UNIVERSITÄT
DES
SAARLANDES

Leveraging Input Features for Testing and Debugging

by
Alexander Kampmann

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

Saarbrücken, 2022

Day of Colloquium	19th December 2022
Dean of the Faculty	Univ.-Prof. Dr. Jürgen Steimle
Chair of the Committee	Prof. Dr. Sebastian Hack
Reporters	
First Reviewer	Prof. Dr. Andreas Zeller,
Second Reviewer	Prof. Dr. Lars Grunske,
Third Reviewer	Prof. Dr. Thorsten Holz,
Academic Assistant	Dr. Dominic Steinhoefel

Abstrakt

Bei der Fehlersuche in Softwaresystemen ist es wichtig, die Ursache des Fehlers zu verstehen. Wie können wir den Entwickler hierbei unterstützen, und den Prozess des Programmverstehens automatisieren?

Das Verhalten eines Programms wird durch seinen Input bestimmt. Ein typischer Schritt bei der Fehlersuche ist daher, zu verstehen wie der Input das Programmverhalten beeinflust. In dieser Arbeit präsentiere ich Alhazen, ein Programm, das eine Feedback Loop zwischen einem Decision Tree und einen Testgenerator einrichtet, um eine Erklärung dafür zu generieren, welche Inputs ein bestimmtes Programmverhalten auslösen. Das kann benutzt werden, um einen Programmfehler automatisch zu diagnostizieren.

Es wird evaluiert, ob Alhazen eingesetzt werden kann um zusätzliche Inputs zu generieren und Inputs, die den Fehler auslösen, zu erkennen. Die Arbeit enhält erste Schritte für eine Nutzerstudie, die zeigen soll, ob Alhazen Softwareentwicklern bei der Fehlersuche hilft.

Die Ergebnisse zeigen, dass Alhazen in beiden Szenarien effektiv ist, und deuten darauf hin, dass maschinelle Lernverfahren beim Programmverstehen helfen können.

Alhazen wirft die Frage auf, ob die Betrachtung von Input Features andere Möglichkeiten bietet. Diese Thesis präsentiert Basilisk, ein Werkzeug, das Testgenerierung auf bestimmte Teile des Inputs fokussiert. Basilisk wird als eigenständiges Tool evaluiert, und eine Integration mit Alhazen wird ausprobiert. Die Ergebnisse sind vielversprechend, zeigen aber auch Probleme mit der Idee auf.

Insgesamt zeigt die Arbeit das Protential von Input Features.

Abstract

When debugging software, it is paramount to understand why the program failed in the first place. How can we aid the developer in this process, and automate the process of understanding a program failure?

The behavior of a program is determined by its inputs. A common step in debugging is to determine how the inputs influences the program behavior. In this thesis, I present Alhazen, a tool which combines a decision tree and a test generator into a feedback loop to create an explanation for which inputs lead to a specific program behavior. This can be used to generate a diagnosis for a bug automatically.

Alhazen is evaluated for generating additional inputs, and recognizing bug-triggering inputs. The thesis makes first advances towards a user study on whether Alhazen helps software developers in debugging.

Results show that Alhazen is effective for both use cases, and indicate that machine-learning approaches can help to understand program behavior.

Considering Alhazen, the question what else can be done with input features follows naturally. In the thesis I present Basilisk, a tool which focuses testing on a specific part of the input. Basilisk is evaluated on its own, and an integration with Alhazen is explored. The results show promise, but also some challenges.

All in all, the thesis shows that there is some potential in considering input features.

Contents

Chapter 1

Problem Statement

A program fails. This sets a process in motion: A bug report is being written, someone checks whether the problem can be reproduced. Someone investigates how severe the problem is and assigns a priority to the bug. And, in the end, someone needs to figure out *why* the program failed. This can be a lengthy and complicated endeavor, however, it is a prerequisite for solving the problem. Attempts to solve a problem which is not yet full understood lead to incomplete fixes, or introduce new bugs.

Zeller [77] suggests the scientific method for debugging: Pose a hypothesis of why the program fails, and conduct an experiment to refute the hypothesis. If you cannot refute it, the hypothesis must be true, and explain the program failure.

In this thesis, I am going to *automate* this process. I will present a tool, named Alhazen, which can automatically determine the circumstances of a failure. Alhazen provides an explanation for a given software behavior, based on features extracted from the program input. Alhazen is not limited to explanations for crashes or bugs, any observable behavior of the program under test can be explained.

Alhazen can be used in three different scenarios:

Predicting program behavior If a program fails with a specific input, this can have severe consequences. For example, consider the observation by a twitter user in Figure 1.1: When he joined a wireless network with the id `"%p%s%s%s%s%n"`, the wifi capabilities of his iPhone were permanently disabled. Once a model for a specific bug is learned, Alhazen can be used to recognize inputs which trigger this bug, and prevent them from being processed. This can serve as a first line of defence and buy the developers time to come up with a proper fix.

Generating additional inputs Several techniques in software engineering, for example statistical debugging[65], automated program repair[73, 42] or specification mining[35, 6, 17] require large sets of inputs. Alhazen can be used to generate more inputs which trigger a specific behavior.

Understanding program behavior Last but not least, Alhazen's automatically generated hypothesis may be helpful to software developers. They provide

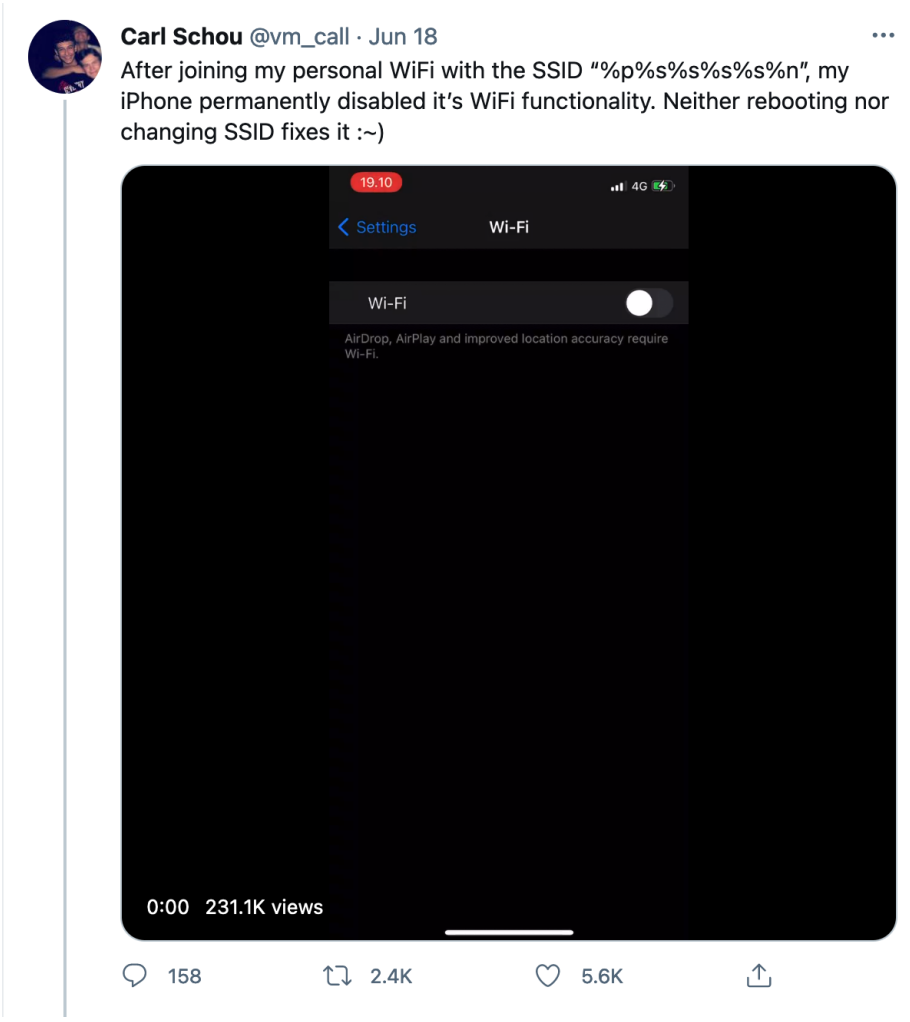


Figure 1.1: A tweet about problems with a malformed SSID on iOS[1].

a good understanding of the behavior in question, which may help to come up with a fix.

1.1 Thesis Structure

ALHAZEN consists of two components, which each receive their own chapter in this thesis:

1. In Chapter 2, I will extract features from program inputs, and train a decision tree learner to recognize bug-triggering inputs. This is, however, not a full automation of the scientific method, it cannot yet perform experiments to refute a hypothesis.
2. In Chapter 3, I will therefore present a test generator which can extract predicate sets from a decision tree, and generate samples which violate

(or fulfill) those predicates. This tool can then be used in a feedback loop with the decision tree learner: It generates the inputs needed to refute the hypothesis.

3. In Chapter 5, I explore another way to work with partial inputs: Testing of specific program parts, based on which input parts they process. While this is a project on its own, I also explore how it can be combined with ALHAZEN.

The evaluation in Chapter 2 and Chapter 3 addresses predicting program behavior and generating additional inputs. In Chapter 4, I discuss how to evaluate whether ALHAZEN helps software developers to understand software behavior. This requires a user study, and Chapter 4 presents insights into how to conduct such a study. Finally, Chapter 6 presents related work and concludes the thesis.

1.2 How to read this thesis?

The material in this thesis falls in three categories:

Contributions The thesis presents the results of my scientific inquiry. A reader who follows the flow of the text meets all those results, ordered such that they build on each other.

Background Some of the ideas presented in this thesis are based on well-known concepts of computer science, or work done by others. If understanding these is necessary to follow the thesis, they are presented within the flow of the text, and marked as such. Sections which contain only those definitions say so in their opening paragraph, and can be skipped if the reader feels familiar with the concepts presented. Definitions, theorems and observations are *boxed*.

