would be possible to *omit writing test inputs or grammars manually*—reducing manual effort and human bias.

Efficiency A second problem arising from missing structural knowledge is the inefficiency of our approach. We learn and infer inputs as we cover the code, needing many iterations while dynamically tainting the program execution. This costs time which could be avoided by not only inferring inputs on the go but also learning structural information about the parser, which could then be leveraged to make even better decisions during input inference. One example could be, instead of inferring the contents between two parenthesis in an arithmetic expression by probing the subject, one could just leverage information from previous runs. For example, if we need to find a substitution for X in "`2 + ( X`" we would currently probe the subject until we found a correct substitution and then append a closing parenthesis. But, if we already produced another input that is a valid expression it might be beneficial to test if X can just be replaced with a known expression. For example, if we already generated "`1 + 2`", we can produce "`2 + ( 1 + 2`" and see if the input is accepted up until the number 2—reducing the number of needed runs. There might be many improvements of our prototype, which will make the input inference faster and give more room for semantic level fuzzing or other techniques.

Other Inference Techniques Speaking of efficiency, it is definitely possible (maybe even likely) that our approach is just a basic approach to infer inputs for parsers. Indeed, we decided

for a sufficient approach with low complexity for input inference: using comparisons to build valid inputs from and for the parser of the subject under test. This leverages one fundamental property any recursive descent parser has: it needs to compare input characters *in order* to decide for each character if it is valid at the respective position. Still, one can imagine other ways to make use of this property instead of analyzing a stream of comparisons extracted with dynamic tainting. Many parsers for instance already use some kind of error feedback, indicating if and where an input is invalid. Completely without instrumentation, just by using the error feedback, it is possible to find the *"maximal valid prefix of the input that if combined with some valid suffix will be accepted by the program"* [57] (even if this is done randomly, the speedup by not using dynamic tainting might already be sufficient to make this technique faster). One could even infer a grammar from a set of inputs only, without any other information (except the general class of grammars like context-free grammars) [30].

Analyzing A C++ Compiler Technically, this paragraph is not about analyzing a C++ compiler, this language is just a placeholder for all the target languages that are hard to analyze and fuzz fully automatically, if not impossible (for now). We chose C++, because we think it has one of the most complex features and is a very feature rich language, including many syntactical as well as semantical options to build inputs while on the other hand being well known and used a lot in real world. Now, we imagine a future in which it is possible to create inputs for C++ (and all the other complex languages) *out-of-thin-air*, without human knowledge involved, just by analyzing the fuzzer—be it statically, dynamically, or both. While we are still very far away from reaching this goal, we think it makes sense to give some perspective for the future, showing what we imagine our technique to be a foundation for. The ideas mentioned above are some of the larger milestones towards this goal—being able to analyze and fuzz programs with underlying languages as complex as C++ is certainly a tremendous milestone for *Fuzzing Systems with Complex Input Formats*.

8.4 Prototype Implementation

A non-negligible part of this work is about breaking other software, i.e. finding bugs in systems making them more secure, more reliable, and certainly closer to what they are supposed to be—the typical developer does not add a bug into her or his program on purpose. And testing approaches like fuzzing find thousands of bugs in systems that were deemed to be secure and even then those systems might still have vulnerabilities and produce crashes—even in software that is well written by many people and secured over years.

This brings us to three (maybe obvious) points that we still want to mention here: *no piece of software is completely bug-free* (if we do not know of any bug we likely just did not find it yet), it does not matter how hard one tries to document a piece of software correctly *there will always be a deviation between the documentation and the actual code*, and finally: *no matter how great the idea behind an approach is, there might always be a better one* (if this was the best approach to analyzing and fuzzing parsers, we would be done now, but this is utopical). This dissertation and the papers supporting this work were written with the highest standards to make sure that

everything is well documented, the code works as explained, we did not miss to mention any steps we took, and to come up with ideas that work as good as possible. Still, as with any other piece of software, as with any other work out there, especially for such long-running and large projects, it might be that we missed to mention something in this document.

While we cannot print all the code, the evaluation scripts, and the results in this thesis, at least we provided a replication package [103] along with the paper about *Learning Input Tokens for Effective Fuzzing* [102] which contains most of the information used in this thesis on a technical level. Of course, this does not guarantee that anything along the path between running the experiments, uploading the evaluation code and results, and improving the presentation style of those results for this thesis[1] did not go missing or went wrong, but it gives a second reference for the work done, i.e. the technical outcome of all the research throughout the years. We tried to make sure that the description as given in this thesis is a correct representation of what was implemented (possibly also correcting mistakes from the published papers), still, due to the size of the project there might be slight mistakes.

8.5 Final Words

Writing a dissertation is a long journey, a journey that begins with interesting and important open research questions, adds several new insights to the research corpus and ends with a summary of what was done—*this thesis*. During this journey, one does not simply focus on the path straight ahead—research is always about looking right and left, smoothing out roadblocks and improving over the *state-of-the-art* wherever possible. And most importantly: **great research does not only answer questions, it asks new ones that are important to solve in the future**. This section gives an informal final overview on this journey: speaking about the path taken, what we can take away when looking back at the years of research done, and what the future might look like.

We started with an overview on the background of our work, the foundations of our approach, in Chapter 2. This is the reason why we decided that there is a need of better analysis and fuzzing tools for systems with complex input formats, i.e. systems that parse inputs. From there we digged into the commonalities of different parsers and came up with our approach of using the intersection between the theoretical background of parsers (the grammars) and the actual code—*comparisons as the instances of terminals*, the building blocks of each input. But this is not enough: parsing code is more complex and well engineered code uses techniques that avoid direct parsing to abstract away details—*the lexing code*. Therefore, in a follow up we also analyzed the lexer and enhanced our initial idea to make use of the lexer and the parser together. Finally, we also encountered the problem of not *testing the semantic level* of the underlying

[1] The new presentation style and possible fixes are obviously not reflected in the uploaded code, hence the replication package does not fully represent the data and scripts used in this thesis. Mostly, we just altered the look of the graphs and the results are (mainly) based on the data that was uploaded. Also, we fixed some bugs in the evaluation scripts that generate the data, including bugs in the lexer we used to calculate the metrics of found tokens.

languages, which we approached by building a symbioses between our very targeted syntactical analyzer, and the broader but faster fuzzer AFL, covering portions of the semantic layer as well.

What did we learn on this long journey from our first idea of *Fuzzing Systems with Complex Input Formats* towards this document, summarizing the outcomes of this idea? First and foremost: it was possible to improve over the *state-of-the-art* and using domain specific testing tools is certainly a research path we would like to see more in the future. While general fuzzers are great for starting fuzzing, at some point they are not targeted enough to the specific problem and are too inefficient for proper testing of the subjects. Domain specific fuzzers though are able to keep most of the convenience of general fuzzers (especially the very low initial hurdle to start the fuzzer on the subject under test) while being able to test the subject faster and in more depth.

While it is great to see that our approaches work well, the journey is not over. We said in the beginning of this section that there must be new questions to answer, and we already show ideas for future research in Section 8.3, but to summarize: *the current state of fuzzing is far from perfect*—the domain is large, the subjects under test are a moving target (there are always new concepts that need a new way of testing), and the code depth the fuzzer reaches as well as the oracles it can automatically quantify are still weak. We contributed to the area of fuzzing systems with complex input formats, and now we look forward to a future in which fuzzing in general as well as in this specific domain will be improved.

And who knows, at some point we might be able to write code, give it to a machine and in minutes, seconds, or even immediately we get feedback about bugs in the code. This machinery might be fuzzing, but it might also be anything else, we will see what the current and future generations of researchers produce. But this day is not today. While the path towards more reliable and secure software is still foggy, we follow it step-by-step, building better and more efficient techniques that improve the world of software testing.

ONE THING IS CERTAIN:

AS LONG AS SOFTWARE HAS BUGS, TESTING RESEARCH WILL NOT STOP.

Abbreviations And Glossary

AFL—*A*merican *F*uzzy *L*op

"American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary." [160]

ASCII

"This coded character set is to be used for the general interchange of information among information processing systems, communication systems, and associated equipment." [25]

BNF—*B*ackus-*N*aur-*F*orm

A notation used to specify the syntax of a context-free language. [3]

CI/CD—*C*ontinous *I*ntegration/*C*ontinous *D*elivery

Modern software is often build, tested, and delivered mostly automatically, which is described by this term.

Fuzzer

In the broadest sense a test generator mostly driven by random decisions.

GUI—*G*raphical *U*ser *I*nterface

The visual interface between the user and the machine.

I/O—*I*nput/*O*utput

A typical abbreviation for the input and output of a system.

LLVM

"The LLVM Project is a collection of modular and reusable compiler and tool chain technologies." [95]

LLVM Bitcode

"What is commonly known as the LLVM bitcode file format (also, sometimes anachronistically known as bytecode) is actually two things: a bitstream container format and an encoding of LLVM IR into the container format." [94]

LLVM IR

"LLVM IR is encoded into a bitstream by defining blocks and records. It uses blocks for things like constant pools, functions, symbol tables, etc. It uses records for things like instructions, global variable descriptors, type descriptions, etc." [94]

LLVM Optimization Pass

"The opt command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results." [91]

Oracle

"A test oracle is a (set of) assertion(s) that should pass when the behavior of the module under test is correct, and fail otherwise." [83]

Bibliography

[1] Abhishek Arya (@infernosec). *Tweet: "100k+ CPU cores, mostly n1-standard-1 vms"*. https://twitter.com/infernosec/status/1429091401069862915, 4:41 PM, 2021-08-21. Accessed: 2021-09-06. Aug. 2021.

[2] afl++ Contributors. *The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn_mode, and a lot more!* https://github.com/AFLplusplus/AFLplusplus. Accessed: 2022-03-26. 2022.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[4] Aki Helin and the Radamsa Developers. *Radamsa*. https://gitlab.com/akihe/radamsa. Accessed: 2021-09-09. 2021.

[5] AlDaniel. *cloc counts blank lines, comment lines, and physical lines of source code in many programming languages*. https://github.com/AlDanial/cloc/. Accessed: 2023-11-05; loaded through homebrew: https://formulae.brew.sh/formula/cloc. 2023.

[6] Alvaro Cabrera and Contributors. *JSON Schema Faker*. https://github.com/json-schema-faker/json-schema-faker. Accessed: 2021-09-14. 2021.

[7] Andreas Zeller and Sascha Just; with Kai Greshake. *When Results Are All That Matters: The Case of the Angora Fuzzer*. https://andreas-zeller.info/2019/10/10/when-results-are-all-that-matters-case.html. Accessed: 2024-02-11. 2019.

[8] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and Computation* 75.2 (1987), pp. 87–106. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(87)90052-6. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526.

[9] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/.

[10] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[11] Barton P. Miller. *Computer Sciences Departement University of Wisconsin-Madison - Project List*. http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf. Accessed: 2021-09-07. 1988.

[12] Barton P. Miller, Louis Fredriksen, and Bryan So. *An Empirical Study of the Reliability of UNIX Utilities – ACM Website*. https://dl.acm.org/doi/abs/10.1145/96267.96279. Accessed: 2024-01-12. 2024.

[13] 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: Association for Computing Machinery, 2017, pp. 95–110. ISBN: 9781450349888. DOI: 10.1145/3062341.3062349. URL: https://doi.org/10.1145/3062341.3062349.

[14] Ben Hoyt and Contributors. *inih - Simple .INI file parser in C, good for embedded systems*. https://github.com/benhoyt/inih. Accessed: 2018-10-25. 2018.