> **Boxed Content**
> Throughout the thesis, boxed content will be repeated when it is referred to.

Therefore, a reader who skipped such a section can look at the definitions later. All related work which was not directly used within my projects is described in Section 6.1.

Definitions Just as definitions contributed by others, my own definitions are *boxed*, and will be repeated when they are referred to.

1.3 Motivating Example

In this section, I will illustrate how ALHAZEN works with an example. All steps will be discussed in detail later. This section just serves to give a brief overview.

Within the example, I consider a simple calculator program. This program reads inputs as specified by the grammar in Figure 1.2. The input consists of a function name, one of `"tan"`, `"cos"`, `"sin"` or `"sqrt"`, and a number. The program then outputs the result of applying the given function to the given

⟨*start*⟩ → ⟨*function*⟩ `"("` ⟨*number*⟩ `")"`

⟨*function*⟩ → `"tan"` | `"cos"` | `"sin"` | `"sqrt"`

⟨*number*⟩ → `"0"` | `/-[0-9+](.[0-9+])?/` | `/[0-9+](.[0-9+])?/`

Figure 1.2: The input grammar for a simple calculator.

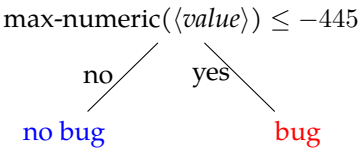


Figure 1.3: The decision tree obtained by ALHAZEN0 on the example.

number, e.g. when presented with the input `"sqrt(4)"`, it outputs 2. Let's suppose the program crashes on the input `"sqrt(-900)"`. The astute reader may have a theory on what causes the problem already: The result of $\sqrt{-900}$ is a complex number, and the implementation of `"sqrt"` probably does not support this. This theory is in fact correct, and I will now outline how ALHAZEN comes to the same conclusion.

But before I discuss the full ALHAZEN approach, as presented in Chapter 3, I will present its predecessor, ALHAZEN0 which will be presented in Chapter 2.

ALHAZEN0 starts with two input samples: One input which triggers the bug, and one input which does not, say `"sqrt(9)"` and `"sqrt(-900)"`. ALHAZEN0 now proceeds in 3 steps.

Parsing ALHAZEN0 uses the context-free grammar in Figure 1.2 to decompose the inputs into individual input parts. Each input part corresponds to an element of the grammar.

Feature Extraction ALHAZEN0 extracts numeric *features* from the decomposed inputs. ALHAZEN0 for each input element, ALHAZEN0 introduces a feature which is 0 if the element is not present, and 1 otherwise. It also considers numeric interpretation of input elements. Within the example, that is e.g. -900 for the substring `"-900"`, and the length of input elements. E.g 4 for `"sqrt"`.

Posing a hypothesis ALHAZEN0 uses a *decision tree learner* to generate a hypothesis of why the program fails on the failing input. The retrieved decision tree can be seen in Figure 1.3. This tree proposes that the program fails if the number is smaller than -445. This is not correct, it ignores the fact that the failure only occurs if the function is `"sqrt"`, and it assumes that e.g. `"sqrt(-9)"` would work out, which it does not. However, all observations made so far, `"sqrt(-900)"` and `"sqrt(9)"`, are explained by this hypothesis.

Within Chapter 3, ALHAZEN0 will be extended to ALHAZEN, which solves this problem, by applying the concept of scientific inquiry: ALHAZEN performs additional experiments, and uses the outcome to refine the hypothesis. Within

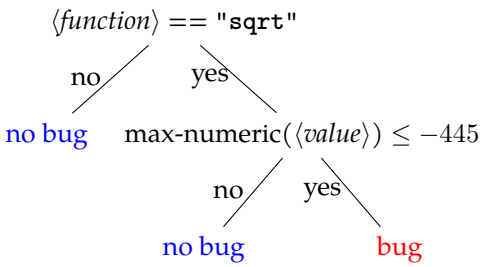


Figure 1.4: The decision tree obtained in the first iteration of ALHAZEN on the example.

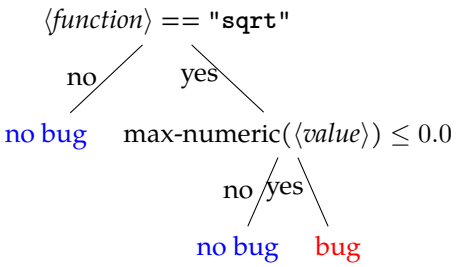


Figure 1.5: The decision tree obtained after 29 iterations of ALHAZEN on the example.

the example, this means that ALHAZEN generates two new inputs: One with a number larger than -445, and one with a smaller value. The hypothesis does not specify which function to use, so ALHAZEN chooses randomly. It may therefore come up with the inputs `"cos(-25)"` and `"cos(-500)"`.

Both inputs do not fail, and ALHAZEN again trains a decision tree, just as ALHAZEN0 did, to obtain a refined hypothesis. The new tree is shown in Figure 1.4. The learner now proposes that the function is relevant, it saw some data indicating that it is. The hypothesis still asks for the number to be smaller than -445, because there is no new data indicating something else.

By re-iterating the process of generating new inputs and re-training the decision tree, ALHAZEN can refine the hypothesis further. After 29 iterations, it provides the tree in Figure 1.5, which correctly names the failure circumstances: The function `"sqrt"` needs to be used, and the value needs to be smaller than 0.

Looking at intermediate results of those 29 iterations, ALHAZEN works on finding the correct boundary value. The decision tree learner always proposes a value in between the smallest non-failing and the largest failing value. E.g. the samples `"sqrt(-900)"` and `"sqrt(9)"` lead to a value of -445. ALHAZEN now tests values larger and smaller than -445, and, in the next iteration, comes up with a better boundary. Over several iterations, it arrives at 0.

1.4 Behavior of Interest and Predicate of Interest

While the example and discussion so far focused on ALHAZEN for debugging, ALHAZEN can be used to explain other program behaviors. The only requirement is that input samples can be labeled as triggering the behavior or not. In theory, input samples could be labeled by hand, however, as ALHAZEN requires quite a few input samples (this will be evaluated in Chapter 2), labeling them automatically is a better approach. In order to do so, I am using a *predicate of interest*.

> **Definition 0: Predicate of Interest**
> The *predicate of interest* is a predicate over observable program behavior. The program behavior which is recognized by the predicate of interest is the *behavior of interest*. If the program exhibits the behavior of interest while processing a specific input, this input is said to be *behavior-triggering*.

This definition is vague on purpose. It says little more than that the predicate of interest separates the space of all possible program executions into two classes, and that it can be calculated by *observing* a program run. However, the meaning of *observable* may depend on the applied instrumentations.

In this form, the definition allows for a wide range of debugging problems. Predicates of interest could consider execution times, crashes, or other properties.

Obviously, a good predicate of interest is paramount for ALHAZEN's performance. I will show what happens if an insufficient predicate is used in Section 3.3.3. For the time being, let's have a look at different options for writing such predicates.

1.4.1 General-Purpose Predicates of Interest

There are some behaviors which are unacceptable for every program. A predicate which recognizes such a behavior can be reused with multiple programs. I call it a general-purpose predicate of interest. s

Crash Oracles Most operating systems and hardware platforms provide some safeguards: They limit resource usage, detect it if a program tries to access memory areas that have not been assigned to this program, or detects division by zero. Program execution is discontinued if one of those checks triggers. If this happens, users usually say: "The program crashed." This is unacceptable in almost all situations. Besides the terrible user experience, programs which are just discontinued may leave data in a corrupt state. Therefore, such incidents can be regarded as a bug.

Timeout Oracles In many cases, there are requirements – or at least expectations – on the runtime of a program. Program runs which exceed the expected run time can be considered erroneous, which provides another, simple way to implement a predicate of interest.

Oracles Based On Error Messages Programs which detect error states themselves usually provide an error message. An analysis of the error message — in the easiest case string equality with an observed error message — can be used to recognize specific behaviors. This provides another option to implement predicates of interest.

A/B Oracles If there are two programs which do the same thing, one can be used as an oracle for the other. As an example, database engines which implement the SQL standard should provide the same output, if the database contents are the same, and they receive the same query. While this makes for good oracles, it is not actually a practical idea: The situation that there are two programs which are supposed to behave in the same way is just too uncommon. Even database engines, which I just used as an example, in practise often implement custom extensions to the SQL standard, which would make for enough of a difference to make an A/B oracle impractical. Still, practitioners should have this option in mind. In some cases, it may be possible to obtain an A/B oracle nevertheless. In the case of SQL engines, one could test the subset of SQL for which both databases behave in the same way. Another option is to build an alternative implementation which does not meet the same performance goals: A red-black tree, which can perform lookups in logarithmic time, can be tested against a simple list, which performs the same task in linear time. The two implementations differ in the expected performance, but not in the expected output. For testing on small data sets, the performance difference is acceptable. Similarly, a distributed database engine, which stores data on multiple, connected machines, could be tested against a deployment of the same engine on a single node. However, this approach has downsides: It can easily happen that the development effort in making the comparison program gets out of hand. Software engineers should make sure that the effort is justified for the advantages achieved. Also, every bug discovered may be in either the program under test, or the program used for comparison. This may pose additional challenges in debugging.

Regression Oracles Most software developers do not write a program from scratch, but improve upon an existing program. A new version of the program under test is created whenever a bug is fixed or a new feature is added. Those changes sometimes introduce new problems: Regression bugs. But then, an A/B oracle is easily available: The old version of the program can be used for comparison. If the behavior of the old version and the behavior of the new version are different, and the difference is not expected, that is, it is not related to a bug fix or new feature, this is a (newly introduced) bug. But then comparing the old and new behavior can be used as an oracle.

Regression oracles may be hard to implement: Small changes, e.g. the rewording of an error message, or an additional white space in the output, make a simple comparison of the output far to sensitive. On the other hand, such changes are usually easy to detect and ignore. Throughout the thesis, I use some regression oracles. All of those had actions taken to prevent this kind of problems.