[15] 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*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 260–268. ISBN: 9781450392655. DOI: 10.1145/3519939.3523716. URL: https://doi.org/10.1145/3519939.3523716.

[16] Trail of Bits et al. *PolyTracker – An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing*. https://github.com/trailofbits/polytracker. Accessed: 2024-02-10. 2019.

[17] Tim Blazytko et al. "GRIMOIRE: Synthesizing Structure while Fuzzing". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko.

[18] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. "Fuzzing: Challenges and Reflections". In: *IEEE Software* 38.3 (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[19] Marcel Böhme and Soumya Paul. "On the efficiency of automated testing". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 632–642. ISBN: 9781450330565. DOI: 10.1145/2635868.2635923. URL: https://doi.org/10.1145/2635868.2635923.

[20] Marcel Böhme, László Szekeres, and Jonathan Metzman. "On the reliability of coverage-based fuzzer benchmarking". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1621–1633. ISBN: 9781450392211. DOI: 10.1145/3510003.3510230. URL: https://doi.org/10.1145/3510003.3510230.

[21] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. "DroidMate-2: a platform for Android test generation". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, pp. 916–919. ISBN: 9781450359375. DOI: 10.1145/3238147.3240479. URL: https://doi.org/10.1145/3238147.3240479.

[22] W. H. Burkhardt. "Generating test programs from syntax". In: *Computing* 2 (Mar. 1967), pp. 53–73. DOI: 10.1007/BF02235512. URL: https://doi.org/10.1007/BF02235512.

[23] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *USENIX conference on Operating systems design and implementation*. Vol. 8. 2008, pp. 209–224.

[24] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[25] V.G. Cerf. *ASCII format for network interchange*. RFC 20. Oct. 1969. DOI: 10.17487/RFC0020. URL: https://www.rfc-editor.org/rfc/rfc20.txt.

[26] Cesanta Software. *Embedded JavaScript engine for C/C++ https://mongoose-os.com*. https://github.com/cesanta/mjs. Accessed: 2018-06-21. 2018.

[27] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[28] Yuanliang Chen et al. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang.

[29] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (Sept. 1956), pp. 113–124. ISSN: 2168-2712. DOI: 10.1109/TIT.1956.1056813. URL: https://www.its.caltech.edu/~matilde/Chomsky3Models.pdf.

[30] Alexander Clark. "Learning deterministic context free grammars: The Omphalos competition". In: *Machine Learning vol. 66*. Springer, 2007, pp. 93–110. DOI: 10.1007/s10994-006-9592-9.

[31] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: Association for Computing Machinery, 2007, pp. 196–206. ISBN: 9781595937346. DOI: 10.1145/1273463.1273490. URL: https://doi.org/10.1145/1273463.1273490.

[32] Jake Corina et al. "DIFUZE: Interface Aware Fuzzing for Kernel Drivers". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2123–2138. ISBN: 9781450349468. DOI: 10.1145/3133956.3134069. URL: https://doi.org/10.1145/3133956.3134069.

[33] Csmith Contributors. *Csmith, a random generator of C programs*. https://github.com/csmith-project/csmith. Accessed: 2022-04-16, Branch: Master, Commit: deddca60d146c692e0ec5e4e345c466bbb3594b1. 2022.

[34] Dave Gamble and Contributors. *cJSON - Ultralightweight JSON parser in ANSI C*. https://github.com/DaveGamble/cJSON. Accessed: 2018-10-25. 2018.

[35] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[36] Jay Earley. "An Efficient Context-Free Parsing Algorithm". In: *Commun. ACM* 13.2 (Feb. 1970), pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035. URL: https://doi.org/10.1145/362007.362035.

[37] Eaton, Phil. *Parser generators vs. handwritten parsers: surveying major language implementations in 2021*. https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html. Accessed: 2021-09-22. 2021.

[38] Arash Ale Ebrahim et al. "FuzzingDriver: the Missing Dictionary to Increase Code Coverage in Fuzzers". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 268–272. DOI: 10.1109/SANER53432.2022.00042.

[39] ECMAScript Community. *ECMA-262, 12th edition, June 2021 ECMAScript® 2021 Language Specification*. https://262.ecma-international.org/12.0/. Accessed: 2022-03-26. 2022.

[40] George Fink et al. "Towards a property-based testing environment with applications to security-critical software". In: *Proceedings of the 4th Irvine Software Symposium*. Vol. 39. 1994, p. 48.

[41] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. "WEIZZ: automatic grey-box fuzzing for structured binary formats". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450380089. DOI: 10.1145/3395363.3397372. URL: https://doi.org/10.1145/3395363.3397372.

[42] Andrea Fioraldi et al. "Dissecting American Fuzzy Lop: A FuzzBench Evaluation". In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Mar. 2023). ISSN: 1049-331X. DOI: 10.1145/3580596. URL: https://doi.org/10.1145/3580596.

[43] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machinery, 2004, pp. 111–122. ISBN: 158113729X. DOI: 10.1145/964001.964011. URL: https://doi.org/10.1145/964001.964011.

[44] Free Software Foundation, Inc. *GNU Bison*. https://www.gnu.org/software/bison/. Accessed: 2021-09-14. 2021.

[45] Free Software Foundation, Inc. *Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-fno-inline. Accessed 28-August-2023. 2023.

[46] R. Frost and J. Launchbury. "Constructing Natural Language Interpreters in a Lazy Functional Language". In: *The Computer Journal* 32.2 (Jan. 1989), pp. 108–121. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.108. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/108/1445656/320108.pdf. URL: https://doi.org/10.1093/comjnl/32.2.108.

[47] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[48] GitHub, Inc. *CodeQL*. https://codeql.github.com. Accessed: 2024-02-15. 2021.

[49] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: Machine Learning for Input Fuzzing". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 50–59. ISBN: 9781538626849.

[50] Google LLC. *Protocol Buffers - Tutorials*. https://developers.google.com/protocol-buffers/docs/tutorials. Accessed: 2021-09-14. 2021.

[51] Google LLC. *Protocol Buffers Documentation*. https://protobuf.dev. Accessed 15-July-2023. 2023.

[52] Google LLC and Contributors. *libprotobuf-mutator*. https://github.com/google/libprotobuf-mutator. Accessed: 2021-09-14. 2021.

[53] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "If You Can't Kill a Supermutant, You Have a Problem". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 18–24. DOI: 10.1109/ICSTW.2018.00023.

[54] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "Mining Input Grammars from Dynamic Control Flow". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 172–183. ISBN: 9781450370431. DOI: 10.1145/3368089.3409679. URL: https://doi.org/10.1145/3368089.3409679.

[55] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "Mining Input Grammars". In: *Software Engineering 2021* (2021).

[56] Rahul Gopinath et al. "Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing". In: *CoRR* abs/1810.08289 (2018). arXiv: 1810.08289. URL: http://arxiv.org/abs/1810.08289.

[57] Rahul Gopinath et al. *Fuzzing with Fast Failure Feedback*. 2020. DOI: 10.48550/ARXIV.2012.13516. URL: https://arxiv.org/abs/2012.13516v1.

[58] Philipp Görz et al. "Systematic Assessment of Fuzzers using Mutation Analysis". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4535–4552. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gorz.

[59] Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. *PEP 617 – New PEG parser for CPython*. https://www.python.org/dev/peps/pep-0617/. Accessed: 2024-01-24. 2020.

[60] Nikolas Havrikov and Andreas Zeller. "Systematically Covering Input Structure". In: *ASE 2019*. Nov. 2019. URL: https://publications.cispa.saarland/2971/.

[61] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Nov. 2020). DOI: 10.1145/3428334. URL: https://doi.org/10.1145/3428334.

[62] Adrian Herrera et al. "Seed Selection for Successful Fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 230–243. ISBN: 9781450384599. DOI: 10.1145/3460319.3464795. URL: https://doi.org/10.1145/3460319.3464795.

[63] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[64] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

[65] Matthias Höschele and Andreas Zeller. "Mining Input Grammars from Dynamic Taints". In: *IEEE/ACM Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 720–725. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970321. URL: http://doi.acm.org/10.1145/2970276.2970321.

[66] Chaojian Hu et al. "File Parsing Vulnerability Detection with Symbolic Execution". In: *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*. 2012, pp. 135–142. DOI: 10.1109/TASE.2012.13.

[67] Hui Huang et al. “Protocol Knowledge Combined Directed Symbolic Execution for Binary Programs”. In: *2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control*. 2013, pp. 120–124. DOI: 10.1109/IMCCC.2013.32.

[68] Graham Hutton. “Higher-order functions for parsing”. In: *Journal of Functional Programming* 2.3 (1992), pp. 323–343. DOI: 10.1017/S0956796800000411.

[69] IBM. *Get started with COBOL*. https://developer.ibm.com/languages/cobol/. Accessed: 2021-09-18. 2021.

[70] IETF Trust and the persons identified as the document authors. *URI Fragment Identifiers for the text/csv Media Type*. https://datatracker.ietf.org/doc/html/rfc7111. Accessed: 2022-03-26. 2014.

[71] IETF Trust and the persons identified as the document authors. *The JavaScript Object Notation (JSON) Data Interchange Format*. https://www.rfc-editor.org/rfc/rfc8259.html. Accessed: 2022-03-26. 2017.

[72] ISO/IEC. *Programming languages — C; ISO/IEC9899:2017 [Working Document]*. https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf. Accessed: 2021-09-07, 6.5.3.2 Address and indirection operators, Page 64. 2021.

[73] ISO/IEC. *Programming languages — C; ISO/IEC9899:2017 [Working Document]*. https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf. Accessed: 2021-09-07. 2021.

[74] James Ramm and Contributors. *csv_parser - C library for parsing CSV files*. https://github.com/JamesRamm/csv_parser. Accessed: 2018-10-25, the original repository does not exist anymore as of 2022-03-26, but the used version can still be accessed in the LFUZZER artifact publication at https://dl.acm.org/do/10.1145/3406885/full/, e.g. in the folder ./lfuzzer-data/1/chains/samples/csv/csv_parser/. 2018.

[75] Konrad Jamrozik and Andreas Zeller. “DroidMate: a robust and extensible test generator for Android”. In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. MOBILESoft ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 293–294. ISBN: 9781450341783. DOI: 10.1145/2897073.2897716. URL: https://doi.org/10.1145/2897073.2897716.

[76] Justin Meiners. *Embeddable lisp interpreter written in C*. https://github.com/justinmeiners/lisp-interpreter. Accessed: 2019-03-19. 2019.

[77] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: 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, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. New York, NY: Plenum Press, New York, 1972, pp. 85–103. ISBN: 0-306-30707-3.

[78] Kartik Talwar. *Tiny-C Compiler*. https://gist.github.com/KartikTalwar/3095780. Accessed: 2018-10-25. 2018.

[79] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. URL: https://doi.org/10.1145/3243734.3243804.

[80] Donald E Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145. DOI: 10.1007/BF01692511. URL: https://doi.org/10.1007/BF01692511.

[81] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. "Learning Highly Recursive Input Grammars". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2021, pp. 456–467. DOI: 10.1109/ASE51524.2021.9678879.

[82] laf-intel and the LAF LLVM Passes Developers. *LAF LLVM Passes*. Code: https://gitlab.com/laf-intel/laf-llvm-pass/tree/master, Referenced Blogpost: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. Accessed: 2022-04-16. 2021.

[83] Caroline Lemieux et al. "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 919–931. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00085. URL: https://doi.org/10.1109/ICSE48619.2023.00085.