Time-traveling Regression Oracles The specific situation of scientific experimentation allows for a different kind of regression oracles. If I use bugs which have already been fixed as evaluation subjects, I can also compare the program output with more recent versions of the program under test. While those clairvoyant regression oracles cannot be used in everyday software development [1], they simplify the development of predicates of interest, and therefore broaden the number of available subjects for this thesis.

Summary Here is an overview over predicates of interest that can be reused with multiple programs.

General-purpose Predicates of Interest
The following predicates of interest are reusable with different programs:

Crash Oracle Some error conditions, e.g. memory protection faults, are detected by the operating system, which, in turn, terminates the program. Crash Oracles consider this forceful termination as behavior of interest.

Timeout Oracle For many programs, a resonable maximal execution time can be defined. Timeout Oracles classify executions which exceed this maximum as buggy.

Error-message Oracle Many programs output detailed error messages. In this case, the presence (or absence) of a specific error message in the program output can be used as an oracle. This is especially useful for Java Programs, as the Java Runtime offers mechanisms for error checking, and outputs detailed stack traces if one of those checks fails. This allows a predicate of interest to check specifically for the line where an error occurs.

Regression Oracle As programs are being developed, new problems may be introduced in old code. A regression oracle checks whether the output of a program is the same as in an old version.

1.4.2 Specific Predicates of Interest

For some bugs, general-purpose predicates are not sufficient. In those cases, the tester need to be creative, and come up with an idea on how to write an oracle. The main goal is always to be economic in the development of the predicate: If the predicate of interest takes several days of implementation work, it is likely better to spend the time on writing a huge number of tests manually, or — in the context of this thesis — debug without using Alhazen. Still, this section presents some ideas on how to derive predicates of interest.

Property-based Oracles For some programs, there are properties of the output that can be used to check whether they operate correctly. Property-based testing[19] proposes to build oracles by checking such properties. If, within an

[1] More research in time-travel technology is required, but out of scope for this thesis.

online shopping application, you calculate the total of the wares selected by the user, a negative result is definitely a bug. As a second example, `grep` reports whether a specific pattern is contained within the input text, and outputs those lines of the input text which match the pattern. This means that every line of the output should be a line of the input: `grep` does not invent new text fragments. Identifying such properties can be difficult, but it makes for powerful predicates of interest.

Construction-based Oracles Construction-based Oracles are a special case of property-based oracles. As property-based oracles, they rely on a property of the output, however, they also require inputs which were crafted to make sure that this property exists.

In Chapter 4, I am going to use a program which extracts kernel version numbers from machine descriptors. As an example, when presented with the input `"FreeBSD 12.2-RELEASE amd64 GENERIC"`, the program reports that the string describes a FreeBSD kernel in a `"release"` build for major version 12 and minor version 2. It is simple to implement a program which accepts the inputs `"FreeBSD"`, `"Release"`, `"12"` and `"2"`, and outputs `"FreeBSD 12.2-RELEASE amd64 GENERIC"`. This program is the inverse to the program under test. Now, the program under test can be tested by applying its inverse to the output, and check whether I obtain the input again. As a second example, consider a program which sums up two numbers. Here, the function is not simply the inverse: `"6+6"`, `"2+10"`, or `"4+8"` all give an output of 12. However, I can craft inputs such that the output is known. E.g. if I provide an input $\frac{a}{2} + \frac{a}{2}$, I should receive a as an output. The construction-based oracle does not work for arbitrary program inputs, but only for those which follow the schema used by the construction part of the oracle. When using construction-based inputs with Alhazen, one would write a grammar which generates the input to the program which constructs the input, e.g. the kernel name and versions in the first example or the number a in the second example, rather than the input to the program under test. Practitioner's using this approach should be aware that they may miss bugs in samples that do not follow the schema of their constructed inputs.

1.5 State of the Art

The predicate of interest defines which behavior needs diagnosis. However, it does not provide any diagnosis itself. In the first part of this section, I will look at existing work that aims to generate a diagnosis to why a program behaves the way it does.

In the above, I remarked that Alhazen may generate additional samples, which may be useful for automated program repair. In the second part of this section, I will look closer at automated program repair, and explain how Alhazen's samples can help this field.

1.5.1 Debugging

In general, work in debugging focuses on helping software developers to fix program faults faster. They fall into two general families:

Input Reduction attempt to give the developer a smaller bug-triggering input sample.

Fault Localization aims to pinpoint the faulty program statements, telling the developer where to apply a fix.

A related field is **specification mining**, which aims to provide a model for all program behavior, rather than an explanation for one specific program behavior. Each approach will be examined in one of the following sections:

Input reduction

Input reduction approaches aim to generate a smaller input which triggers the same bug. The intuition is that smaller inputs mean smaller program runs, and therefore less program states to analyze.

The best-known algorithm in this area is delta-debugging[78]. This algorithm uses a divide and conquer approach: In each step, half of the input is removed, until removing any more character from the input means that the failure does not occur any longer. The approach was successful, Zeller [77] reports that it has made it into the instructions on how to write a bug report for the FireFox browser. The approach was later refined. Grammar-based delta debugging[50, 69] uses a grammar to parse the input, and always removes entire constituents. This way, each intermediate input is valid according to the grammar, and delta-debugging converges sooner.

Specification Mining

Many approaches which generate richer models come from *specification mining*. This field attempts to infer a specification for all program behavior, rather than just a specific behavior of interest.

Another family of approaches aims at generating models which give properties of behavior-triggering inputs. DDSET[25] is a variant of grammar-based delta debugging that outputs a specialisation of the grammar, which generates behavior-triggering inputs only. This approach will be further discussed in Section 6.1.4.

DAIKON[17] derives invariants from predefined templates. It generates all instances of those templates, filling in template variables with values observed from program runs, and reports resulting formulas if they are fulfilled for a sufficiently high number of observed program runs. For the example I used earlier, DAIKON would generate the invariant $x \geq 0$ in sqrt(x) if x was positive in all observed runs. DAIKON uses statistical methods to decide which formula remains, and is considered an invariant of the program under test, and which are removed.

Fault Localization

Besides program input values, researchers also attempted to find the code location that contains the bug. One approach to do so is via statistical analysis[65]: If a set of code locations is executed in every behavior-revealing run, but not in runs that do not exhibit the behavior, those code locations must be related to the

behavior. Statistical fault localization tools therefore attempt to establish a statistical correlation between code locations and program failures. There is some discussion about whether this is useful for programmers[53], but some works show the usefulness of those approaches for automated program repair[42, 73]. Chen et al. [9] apply this idea to a more coarse-grained idea of code location. Rather than line numbers, they determine which component in a large system is responsible for a failure. There approach is based on a decision tree, rather than probability analysis.

This idea can be combined with test generation. Rößler et al. [61] estblishes a relation between program behavior and code locations, and then attempts to generate tests which exercise those program locations. Johnson, Brun, and Meliou [37] takes a similar approach, using program input values and a large set of randomly generated tests.

1.5.2 Automated Program Repair

If a program is defective, the defect does not only need to be diagnosed, it also needs to be fixed. Therefore, *automated program repair* attempts to find a modification of the program which leads to correct behavior.

This problem statement can be broken down in two parts:

1. Searching for a modification of the program; and

2. identifying correct behavior

The later problem is related to my behavior of interest. Rather than identifying an existing behavior, automated program repair techniques need to define a non-existing behavior. The behavior which is correct, as opposed to the existing, incorrect, behavior. This problem is much harder than the one my predicates of interest solve. A predicate of interest needs to identify one specific behavior, and mark it as incorrect, whereas automated program repair techniques need statements over all program behaviors.

Writing such a specification is impossible in most cases, and therefore researchers use test cases instead. Correct behavior is the behavior which makes all test cases pass. This is where Alhazen combines with automated program repair. Alhazen can provide more test cases, it can even generate additional test cases, which show the behavior of interest. Those can be used in to identify the desired program behavior in automated program repair.

While relying on test cases is ubiquitous in automated program repair, techniques for searching for a modification of the program are more diverse. They, basically, fall into two families. The first is search-based approaches. Conceptually, those build the space of all possible modifications, and use a fitness function or ranking to select the best one. Obviously, no implementation actually enumerate all possible modifications. There are, potentially, infinitely many. They usually generate modifications with different techniques, and therefore sample the search space.

The first step in such an approach is to identify the area of the search space that needs modification. Here, automated program repair uses techniques from statistical debugging. As discussed in the previous section, those techniques attempt to identify the program locations that contain the reason for the defective

behavior. Automated program repair techniques than focus on modifications which modify this area of the program.

Actual repair candidates then can be derived by mutation of existing modifications[27], which lends itself to a genetic programming approach to searching the space of possible modifications[73]. Another approach is to move program fragments from elsewhere in the program into the defective location. This approach is based on the idea that source code is repetitive[59], and assumes the *competent programmer hypothesis*: It states that programmers likely supplied a correct solution to the same problem at a different location in the program.

Once a technique to sample repair candidates is chosen, those approaches need to find the best one. The first question here is which is the best candidate. Approaches may concentrate on the smallest modification or some notion of readability[40]. Search strategies range from genetic programming[73] over various heuristics (e.g. [42]) to simple random sampling[58].

The second family of techniques to identify repair candidates in literature is constraint-based repair. Approaches like SemFix[51] use symbolic execution to generate symbolic representations of correct behavior, and then apply program synthesis to generate functions which exhibit the symbolically described behavior. One could say that those techniques construct a repair candidate, rather than searching for it.

Chapter 2

Learning Explanations for Program Behavior

In this chapter, I will show how to use *evidence* in order to provide an *hypothesis* of why the program behaves the way it does. This idea is implemented in a tool called Alhazen0.

Alhazen0 uses a *context-free grammar* to decompose the program input and define *features*, numerically describing the input, from this decomposition. The features will be used to train a *decision tree*, a classifier which provides a human-readable explanation for its classification. This explanation is my hypothesis for why the program behaves the way it does.