[84] Guangcheng Liang et al. "Effective Fuzzing Based on Dynamic Taint Analysis". In: *2013 Ninth International Conference on Computational Intelligence and Security*. 2013, pp. 615–619. DOI: 10.1109/CIS.2013.135.

[85] Linux Manual Page Contributors. *feature_test_macros(7) — Linux manual page*. https://man7.org/linux/man-pages/man7/feature_test_macros.7.html. Accessed 28-August-2023. 2023.

[86] LLVM Contributors and Chad Rosier. *[frontend] Fix how the frontend handles -fno-inline*. https://github.com/llvm/llvm-project/commit/9c76d24f9c562045aea28198ab0dcd0e81f37380. Accessed 04-September-2023. 2012.

[87] LLVM Project. *Clang 4 documentation - clang - the Clang C, C++, and Objective-C compiler*. https://releases.llvm.org/4.0.1/tools/clang/CommandGuide/clang.html. Accessed: 2023-08-30. 2017.

[88] LLVM Project. *LLVM Language Reference Manual*. https://releases.llvm.org/4.0.1/docs/LangRef.html. Accessed: 2024-01-29. 2017.

[89] LLVM Project. *LLVM's Analysis and Transform Passes – -reg2mem: Demote all values to stack slots*. https://releases.llvm.org/4.0.1/docs/Passes.html. Accessed: 2022-05-01. 2017.

[90] LLVM Project. *Low Level Virtual Machine (LLVM) - llvm/tools/opt/opt.cpp*. https://github.com/llvm/llvm-project/blob/449c3ef93afc7a668eb35e67a83717453e28b25a/llvm/tools/opt/opt.cpp#L106C4-L106C4. Accessed: 2023-07-08; because this cli argument is hidden, we reference the cli parser here. 2017.

[91] LLVM Project. *opt - LLVM optimizer*. https://releases.llvm.org/4.0.1/docs/CommandGuide/opt.html. Accessed: 2024-02-17. 2017.

[92] LLVM Project. *The ModulePass class*. https://releases.llvm.org/4.0.1/docs/WritingAnLLVMPass.html#the-modulepass-class. Accessed: 2022-03-05. 2017.

[93] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-07-16. 2021.

[94] LLVM Project. *LLVM Bitcode File Format*. https://llvm.org/docs/BitCodeFormat.html. Accessed: 2024-02-17. 2024.

[95] llvm-admin team. *The LLVM Compiler Infrastructure*. https://llvm.org. Accessed: 2024-02-17. 2024.

[96] Andreas Löscher and Konstantinos Sagonas. "Targeted Property-Based Testing". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 46–56. ISBN: 9781450350761. DOI: 10.1145/3092703.3092711. URL: https://doi.org/10.1145/3092703.3092711.

[97] Andreas Löscher and Konstantinos Sagonas. "Automating Targeted Property-Based Testing". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 70–80. DOI: 10.1109/ICST.2018.00017.

[98] Jinxin Ma, Tao Zhang, and Puhan Zhang. "Enhancing fuzzing with a minimum set solver". In: *2015 Seventh International Conference on Advanced Computational Intelligence (ICACI)*. 2015, pp. 23–26. DOI: 10.1109/ICACI.2015.7184730.

[99] Valentin Jean Marie Manès et al. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: 10.1109/TSE.2019.2946563.

[100] Björn Mathis. "Dynamic Tainting for Automatic Test Case Generation". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 436–439. ISBN: 9781450350761. DOI: 10.1145/3092703.3098233. URL: https://doi.org/10.1145/3092703.3098233.

[101] Björn Mathis. *Dynamic tainting on LLVM bitcode*. eng. Saarbrücken, 2017. URL: http://primo-fe.mpi-klsb.mpg.de:1701/permalink/f/1nnuu09/cim01_aleph000124311.

[102] Björn Mathis, Rahul Gopinath, and Andreas Zeller. "Learning Input Tokens for Effective Fuzzing". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 27–37. ISBN: 9781450380089. DOI: 10.1145/3395363.3397348. URL: https://doi.org/10.1145/3395363.3397348.

[103] Björn Mathis, Rahul Gopinath, and Andreas Zeller. *Replication Package for LFuzzer - Learning Input Tokens for Effective Fuzzing*. URL: https://doi.org/10.1145/3406885.

[104] Björn Mathis et al. "Detecting information flow by mutating input data". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 263–273. DOI: 10.1109/ASE.2017.8115639.

[105] 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: ACM, June 2019, pp. 548–560. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314651. URL: http://doi.acm.org/10.1145/3314221.3314651.

[106] Phil McMinn. "Search-based software test data generation: a survey". In: *Software Testing, Verification and Reliability* 14.2 (2004), pp. 105–156. DOI: https://doi.org/10.1002/stvr.294. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.294. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294.

[107] Jonathan Metzman et al. "FuzzBench: An Open Fuzzer Benchmarking Platform and Service". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1393–1403. ISBN: 9781450385626. DOI: 10.1145/3468264.3473932. URL: https://doi.org/10.1145/3468264.3473932.

[108] Michael Sperber and R. Kent Dybvig and Matthew Flatt and Anton van Straaten (Editors); Richard Kelsey, William Clinger, Jonathan Rees (Editors, Revised^5 Report on the Algorithmic Language Scheme); Robert Bruce Findler, Jacob Matthews (Authors, formal semantics). *Revised^6 Report on the Algorithmic Language Scheme*. http://www.r6rs.org/final/r6rs.pdf. Accessed: 2022-03-26. 2022.

[109] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[110] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. "The Relevance of Classic Fuzz Testing: Have We Solved This One?" In: *IEEE Transactions on Software Engineering* 48.6 (2022), pp. 2028–2039. DOI: 10.1109/TSE.2020.3047766.

[111] Mozilla Foundation. *JavaScript language resources*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources. Accessed: 2022-03-26. 2022.

[112] OpenJS Foundation and JSON Schema Contributors. *JSON Schema*. https://json-schema.org. Accessed 15-July-2023. 2023.

[113] Oracle. *Chapter 10. Arrays*. https://docs.oracle.com/javase/specs/jls/se16/html/jls-10.html#jls-10.4. Accessed: 2021-09-07. 2021.

[114] Oracle. *Module java.xml*. https://docs.oracle.com/en/java/javase/17/docs/api/java.xml/module-summary.html. Accessed: 2021-09-24. 2021.

[115] Rohan Padhye, Caroline Lemieux, and Koushik Sen. "JQF: Coverage-Guided Property-Based Testing in Java". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 398–401. ISBN: 9781450362245. DOI: 10.1145/3293882.3339002. URL: https://doi.org/10.1145/3293882.3339002.

[116] Rohan Padhye et al. "Semantic Fuzzing with Zest". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 329–340. ISBN: 9781450362245. URL: https://doi.org/10.1145/3293882.3330576.

[117] panda.re Authors and Contributors. *PANDA is an open-source Platform for Architecture-Neutral Dynamic Analysis*. https://panda.re. Accessed: 2022-03-05. 2022.

[118] Van-Thuan Pham et al. "Smart Greybox Fuzzing". In: *IEEE Transactions on Software Engineering* 47.9 (2021), pp. 1980–1997. DOI: 10.1109/TSE.2019.2941681.

[119] pyparsing and Contributors. *PyParsing – A Python Parsing Module*. https://github.com/pyparsing/pyparsing. Accessed: 2022-01-09. 2022.

[120] Python Software Foundation. *Built-in Exceptions*. https://docs.python.org/3/library/exceptions.html#IndexError. Accessed: 2022-04-12. 2022.

[121] Python Software Foundation. *10. Full Grammar specification*. https://docs.python.org/3/reference/grammar.html#full-grammar-specification. Accessed: 2024-01-24. 2024.

[122] qemu Authors and Contributors. *QEMU - A generic and open source machine emulator and virtualizer*. https://www.qemu.org. Accessed: 2022-03-05. 2022.

[123] Sanjay Rawat et al. “VUzzer: Application-aware Evolutionary Fuzzing.” English. In: *Network and Distributed System Security Symposium (NDSS), 2017*. Vol. 17. Internet Society, Feb. 2017, pp. 1–14. DOI: 10.14722/ndss.2017.23404.

[124] Jesse Ruderman. *Introducing jsfunfuzz*. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz. Accessed: 2024-02-12. 2007.

[125] Michael Schröder. “Grammar Inference for Ad Hoc Parsers”. In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 38–42. ISBN: 9781450399012. DOI: 10.1145/3563768.3565550. URL: https://doi.org/10.1145/3563768.3565550.

[126] 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.

[127] Koushik Sen and Gul Agha. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools”. In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 419–423. ISBN: 978-3-540-37411-4.

[128] Koushik Sen and Gul Agha. “A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs”. In: *Hardware and Software, Verification and Testing*. Ed. by Eyal Bin, Avi Ziv, and Shmuel Ur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 166–182. ISBN: 978-3-540-70889-6.

[129] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. URL: http://www1.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/cute.pdf.

[130] Serge Guelton. *Toward _FORTIFY_SOURCE parity between Clang and GCC*. https://developers.redhat.com/blog/2020/02/11/toward-_fortify_source-parity-between-clang-and-gcc. Accessed 28-August-2023. 2020.

[131] Bhargava Shastry et al. “Static Program Analysis as a Fuzzing Aid”. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Marc Dacier et al. Cham: Springer International Publishing, 2017, pp. 26–47. ISBN: 978-3-319-66332-6.

[132] Hannes Sochor, Flavio Ferrarotti, and Daniela Kaufmann. “Fuzzing-Based Grammar Inference”. In: *Model and Data Engineering*. Ed. by Philippe Fournier-Viger, Ahmed Hassan, and Ladjel Bellatreche. Cham: Springer Nature Switzerland, 2023, pp. 72–86. ISBN: 978-3-031-21595-7.

[133] Hannes Sochor, Flavio Ferrarotti, and Daniela Kaufmann. "Fuzzing-based grammar learning from a minimal set of seed inputs". In: *Journal of Computer Languages* 78 (2024), p. 101252. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2023.101252. URL: https://www.sciencedirect.com/science/article/pii/S259011842300062X.

[134] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. http://dinosaur.compilertools.net/yacc/index.html. Accessed: 2021-09-24. 2021.

[135] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *Network and Distributed System Security Symposium*. Vol. 16. 2016, pp. 1–16.

[136] Terence Parr and The Antlr Contributors. *ANTLR*. https://www.antlr.org/index.html. Accessed: 2021-09-14. 2021.

[137] The Clang Contributors. *Clang – Parser.cpp*. https://github.com/llvm/llvm-project/blob/llvmorg-12.0.1/clang/lib/Parse/Parser.cpp. Accessed: 2021-09-22. 2021.

[138] The ClusterFuzz Authors. *ClusterFuzz GitHub Repository*. https://github.com/google/clusterfuzz. Accessed: 2021-09-07. 2021.

[139] The Fortran Developers. *Fortran – High-performance parallel programming language*. https://fortran-lang.org. Accessed: 2021-09-18. 2021.

[140] The GCC Contributors. *GCC – c-parser.c*. https://github.com/gcc-mirror/gcc/blob/master/gcc/c/c-parser.c. Accessed: 2021-09-22. 2021.

[141] The gcovr Authors. *gcovr*. https://gcovr.com/en/4.1/index.html. Accessed: 2023-07-12. 2018.

[142] The KLEE Team. *Publications and Systems Using KLEE*. https://klee.github.io/publications/. Accessed: 2021-09-12. 2021.