Alhazen0 was designed to support two use cases:

Generating inputs Many techniques within software engineering, for example statistical debugging[65], automated program repair[73, 42] or specification mining[35, 6, 17] require large sets of behavior-triggering inputs. Alhazen0 can be used as a *generator* to synthezise such input sets.

Filtering inputs If a specially crafted input causes a program to misbehave, this can have severe consequences. Especially in situations where program interfaces are public, such a bug may be abused for Denial-of-service attacks. The models generated by Alhazen0 can be used to predict whether a given input is going to trigger such a bug. Then, the input can be rejected before it reaches the program. Therefore, Alhazen0 would protect the program against such malicious inputs, and by the software developers time to come up with a real fix for the problem.

In order to facilitate those use cases, I will evaluate whether Alhazen0 is capable of

1. generating more inputs which exhibit the behavior of interest
2. predicting whether a given input is going to exhibit the behavior of interest

⟨*start*⟩ → ⟨*function*⟩ `"("` ⟨*number*⟩ `")"`

⟨*function*⟩ → `"tan"` | `"cos"` | `"sin"` | `"sqrt"`

⟨*number*⟩ → `/-?[0-9]+(.[0-9]+)?/`

Figure 2.1: The grammar for the calculator example in Section 1.3.

2.1 Context-Free Grammars

This section presents definitions for grammars and the properties of grammars I am going to work with. Similar definitions can be found in many textbooks about theoretical computer science, so if you feel confident that you know the material, feel free to skip this section. Important definitions will be referenced whenever they are used.

As Alhazen0 relies on context-free grammars for input decomposition, I start by defining them.

Definition 1: Context-Free Grammar
A *context-free grammar* is a tuple (N, T, P, S), where N is a set of non-terminal symbols, T is a set of terminal symbols, P is a set of production rules and $S \in N$ is the start symbol.
A production rule $p \in P$ has the form $\langle A \rangle \rightarrow \langle c \rangle$, where $\langle c \rangle$ is one of the following control forms:

Reference A reference to a non-terminal symbol.

Terminal Symbol A terminal symbol.

Alternation A sequence of control forms $\langle C_1 \rangle$ | ...| $\langle C_n \rangle$ separated by |, with $n > 1$.

Concatenation A sequence of control forms $\langle C_1 \rangle$ ...$\langle C_n \rangle$, with $n > 1$.

Quantification A control form, called the subject of the quantification, annotated with `+`, `*` or `?`

P contains exactly one production rule of the form $\langle A \rangle \rightarrow \langle c \rangle$ for each $\langle A \rangle \in N$.

In the following, I will write $\langle C \rangle$ both for the non-terminal symbol $\langle C \rangle$, and a reference to $\langle C \rangle$. This is should not cause confusion, as non-terminals appear only on the right-hand side of production rules, and references occur only on the left-hand side. However, there may be more than one reference $\langle C \rangle$.

Most other formal definitions of context-free grammars allow more than one production rule $\langle A \rangle \rightarrow \langle c \rangle$ for each $\langle A \rangle \in N$. My definition allows at most one production rule per nonterminal. However, the following definitions are made such that the effect of two production rules $\langle A \rangle \rightarrow \langle C_1 \rangle$ and $\langle A \rangle \rightarrow \langle C_2 \rangle$ can be achieved with one production rule that uses an alternation: $\langle A \rangle \rightarrow \langle C_1 \rangle$ | $\langle C_2 \rangle$. Therefore, this definition is equally powerful, but some of the following definitions are simplified by this restriction.

Figure 2.1 shows the grammar for the calculator example from Section 1.3.

⟨*number*⟩ → `"-"`? ⟨*regex.1*⟩+ ⟨*regex.2*⟩?
⟨*regex.1*⟩ → `"0"` | ...| `"9"`
⟨*regex.2*⟩ → `"."` ⟨*regex.1*⟩+

Figure 2.2: The regular expression `/-?[0-9]+(.[0-9]+)?/`, represented as a context-free grammar.

As all grammars in the thesis, this grammar is given as a list of its production rules P. The non-terminal symbols can be derived from this presentation, as they are on the left-hand side of the production rules. On the right-hand side of a production rule, terminal symbols are written in quotes. The grammar representation does not give the start symbol. In many cases, the top-most production rule will be the start symbol, as is the case in the example, where ⟨*start*⟩ is the start symbol. If it is not the topmost production rule anyways, the start symbol can be determined from the context. For the grammar in Figure 2.1, it is

$$N = \{\langle start\rangle, \langle function\rangle, \langle number\rangle\}$$
$$T = \{\texttt{"("}, \texttt{")"}, \texttt{"tan"}, \texttt{"cos"}, \texttt{"sin"}, \texttt{"sqrt"}, \texttt{"-"}, \texttt{""."}, \texttt{"0"}, \ldots, \texttt{"9"}\}$$
$$S = \langle start\rangle$$

Figure 2.1 also shows syntactic sugar I allow myself throughout the thesis. The production rule for ⟨*number*⟩ is given as a regular expression, which is not allowed by the definition. However, every regular expression can be represented as a context-free grammar. Figure 2.2 shows how to do this for the regex within the example. In many cases, a regular expression provides a more concise representation, which is why I allow myself to use them when I write down grammars. All algorithms apply to context-free grammars only, and regular expressions would be re-written before applying them.

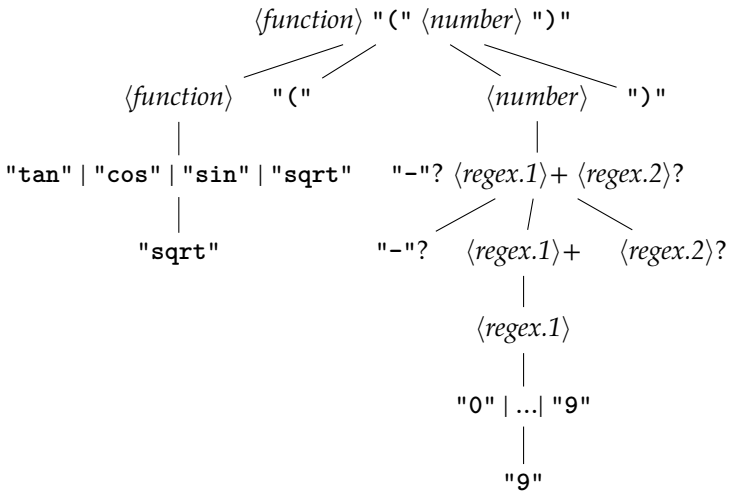


Figure 2.3: Parse tree for `"sqrt(9)"` within the grammar in Figure 2.1.

A context-free grammar describes a *language*, which is composed of all the words in the grammar.

Definition 2: Parse Trees and Words in the Language
A *parse tree* for a context-free grammar (N, T, P, S) is a tree where each node is labeled with a control form. The root node is labeled with a control form $\langle s \rangle$ such that $S \rightarrow \langle s \rangle$ is in P. If a node is labeled with

1. a reference $\langle A \rangle$, it has one child which is labeled with a control form $\langle C \rangle$, such that $\langle A \rangle \rightarrow \langle C \rangle$ is a production rule in P.
2. a terminal symbol, it has no children (it is a leaf node).
3. an alternation $\langle C_1 \rangle$ | ...| $\langle C_n \rangle$, it has one child, labeled with one of the $\langle C_i \rangle$'s.
4. a concatenation $\langle C_1 \rangle$ ...$\langle C_n \rangle$, it has n children, where the i-th child is labeled with $\langle C_i \rangle$.
5. a quantification, all its children are labeled with the subject of the quantification. The node has one or more children if the annotation is `+`, zero or more children if the annotation is `*` and zero or one children if the annotation is `?`.

The sequence of terminal symbols that consists of the labels of the leaves labeled with a terminal symbol of the parse tree in preorder is the *leaf word* of this tree. It can also be said that the parse tree *derives* its leaf word. All words which have valid parse trees, that is, trees formed according to the rules above, are *words of the language*.

Figure 2.3 gives the parse tree for the word `"sqrt(9)"` within the grammar in Figure 2.1. Each node is labeled with a control form, which means that,

for example, a node labeled with the reference ⟨*function*⟩ has a single child labeled with the alternation `"tan"` | `"cos"` | `"sin"` | `"sqrt"`. Many textbooks use a notation where this node gets a child labeled with `"sqrt"` directly, omitting the node labeled with the alternation. If those nodes are not relevant to the example at hand, I will do the same within this thesis. In Figure 2.3, I expanded the nodes below ⟨*number*⟩ as defined by the rewritten regex in Figure 2.2. For the remainder of the thesis, I will avoid a blow-up of the parse trees by allowing nodes to be labeled with a regex.

A parse tree is defined to have its root node labeled with the control form that the start symbol expands to. However, in some situations it is useful to reason about parse trees with different symbols at the root. Those trees follow the same definition as a parse tree, except for a different root symbol.

2.2 Representing a Grammar as a Graph

In the following (especially Section 3.2) I will argue about different properties of grammars. It turned out that a graph representation of grammars, as it was introduced by Havrikov and Zeller [31], simplifies many of those analyses and observations. Therefore, this section presents the definition of a grammar graph, which was originally introduced by Havrikov and Zeller [31], and some observations on the relationship between grammar graphs and parse trees.

Definition 3: The Control Forms of a Grammar
The set of *control forms of a grammar* G is the smallest set V such that

1. For all production rules $\langle A \rangle \rightarrow \langle C \rangle$ in G,
   the control form $\langle C \rangle$ is in V
2. If $\langle C \rangle$ is in V, and $\langle C \rangle$ is a concatenation $\langle C_0 \rangle$ ...$\langle C_n \rangle$,
   all $\langle C_i \rangle$ are in V
3. If $\langle C \rangle$ is in V, and $\langle C \rangle$ is an alternation $\langle C_0 \rangle$ | ...| $\langle C_n \rangle$,
   all $\langle C_i \rangle$ are in V
4. If $\langle C \rangle$ is in V, and $\langle C \rangle$ is a quantification with the subject $\langle D \rangle$,
   $\langle D \rangle$ is in V

All terminal symbols in the grammar are in V, as they appear on the right-hand side of production rules. However, non-terminal symbols are never in V. V contains control forms, so a reference to a non-terminal may be in V, but not the non-terminal itself. As explained before, I use the same notation for non-terminals and references to non-terminals. There may be more than one reference to a non-terminal $\langle C \rangle$, in which case V contains all those references.

⟨*Expression*⟩	→	⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
⟨*UnaryExpression*⟩	→	⟨*Literal*⟩ | ⟨*Invocation*⟩
⟨*Invocation*⟩	→	⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"`
⟨*Function*⟩	→	`"sqrt"` | `"cos"` | `"tan"` | `"sin"`
⟨*Literal*⟩	→	`/[1-9][0-9]*/`

Figure 2.4: A more complex grammar for a calculator. The start symbol is ⟨*Expression*⟩.

Definition 4: Grammar Graph
The *Grammar Graph* for a grammar G is a directed graph. Its nodes are the control forms of the grammar G. Two nodes $\langle C_i \rangle$ and $\langle C_j \rangle$ are connected with an edge if

1. $\langle C_i \rangle$ is a reference, and there is a production rule $\langle C_i \rangle \rightarrow \langle C_j \rangle$; or
2. $\langle C_i \rangle$ is an alternation, and $\langle C_j \rangle$ is part of the sequence; or
3. $\langle C_i \rangle$ is a concatenation, and $\langle C_j \rangle$ is part of the sequence; or
4. $\langle C_i \rangle$ is a quantification, and $\langle C_j \rangle$ is its subject

Figure 2.4 shows a slightly more complex grammar for a calculator. The grammar graph for this grammar is given in Figure 2.5. The central node within the graph is ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩. The start symbol of the grammar, ⟨*Expression*⟩, expands to this control form. The node is not the start symbol ⟨*Expression*⟩, because in this context, ⟨*Expression*⟩ is a non-terminal symbol, not a control form.

Starting at the node ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩, there are outgoing edges to ⟨*UnaryExpression*⟩ and ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩, as those are the members of the alternation. This node has incoming edges from three nodes, and all three are references to ⟨*Expression*⟩. The three ⟨*Expression*⟩ nodes within the grammar graph each corresponds to one ⟨*Expression*⟩ reference within the grammar. The two in the upper right corner of the figure correspond to the references in ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩, and the third one to the reference in ⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"`.

Relationship of Grammar Graphs and Parse Trees

In this section, I will explore the relationship between the grammar graph and parse trees.

Theorem 0:
For every path in a parse tree, there is a path with the same node labels in the grammar graph.

Within a parse tree, each node is labeled with a control form. The same

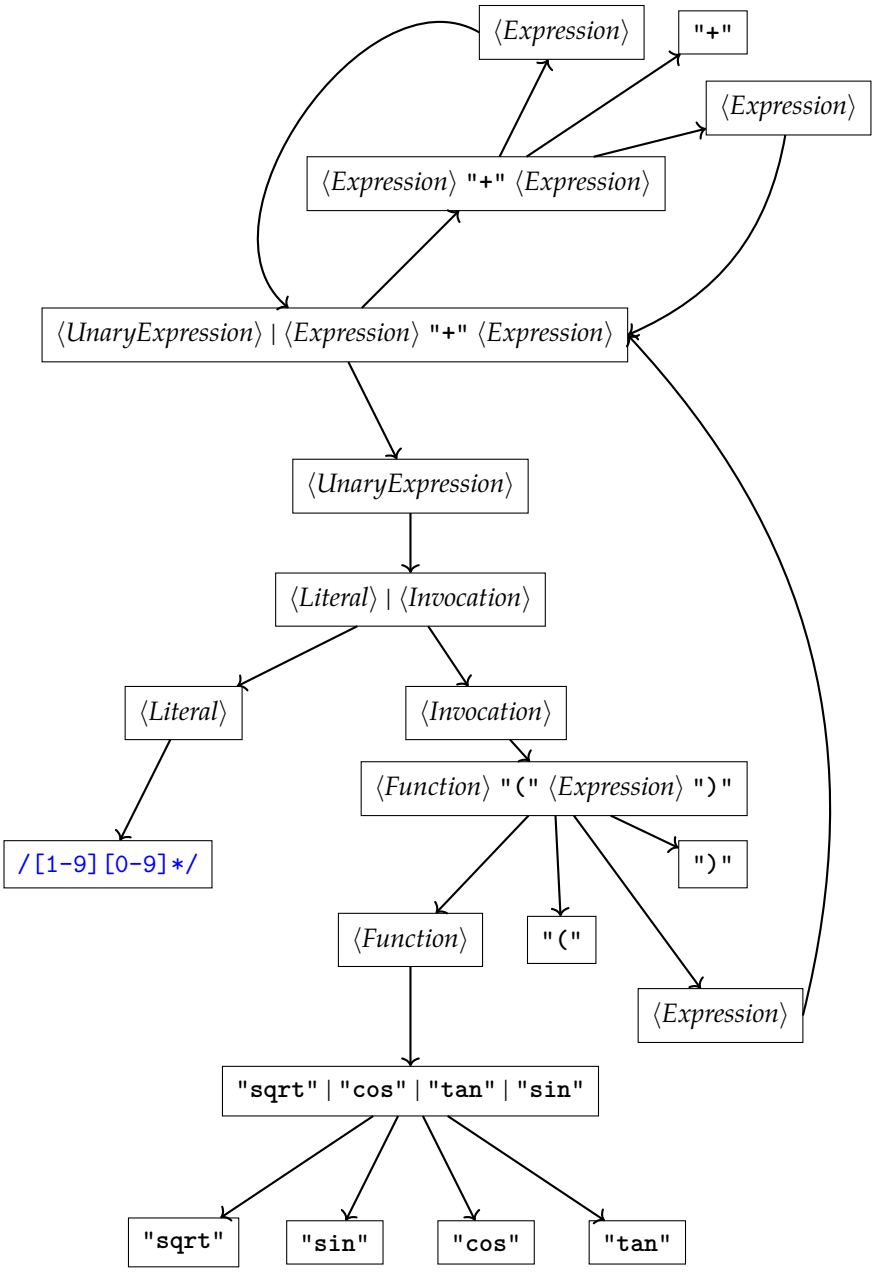


Figure 2.5: The grammar graph for the grammar in Figure 2.4.

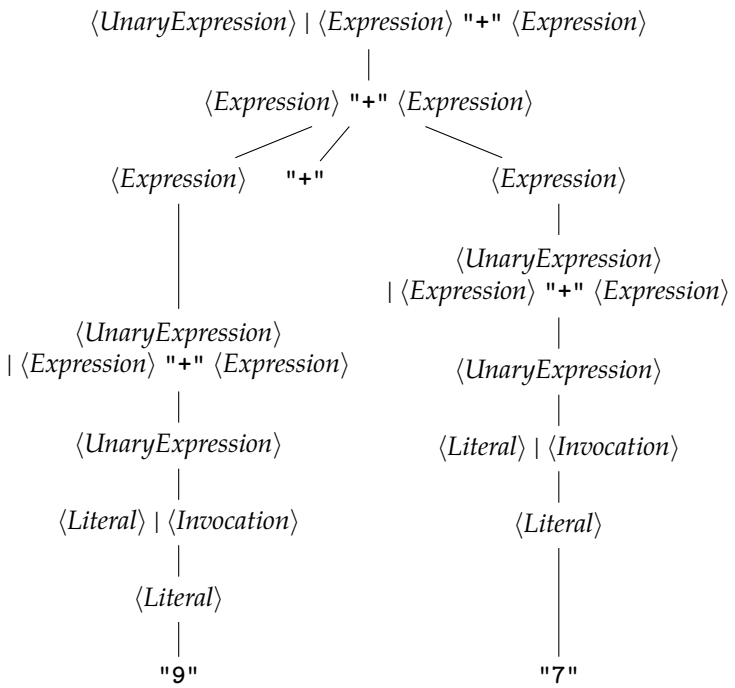


Figure 2.6: The parse tree for "9+7" within the grammar in Figure 2.4.

control form is a node within the grammar graph. Looking at the definition for a parse tree, and comparing to the definition of the grammar graph, it is obvious that if there is a child node labeled with ⟨*C*⟩j below a node labeled with a control from ⟨*C*⟩i in a grammar graph, there is an edge from $\langle C_i \rangle$ to $\langle C_j \rangle$ in the grammar graph.

As an example, look at the parse tree for "9+7", which can be found in Figure 2.6. The root of the parse tree is labeled with ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ "+" ⟨*Expression*⟩, the same as the central node in the grammar graph. The only child is ⟨*Expression*⟩ "+" ⟨*Expression*⟩, which corresponds to the node in the top-right corner of Figure 2.5. This node in the parse tree has three children, each of them corresponds to one outgoing edge in the grammar graph. The node labeled with ⟨*Expression*⟩ is followed by another node labeled with ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ "+" ⟨*Expression*⟩. The grammar graph has back-edges from the nodes labeled with ⟨*Expression*⟩ to the central node.

This relationship also applies the other way round.

Theorem 1:
Let G be a grammar with start symbol $\langle S \rangle$, and a production rule $\langle S \rangle \rightarrow \langle s \rangle$.
For every path in the grammar graph that starts in $\langle s \rangle$, a parse tree which contains a path with the same node labels can be constructed.

While not strictly necessary, assume that the path in the grammar graph ends with a terminal node. If this is not the case, add a suffix to achieve it. Now,

for every node which is labeled with a concatenation, add the required number of child nodes and complete the required subtrees. The result is a valid parse tree.