[143] The Lisp Authors. *Common Lisp*. https://lisp-lang.org. Accessed: 2021-09-18. 2021.

[144] The Python Authors. *string — Common string operations*. https://docs.python.org/3/library/string.html#string.printable. Accessed: 2023-03-12. 2023.

[145] The Python Contributors. *json — JSON encoder and decoder*. https://docs.python.org/3/library/json.html. Accessed: 2021-09-24. 2021.

[146] The Python Contributors. *xml.etree.ElementTree — The ElementTree XML API*. https://docs.python.org/3/library/xml.etree.elementtree.html. Accessed: 2021-09-24. 2021.

[147] Vadim Zaytsev. *Grammar Zoo*. https://slebok.github.io/zoo/. Accessed: 2024-03-26. 2021.

[148] Leslie G. Valiant. “General context-free recognition in less than cubic time”. In: *Journal of Computer and System Sciences* 10.2 (1975), pp. 308–315. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(75)80046-8. URL: https://www.sciencedirect.com/science/article/pii/S0022000075800468.

[149] Junjie Wang et al. “Skyfire: Data-Driven Seed Generation for Fuzzing”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[150] Junjie Wang et al. “Superion: Grammar-Aware Greybox Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081.

[151] Ryan Whelan, Tim Leek, and David Kaeli. “Architecture-Independent Dynamic Information Flow Tracking”. In: *Compiler Construction*. Ed. by Ranjit Jhala and Koen De Bosschere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–163. ISBN: 978-3-642-37051-9.

[152] Wikipedia Contributors. *Fuzzing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=1158518468. Accessed 06-July-2023, 21:28. 2023.

[153] Zhangwei Xie et al. “CSEFuzz: Fuzz Testing Based on Symbolic Execution”. In: *IEEE Access* 8 (2020), pp. 187564–187574. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3030798.

[154] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. *Csmith Website*. https://embed.cs.utah.edu/csmith/. Accessed: 2021-09-08. 2021.

[155] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 283–294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532. URL: https://doi.org/10.1145/1993316.1993532.

[156] Wei You et al. “SLF: Fuzzing without Valid Seed Inputs”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 712–723. DOI: 10.1109/ICSE.2019.00080.

[157] Michał Zalewski. *afl-fuzz: making up grammar with a dictionary in hand*. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html. Accessed: 2023-10-11. 2015.

[158] Michał Zalewski and Contributors. *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/README.txt. Last Accessed: 2023-11-19. 2013.

[159] Michał Zalewski and Contributors. *American Fuzzy Lop - Source Code*. https://lcamtuf.coredump.cx/afl/releases/afl-2.52b.tgz. Accessed: 2024-02-01. 2017.

[160] Michał Zalewski and Contributors. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/. Accessed: 2018-01-28. 2018.

[161] Andreas Zeller et al. *The Fuzzing Book*. Accessed: 2021-07-16 14:52:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/.

Appendix

In the following, we show the full expression parser as used in our examples. This version of the expression parser does not contain a tokenization phase.

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>

char input[100];
int input_size;
int pos;

int strt();
int expr();
int term();
int atom();
int integer();
int num();

int skip_whitespace() {
    while(input[pos] == ' ') {
        pos++;
    }
    return pos;
}

int num(){
    printf("NUM: Parsing %c at pos %d\n", input[pos], pos);
    if (input[pos] >= '0' && input[pos] <= '9') {
        pos++;
        return 1;
    } else {
        return 0;
    }
}

int non_whitespace_integer() {
    printf("INT_NON_WHITESPACE: Parsing %c at pos %d\n",
        input[pos], pos);
    if (num()) {
```

```
37          non_whitespace_integer();
38          return 1;
39      } else {
40          return 0;
41      }
42  }
43
44  int integer() {
45      printf("INT: Parsing %c at pos %d\n", input[pos], pos);
46      skip_whitespace();
47      if (num()) {
48          non_whitespace_integer();
49          return 1;
50      } else {
51          return 0;
52      }
53  }
54
55  int atom() {
56      printf("ATOM: Parsing %c at pos %d\n", input[pos], pos);
57      skip_whitespace();
58      if (input[pos] == '(' ||
59          (pos + 4 < input_size && !strncmp("sin(", input + pos,
            ↪  4)) ||
60          (pos + 4 < input_size && !strncmp("cos(", input + pos,
            ↪  4))
61          ) {
62          if (input[pos] == '(') {
63              pos++;
64          } else {
65              pos += 4;
66          }
67          if (expr()) {
68              skip_whitespace();
69              printf("ATOM2: Parsing %c at pos %d\n", input[pos],
                ↪  pos);
70              if (input[pos] == ')') {
71                  pos++;
72                  return 1;
73              } else {
74                  printf(") not closed!\n");
75                  return 0;
76              }
77          } else {
78              return 0;
79          }
80      }
81      return integer();
82  }
```

```
83
84  int term() {
85      printf("TERM: Parsing %c at pos %d\n", input[pos], pos);
86      skip_whitespace();
87      if (atom()) {
88          printf("TERM2: Parsing %c at pos %d\n", input[pos], pos);
89          skip_whitespace();
90          if (input[pos] == '*' || input[pos] == '/') {
91              pos++;
92              return term();
93          }
94          printf("TERM3: Parsing %c at pos %d\n", input[pos], pos);
95          return 1;
96      }
97      return 0;
98  }
99
100 int expr() {
101     printf("EXPR: Parsing %c at pos %d\n", input[pos], pos);
102     skip_whitespace();
103     if (term()) {
104         skip_whitespace();
105         if (input[pos] == '+' || input[pos] == '-') {
106             pos++;
107             return expr();
108         } else {
109             return 1;
110         }
111     }
112     return 0;
113 }
114
115 int strt() {
116     if (expr()) {
117         skip_whitespace();
118         if (input[pos] == '\0') {
119             return 1;
120         }
121     }
122     return 0;
123 }
124
125 int main(int argc, char** argv) {
126     int in = read(0, input, 99);
127     input[in] = '\0';
128     pos = 0;
129     input_size = strlen(input);
130     int result = strt();
131     printf("Got: %s\n", input);
```