There are two more observations to be made here:

Theorem 2:
Let G be a grammar with start symbol $\langle S \rangle$, and a production rule $\langle S \rangle \rightarrow \langle s \rangle$. Let $\langle c \rangle$ be some other control form in G.
If there is no path within the grammar graph that starts in $\langle s \rangle$ and contains $\langle c \rangle$, $\langle c \rangle$ cannot be part of any parse tree.

The reason is that any path in a parse tree also a path in the grammar graph, and vice versa. So if there is no path in the grammar graph, there is no parse tree which contains such a path as well, but in a parse trees, all nodes are always reachable from the root node. So if there is no path with leads to $\langle c \rangle$ in a parse tree, then $\langle c \rangle$ is not part of any parse tree.

From now on, all theorems will only hold for control forms which can be part of a parse tree, unless noted otherwise.

The second observation is about concatenations.

Theorem 3:
Let $\langle p_0 \rangle \rightarrow ... \rightarrow \langle p_n \rangle$ be a path in a parse tree. For one $i < n$, let $\langle p_i \rangle$ be a concatenation $\langle p_i \rangle \rightarrow \langle c_1 \rangle ... \langle c_m \rangle$.
For each $j < m$, there is a path $\langle p_0 \rangle \rightarrow ... \rightarrow \langle p_i \rangle \rightarrow \langle c_j \rangle$ in the parse tree.
$\langle p_i \rangle$ has m outgoing edges in the grammar graph.

This is because within a parse tree, each concatenation always has as many children as there are members in its sequence. But then, a subpath which ends in a concatenation always has more than one suffix in the parse tree. The last part of the theorem, concerning the number of outgoing edges in the grammar graph, needs no proof, as it follows directly from the definition.

To give an example, consider the path ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩ → ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩ within Figure 2.6. As can be seen in the figure, the node ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩ has three children. Therefore, there are three paths which have this path as a prefix:

1. ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → ⟨*Expression*⟩

2. ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → `"+"`

3. ⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
   → ⟨*Expression*⟩

Also, the node ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩ has three outgoing edges, as can be seen in Figure 2.5.

Strongly Connected Components in Grammar Graphs

An interesting property of grammar graphs is that they are cyclic. Therefore, I will talk about strongly connected components within grammar graphs.

> **Definition 5: Strongly Connected Component**
> A *strongly connected component* in a directed graph is a set of nodes such that each node within the set is reachable from every other node within the set.

Within the example in Figure 2.5, there is a strongly connected component which contains all nodes except the nodes labeled with terminal symbols and the nodes labeled with ⟨*Function*⟩ and ⟨*Literal*⟩.

In fact, nodes labeled with terminal symbols have no outgoing edges. But then, there can be no path from such a node to any other node, and therefore, I can observe that:

> **Theorem 4:**
> A node labeled with a terminal symbol can never be in a strongly connected component.

In a grammar graph, having a strongly connected component means that there is an infinite number of path. If there is a node which belongs to the strongly connected component in a path, there is a path which loops through the strongly connected component, back to this node. This path can be added to the original path, therefore generating a longer path through the grammar graph. Due to Theorem 1, each of those paths can be used to construct a parse tree. We can rule out that the same parse tree is constructed from all those path, as the parse tree would be infinite otherwise. But then, there is an infinite number of parse trees, and therefore the language described by the grammar contains an infinite number of words.

However, each path within a parse tree needs to end in a terminal symbol, and terminal symbols cannot be part of a strongly connected component. Therefore, each strongly connected component in a grammar graph needs an exit.

> **Definition 6: Exit from a Strongly Connected Component on the Grammar Graph**
> Within a strongly connected component in a grammar graph, I call an edge which leads to a node which is not part of the strongly connected component an *exit*. I call the edge a *real exit* if it does not originate in a (node labeled with a) concatenation.

In Figure 2.5, the edge from ⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"` to ⟨*Function*⟩ is an exit, because ⟨*Function*⟩ is not part of the strongly connected component. It is not a real exit, because it originates in a concatenation. The edge from ⟨*Expression*⟩ to ⟨*UnaryExpression*⟩ is not an exit. Both nodes are within the strongly connected component. The edge from ⟨*Literal*⟩ | ⟨*Invocation*⟩ to ⟨*Literal*⟩ is a real exit, because ⟨*Literal*⟩ is not in the strongly connected component, and ⟨*Literal*⟩ | ⟨*Invocation*⟩ is not a concatenation.

As observed before, whenever a path in the parse tree goes through a concatenation, there are other paths with the same prefix which go through another

child of the concatenation. But then, if one of the outgoing edges of the concatenation is not an exit, it means that there is still an unterminated path after going through the concatenation. If all outgoing edges of a concatenation were exits, the concatenation is not in a strongly connected component (which contradicts the assumption that all edges are exits.) Therefore, the exit on a strongly connected component in the grammar graph cannot originate in a concatenation (and needs to be a real exit[1]).

Theorem 5:
Given a path in a parse tree which originates at the root node and ends in a leaf node, it either does not use any control form that is in a strongly connected component in the grammar graph, or contains a real exit.

Otherwise, there would be a path in the parse tree which is infinite, and the derivation tree would not describe a valid word.

Number of Leaf Words

Using the grammar graph, I can count the number of leaf words for a control form $\langle C \rangle$.

Definition 7: Number of leaf words $\#_d(\langle C \rangle)$
$\#_d(\langle C \rangle)$ is the number of valid parse trees rooted in $\langle C \rangle$. It is called the *number of leaf words*.

The number of leaf words of a control form $\langle C \rangle$ can be calculated as

$$\#_d(\langle C \rangle) = \begin{cases} \infty & \text{if } \langle C \rangle \text{ is part of a strongly connected component in the grammar graph.} \\ 1 & \text{if } \langle C \rangle \text{ is a terminal symbol.} \\ \#_d(\langle A \rangle) & \text{if } \langle C \rangle \text{ is a reference and there is a production rule } \langle C \rangle \rightarrow \langle A \rangle \\ \sum_{i=0}^{n} \#_d(\langle C_i \rangle) & \text{if } \langle C \rangle \text{ is an alternation } \langle C_0 \rangle \mid ... \mid \langle C_n \rangle. \\ \prod_{i=0}^{n} \#_d(\langle C_i \rangle) & \text{if } \langle C \rangle \text{ is a concatenation } \langle C_0 \rangle ... \langle C_n \rangle. \\ 1 + \#_d(\langle D \rangle) & \text{if } \langle C \rangle \text{ is a quantification with the subject } \langle D \rangle \text{ and the annotation ?.} \\ \infty & \text{if } \langle C \rangle \text{ is a quantification and the annotation is + or *.} \end{cases}$$

For practical reasons, I also set $\#_d(\langle C \rangle) = \infty$ if the value cannot be represented with a 64 bit two's complement integer.

Within the more complex calculator grammar in Figure 2.4, $\langle \textit{Function} \rangle$ has 4 leaf words. They are `"sqrt"`, `"sin"`, `"cos"` and `"tan"`. The formula gives the same number, as each terminal symbol gives 1, and the alternation gives the sum of $\#_d$ for each of its options. As observed before, there is an infinite number of paths through a strongly connected component, and therefore an infinite number of leaf words. This is why $\#_d(\langle \textit{Expression} \rangle) = \infty$, $\#_d(\langle \textit{Invocation} \rangle) = \infty$ and $\#_d(\langle \textit{UnaryExpression} \rangle) = \infty$. $\langle \textit{Literal} \rangle$ has infinite leaf words, because the regex contains a *, which turns into a quantification with the * annotation when it is rewritten to a context-free grammar.

[1] This is why the definition makes a distinction between exits and real exits

Listing 2.1: An algorithm to calculate the shortest derivation for all control forms in a grammar CFG.

```
// Input: A grammar CFG
Shortest-Derivation(CFG):
      Let T be a stack
      Push all terminal symbols in CFG onto T
      While T is not empty:
            ⟨C⟩ := pop(T)
            old_min := SHORTEST-DERIVATION(⟨C⟩) or undefined,
                  if SHORTEST-DERIVATION(⟨C⟩) is undefined
            if ⟨C⟩ is a terminal symbol:
                  SHORTEST-DERIVATION(⟨C⟩) := 1
            if ⟨C⟩ is a reference:
                  Let ⟨C⟩ → ⟨B⟩ be part of the grammar
                  if SHORTEST-DERIVATION(⟨B⟩) is defined:
                        SHORTEST-DERIVATION(⟨C⟩) := SHORTEST-DERIVATION(
                            ⟨B⟩) + 1
            if ⟨C⟩ is a quantification annotated with +:
                  Let ⟨B⟩ be the subject of ⟨C⟩
                  if SHORTEST-DERIVATION(⟨B⟩) is defined:
                        SHORTEST-DERIVATION(⟨C⟩) := SHORTEST-DERIVATION(
                            ⟨B⟩) + 1
            if ⟨C⟩ is a quantification annotated with ? or *:
                  SHORTEST-DERIVATION(⟨C⟩) := 1
            if ⟨C⟩ is a concatenation ⟨C1⟩ …⟨Cn⟩:
                  if all SHORTEST-DERIVATION(⟨Ci⟩) are defined:
                        SHORTEST-DERIVATION(⟨C⟩) := 1 + max_{i=1}^{n} ⟨Ci⟩
            if ⟨C⟩ is an alternation ⟨C1⟩ | …| ⟨Cn⟩:
                  Let min be the smallest SHORTEST-DERIVATION(⟨Ci⟩) for
                      all i where SHORTEST-DERIVATION(⟨Ci⟩) is defined
                  SHORTEST-DERIVATION(⟨C⟩) := min + 1
            if old_min is undefined or SHORTEST-DERIVATION(⟨C⟩) ≠
                old_min:
                  For all ⟨A⟩ such that there is an edge ⟨A⟩ →
                      ⟨C⟩ in the grammar graph:
                        push ⟨A⟩ to T
```

Shortest Derivation

Throughout the thesis, I need a way to talk about the size of parse trees. Havrikov and Zeller [31] introduced the shortest derivation, which is a useful metric to do so.

> **Definition 8: Shortest Derivation** $s_d(\langle C \rangle)$
> $s_d(\langle C \rangle)$ is the minimal depth of a valid parse tree rooted in $\langle C \rangle$.

Calculating the shortest derivation While this definition is used in Havrikov and Zeller [31], they do not give an algorithm to calculate the shortest derivation. Therefore, I present my own algorithm in Listing 2.1. Looking at the source code of Havrikov and Zeller [31], theirs seems to be similar, but not identical.

The main part of the algorithm is the loop in Line 5. This loop processes control forms from the stack `T` until the stack is empty. Line 4 pushes all terminal symbols to the stack, and in Line 28, all predecessors of an already processed control form are pushed to the stack. Those are the control forms which can occur as a parent of the processed control form in a parse tree.

Within the loop, the algorithm therefore either receives a terminal symbol, or a node which has at least one of its children processed previously. The shortest derivation for a terminal node is 1 (Line 10), because, according to the definition of a parse tree, nodes labeled with a terminal symbol do not have children. The same applies for quantifications which are labeled with * or ? (Line 19). For quantifications labeled with +, the shortest derivation of the subject is known, because the quantification would not be in `T` if it had not been calculated before (Line 15).

The interesting case are alternations and concatenations. A alternation or concatenation $\langle C \rangle$ has multiple control forms $\langle C_i \rangle$ in its sequence, but the loop invariant guarantees prior calculation of the shortest derivation for just one of them. Therefore, values for `SHORTEST-DERIVATION`($\langle C_i \rangle$) may be unknown in Line 24 and Line 21. However, control forms are pushed to the stack for each of their children, and therefore there will be an execution of the body of the loop where all children are defined. This argument does not apply in the case of strongly connected components. In a strongly connected component, there is a path which connects $\langle C \rangle$ with $\langle C \rangle$ again. But then, all control forms on this path have a predecessor with an undefined shortest derivation, because the shortest derivation of $\langle C \rangle$ would be required to calculate it. However, Theorem 5 says that there is an alternation with a real exit somewhere in the strongly connected compnent. For this alternation, the shortest derivation for the real exit will be known, and then Line 24 can assign a shortest derivation for this alternation. At this point, the algorithm ignores undefined values for the other children. But then, there is a path from this alternation to every other control form in the strongly connected component, and therefore each control form in the component has the shortest derivation for one child defined eventually. This is enough to have a value for all control forms but concatenations. For concatenations, the algorithm only provides a value for the shortest derivation if the values for all $\langle C_i \rangle$ are defined. As all children are either an exit, and therefore defined independent of the strongly connected component, or connected to a real exit, concatenations will be defined eventually. But then, all control forms in a strongly connected component have a shortest derivation, and the algorithm is done.

Generating the shallowest parse tree Within their paper, Havrikov and Zeller [31] also have an algorithm to generate the shallowest parse tree. I will later use the same, and therefore, I am presenting the algorithm in Listing 2.2 as well. The algorithm follows the definition of a parse tree: For each node, it generates the subnodes required by the definition. In cases where the definition gives multiple options, that is alternations and quantifications, the algorithm uses the option which has the smallest shortest derivation.

Listing 2.2: Algorithm for generating the shallowest parse tree for a given grammar.

```
// Input: A control form ⟨c⟩ a partial parse tree t
Shortest-Generate(⟨c⟩, t):
      Create a new node n in t and label it with ⟨c⟩
      if ⟨c⟩ is a Reference:
            find a production rule ⟨c⟩ → ⟨b⟩
            add Shortest-Generate(⟨b⟩, t) as a child to n
      if ⟨c⟩ is an alternation ⟨c1⟩ | …| ⟨cn⟩:
            select a ⟨cj⟩ such that Shortest-Derivation(⟨cj⟩) is minimal
            within ⟨c1⟩ …⟨cn⟩
            add Shortest-Generate(⟨cj⟩) as a child to n
      if ⟨c⟩ is a concatenation ⟨c1⟩ …⟨cn⟩:
            for all ⟨cj⟩:
                  add Shortest-Generate(⟨cj⟩) as a child to n
      if ⟨c⟩ is a Quantification ⟨c′⟩+:
            add Shortest-Generate(⟨c′⟩) as a child to n
      if ⟨c⟩ is a Quantification ⟨c′⟩*:
            no operation
      if ⟨c⟩ is a Quantification ⟨c′⟩?:
            no operation
      return n
```

2.3 Ambiguity

There is one property of grammars which requires special attention. That is ambiguity.

> **Definition 9: Ambiguity**
> A grammar is *ambiguous* if there are at least two parse trees that derive the same word.

Parse trees can be derived in two processes:

1. Starting with a word, a parse tree which derives this word can be created. This process is called *parsing*.
2. Starting with the grammar, a parse tree can be created from scratch. This process is called *generating*.

Generator algorithms are usually unaffected by ambiguity: They generate one parse tree and do not care about possible ambiguous interpretations.

Many of the grammars I use in my work stem from fuzzing campaigns, therefore, they were written for generating exclusively, and making them unambiguous was not a concern for their authors.

The situation, however, is different for parsing, that is, constructing parse trees which fit a given language word. If there is more than one parse tree for a given word, the parser needs to make a decision on which one to output. This decision making process is called *disambiguation*.

Disambiguation can be done within the grammar, that is, by rewriting the grammar until all words have just a single parse tree, or within the parser, using

⟨*BooleanExpression*⟩	→	⟨*Identifier*⟩ | ⟨*BooleanLiteral*⟩ | ⟨*BooleanExpression*⟩ `"||"` ⟨*BooleanExpression*⟩ | ⟨*BooleanExpression*⟩ `"&&"` ⟨*BooleanExpression*⟩
⟨*Identifier*⟩	→	`/[a-z*]/`
⟨*BooleanLiteral*⟩	→	`"false"` | `"true"`

Figure 2.7: An ambiguous grammar for simple boolean expressions. The derivation rules for ⟨*Identifier*⟩ and ⟨*BooleanLiteral*⟩ are ambiguous.

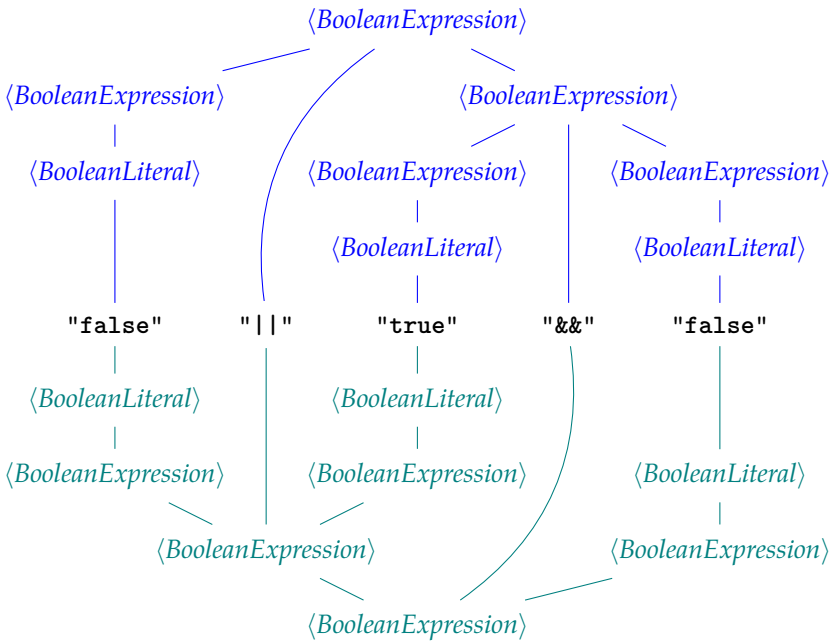


Figure 2.8: Two possible parse trees (one in blue on top, the other one in teal at the bottom) for `"false || true && false"`

some properties that cannot be expressed within the grammar. Consider the grammar for simple boolean expressions in Figure 2.7. This grammar has lots of potential for ambiguity. First of all, the ⟨*BooleanExpression*⟩ is ambiguous. The expression `"false || true && false"` has two parse trees, as depicted in Figure 2.8.

This problem can be solved with a grammar rewrite: An additional production ⟨*UnaryExpression*⟩, as presented in Figure 2.9, can be introduced. There can only be a unary expression behind an operator, forcing the parser to choose the interpretation which is shown in the blue part on top of Figure 2.8.

However, this does not solve all cases of ambiguity. The input `"false"` can be interpreted as ⟨*BooleanLiteral*⟩ or ⟨*Identifier*⟩, and the same goes for `"true"`. This means that `"false || true && false"` still has 8 interpretations: Each occurrence of `"true"` or `"false"` could either be an identifier, or a literal. Such an ambiguity, which exists in nearly all programming languages, would usually

⟨*BooleanExpression*⟩	→	⟨*BooleanExpression*⟩ `"||"` ⟨*UnaryExpression*⟩ | ⟨*BooleanExpression*⟩ `"&&"` ⟨*UnaryExpression*⟩
⟨*UnaryExpression*⟩	→	⟨*Identifier*⟩ | ⟨*BooleanLiteral*⟩
⟨*Identifier*⟩	→	`/[a-z*]/`
⟨*BooleanLiteral*⟩	→	`"false"` | `"true"`

Figure 2.9: A grammar for simple boolean expressions. The production rules for ⟨*Identifier*⟩ and ⟨*BooleanLiteral*⟩ are ambiguous.