```
    printf("Parsed: ");
    for (int i = 0; i < pos; i++) {
        printf("%c", input[i]);
    }
    printf("\n");
    if (result) {
        printf("Valid!\n");
    } else {
        printf("Invalid!\n");
    }
    return !result;
}
```

Appendix A-I: A sample parser for our initial grammar in Figure 2.2.

Next, we show how the expression parser could be implemented if a tokenization phase is included:

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>

char input[100];
int input_size;
int pos;

enum lex_token{PLUS, MINUS, MULT, DIV, PAREN_L, PAREN_R,
    SIN, COS, NUM, WS, UNDEF, END};

int strt();
int expr();
int term();
int atom();
int integer();
int num();

enum lex_token token = UNDEF;

int skip_whitespace() {
    while(input[pos] == ' ') {
        pos++;
    }
    return pos;
}

void next_token_non_whitespace() {
```

```
29      if (input[pos] == '+') {
30          pos++;
31          token = PLUS;
32      } else if (input[pos] == '-') {
33          pos++;
34          token = MINUS;
35      } else if (input[pos] == '*') {
36          pos++;
37          token = MULT;
38      } else if (input[pos] == '/') {
39          pos++;
40          token = DIV;
41      } else if (input[pos] == '(') {
42          pos++;
43          token = PAREN_L;
44      } else if (input[pos] == ')') {
45          pos++;
46          token = PAREN_R;
47      } else if (input[pos] >= '0' && input[pos] <= '9') {
48          pos++;
49          token = NUM;
50      } else if (pos + 4 < input_size &&
51                 !strncmp("sin(", input + pos, 4)) {
52          pos += 4;
53          token = SIN;
54      } else if (pos + 4 < input_size &&
55                 !strncmp("cos(", input + pos, 4)) {
56          pos += 4;
57          token = COS;
58      } else if (input[pos] == '\0') {
59          token = END;
60      } else if (input[pos] == ' ') {
61          pos++;
62          token = WS;
63      } else {
64          // lexing error
65          printf("Undef token at %d: %c", pos, input[pos]);
66          token = UNDEF;
67      }
68  }
69
70  void next_token() {
71      skip_whitespace();
72      next_token_non_whitespace();
73  }
74
75  int num(){
76      printf("NUM: Parsing %c at pos %d\n", input[pos], pos);
77      printf("TOKEN: %d\n", token);
```

```
78      if (token == NUM) {
79          next_token_non_whitespace();
80          return 1;
81      } else {
82          return 0;
83      }
84  }
85
86  int non_whitespace_integer() {
87      printf("INT_NON_WHITESPACE: Parsing %c at pos %d\n",
88          input[pos], pos);
89      if (num()) {
90          non_whitespace_integer();
91          return 1;
92      } else if (token != UNDEF) {
93          if (token == WS) {
94              next_token();
95          }
96          return 1;
97      } else {
98          return 0;
99      }
100 }
101
102 int integer() {
103     printf("INT: Parsing %c at pos %d\n", input[pos], pos);
104     if (num()) {
105         non_whitespace_integer();
106         return 1;
107     } else {
108         return 0;
109     }
110 }
111
112 int atom() {
113     printf("ATOM: Parsing %c at pos %d\n", input[pos], pos);
114     if (token == PAREN_L || token == SIN || token == COS) {
115         next_token();
116         if (expr()) {
117             printf("ATOM2: Parsing %c at pos %d\n", input[pos],
                ↪  pos);
118             if (token == PAREN_R) {
119                 next_token();
120                 return 1;
121             } else {
122                 printf(") not closed!\n");
123                 return 0;
124             }
125         } else {
```

```
126             return 0;
127         }
128     }
129     return integer();
130 }
131
132 int term() {
133     printf("TERM: Parsing %c at pos %d\n", input[pos], pos);
134     if (atom()) {
135         printf("TERM2: Parsing %c at pos %d\n", input[pos], pos);
136         if (token == MULT || token == DIV) {
137             next_token();
138             return term();
139         }
140         printf("TERM3: Parsing %c at pos %d\n", input[pos], pos);
141         return 1;
142     }
143     return 0;
144 }
145
146 int expr() {
147     printf("EXPR: Parsing %c at pos %d\n", input[pos], pos);
148     if (term()) {
149         if (token == PLUS || token == MINUS) {
150             next_token();
151             return expr();
152         } else {
153             return 1;
154         }
155     }
156     return 0;
157 }
158
159 int strt() {
160     next_token();
161     if (expr()) {
162         if (token == END) {
163             return 1;
164         }
165     }
166     return 0;
167 }
168
169 int main(int argc, char** argv) {
170     int in = read(0, input, 99);
171     input[in] = '\0';
172     pos = 0;
173     input_size = strlen(input);
174     int result = strt();
```

```
175     printf("Got: %s\n", input);
176     printf("Parsed: ");
177     for (int i = 0; i < pos; i++) {
178         printf("%c", input[i]);
179     }
180     printf("\n");
181     if (result) {
182         printf("Valid!\n");
183     } else {
184         printf("Invalid!\n");
185     }
186     return !result;
187 }
```

Appendix A-II: A sample parser with lexer for our initial grammar in Figure 2.2.

Finally, for reference, we include the original arithmetic expression grammar from the Fuzzingbook [161] which we adapted for our needs in the main part of this thesis:

```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer>
                | <integer>.<integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Appendix A-III: A sample grammar for parsing an arithmetic expression. This grammar is the original arithmetic expression grammar from the Fuzzingbook [161] presented in the chapter *"Fuzzing with Grammars"*.