⟨*Module*⟩	→	(⟨*Statement*⟩ | ⟨*Declaration*⟩)*
⟨*Expression*⟩	→	⟨*FunctionExpression*⟩
⟨*FunctionExpression*⟩	→	`"function"` ⟨*Identifier*⟩ `"("` ⟨*Arguments*⟩ `")"` ⟨*FunctionBody*⟩
⟨*Statement*⟩	→	⟨*Expression*⟩ (`";"` | `"\n"`) | ...
⟨*Declaration*⟩	→	⟨*FunctionDeclaration*⟩ | ...
⟨*FunctionDeclaration*⟩	→	`"function"` ⟨*Identifier*⟩ `"("` ⟨*Arguments*⟩ `")"` ⟨*FunctionBody*⟩ `"\n"`

Figure 2.10: A subset of the grammar for JavaScript.

be solved by prohibiting the terms `"true"` and `"false"` as identifier names. However, this cannot be expressed easily[2] in a context-free grammar.

A more complex example can be found within the JavaScript grammar, that will be used later. Within Figure 2.10, a module is a list of declarations and statements. A declaration can be a function declaration (other options are omitted within the figure). A statement can be an expression, followed by a semicolon or a line break. This expression can, among other things, be a function expression. For the input `"function foo() {}"`, the parser cannot decide whether it is a function expression within a statement, or a function declaration. If there were a semicolon in the end, it would be a statement, as only statements can end with a semicolon. However, the semicolon at the end of a statement can be omitted if there is a line break.

This ambiguity was likely introduced when JavaScript was extended for functional programming: This paradigm requires that functions can be used as values, e.g. as arguments to other functions, and therefore it requires a function expression like the one in Figure 2.10. It would be possible to avoid ambiguity if a ⟨*FunctionExpression*⟩ where prohibited within a `"Statement"`, however, the necessary grammar rewrite would increase the complexity of the grammar. Also, this issues does not present itself as a problem for an implementation of JavaScript, as the semantics of ⟨*FunctionExpression*⟩ and ⟨*FunctionDeclaration*⟩

[2]The identifier is expressed as a regular expression, and the rule "every word except for `"false"`" can be expressed as a regular expression. Then, regular expressions are closed under intersection, so "every identifier except for `"false"`" can be expressed as a regular expression. However, the resulting expression is too complex to be written — or read — by a human grammar author.

For reference

Definition 0: Predicate of Interest
The *predicate of interest* is a predicate over observable program behavior. The program behavior which is recognized by the predicate of interest is the *behavior of interest*. If the program exhibits the behavior of interest while processing a specific input, this input is said to be *behavior-triggering*.

See Section 1.4

are identical [3]. Still, it leads to an ambiguity within the grammar.

ALHAZEN0 is supposed to be general, and work with every context-free grammar, which means that additional rules, like preventing specific words despite the fact that they are allowed by the grammar, are not an option. Also, ALHAZEN0 does not know about the semantics of an input. Therefore, I have to design all algorithms I am going to use in the thesis such that they can handle ambiguity.

This also means that it is simpler to write grammars to be used with ALHAZEN0. There is no need to spend the time to make them unambiguous. Within the thesis, I was able to use grammars that where written for fuzzing campaigns. Those were written for generating, not for parsing. Therefore, their authors did not care about ambiguity, but still ALHAZEN0, in contrast to most parsers, consumes those grammars without further ado.

2.4 Increasing the Probability of Behavior-Triggering Samples

In the context of ALHAZEN0, I require input data sets for two reasons. First, I need input samples to learn from. Second, I need test data to evaluate whether ALHAZEN0 recognizes behavior-triggering inputs. Generating those data sets is no easy task: In most use cases, the behavior of interest is seldom, that is, triggered by a low number of inputs, while the vast majority of inputs trigger other behaviors. This means that if I were to sample the input space randomly, I'd get only very few input samples which trigger the behavior, and lots of samples which do not.

For ALHAZEN0, I need sample sets which are more likely to contain the behavior-triggering inputs. I therefore modify the grammar to increase the probability to generate such inputs.

Concretely, I look at every node in the parse tree which corresponds to a non-terminal, and I add the leaf word of the subtree rooted at this node to the grammar, if it does not exist already, and the same leaf word was not added to the grammar before.

Within the running example, the grammar in Figure 2.1 and the input `"sqrt(9)"`, I would add `"9"` as an alternative to ⟨*number*⟩ and `"sqrt(9)"` as an alternative to ⟨*start*⟩. I would consider to add `"sqrt"` as an alternative to ⟨*function*⟩, but abstain from doing so, as it is already present. The enriched grammar is shown in Figure 2.11.

[3] The function value is dropped when the expression is in an expression statement and, somewhat counterintuitive, the ⟨*FunctionExpression*⟩ also adds a definition to the current name space.

⟨*start*⟩ → ⟨*function*⟩ `"("` ⟨*number*⟩ `")"` | `"sqrt(9)"`
⟨*function*⟩ → `"tan"` | `"cos"` | `"sin"` | `"sqrt"`
⟨*number*⟩ → `/-?[0-9]+(.[0-9]+)?/` | `"9"`

Figure 2.11: The example grammar, enriched for the input string `"sqrt(9)"`.

There are three possible parse trees for `"sqrt(9)"` now. In addition to the original parse tree in Figure 2.3, there is a parse tree which uses the alternative `"9"`, rather than the regex. And a third one, which derives the entire string at the root symbol.

This is a general property of this rewrite: Adding existing grammar words as alternatives means that there are at least two parse trees for those words now. The grammar is ambiguous. However, as explained before, ALHAZEN0 needs to be able to deal with ambiguous grammar anyways, so this does not cause additional problems.

2.5 Evaluation Setup

The evaluation in this chapter is supposed to assess the *generative* and *predictive* power of ALHAZEN0. In Section 1.3, I showed how ALHAZEN0 works on two inputs. However, ALHAZEN0 is a machine-learning approach, so it stands to reason that more input data leads to better results. Therefore I will evaluate how ALHAZEN0 behaves with

1. just two inputs, as in the example (referred to as the twoInputs configuration),
2. a larger set of inputs, randomly sampled from the grammar (referred to as the sets configuration),
3. a larger set of inputs, based on k-path[31] (referred to as the k-path configuration)

.

2.5.1 Evaluation Metrics

I will evaluate ALHAZEN0 based on *precision* and *accuracy*. For each input, ALHAZEN0 predicts whether it triggers the behavior of interest or not.

This leads to four different outcomes:

True Positive ALHAZEN0 predicts that the input *triggers* the behavior, and it actually *does*.

True Negative ALHAZEN0 predicts that the input *does not* trigger the behavior, and it actually *does not*.

False Positive ALHAZEN0 predicts that the input *triggers* the behavior, but it actually *does not*.

False Negative ALHAZEN0 predicts that the input *does not* trigger the behavior, but it actually *does*.

Obviously, true negative and true positive, the cases where ALHAZEN0 is right, are most desirable. Using those four categories, I can define precision and accurracy.

Definition 10: Precision

$$\text{precision} = \frac{\#\text{true positive}}{\#\text{true positive} + \#\text{false positive}}$$

That is the fraction of correctly classified inputs among all inputs classified as behavior-triggering.

Precision rates how often a classifier correctly recognizes behavior-triggering inputs. Consider the use cases where we want to protect a program against inputs that cause a crash, e.g. to prevent denial-of-service attacks. In this case, high precision would render those attacks impossible. However, precision does not consider the number of true negatives. Within the examplatory use cases, legitimate users may be denied service, as their inputs are wrongly considered harmful.

Definition 11: Accuracy

$$\text{accuracy} = \frac{\#\text{true positive} + \#\text{true negative}}{\#\text{all inputs}}$$

That is the fraction of correctly classified inputs among all inputs.

Accuracy rates how often a classifier correctly classifies all inputs. Accuracy does not suffer from the problem of ignoring true negatives, but it may be thrown off by uneven data distribution: If the number of inputs which do not trigger the behavior is much higher than the number of inputs which do, a classifier with low precision, that is, a classifier which misses many cases, would likely have high accuracy, as it gets the much larger class of non-behavior-triggering inputs right.

In the example of protecting against harmful inputs, high accuracy means that legitimate users will not be affected. However, if high accuracy is paired with low precision, many harmful inputs reach the system, as the number of true negatives is high, but the number of true positives is low.

Within the evaluation, I need to make sure that the number of behavior-triggering and not behavior-triggering inputs is almost identical, as the interpretation of accuracy values would be difficult otherwise.

2.5.2 Subjects

For evaluation, I need a number of subjects to apply ALHAZEN0 to. A subject consists of:

1. a program under test,
2. a grammar for the input language of this program,

3. at least one behavior-triggering input,

4. a predicate of interest,

which recognizes the program behavior to generate an explanation for.

In this section, I describe the subjects I am using in the evaluation. Table 5.9 lists all subjects, with information about the grammar and predicate of interest I used.

`grep` and `find`

`grep` and `find` are software utilities that can be found on most UNIX systems. `grep` reads in a text file, applies a regular expression to each line and outputs all matching lines. `find` searches the file system for a file which matches a given query. For my experiments, I used 5 versions of `grep` and 4 versions fo `find` from the DBGBENCH[7] study. Those `grep` and `find` versions contain known bugs. DBGBENCH also provides bug reports for those bugs, including inputs which trigger the problem. I used the inputs from the bug reports as starting points for ALHAZEN0. The grammars for `grep` and `find` were written by myself.

- For `grep`, the grammar generates a full shell command, consisting of an input, a list of environment variables and an invocation of `grep`. The input is an alphanumeric string, which may contain utf-8 multibyte characters. The grammar allows for all environment variables that are documented in the man page of `grep` for the oldest version we used. The grammar allows for all command line flags that are documented in the man page of `grep` for the oldest version we used.

- The `find` grammar also generates a full shell command, and allows for environment variables and command line flags. In addition, the `find` grammar generates a sequence of `mkdir`, `touch` and `ln` shell commands to generate directories, files and symbolic links.

I used some of the General-purpose Predicates of Interest I introduced in Section 1.4.1, and two subject-specific oracles:

grep.c96b0f2c, grep.7aa698d3 `grep` outputs lines of the input which match the given predicate. So every line in the output of `grep` should also occur in the input. If `grep` outputs character sequences which were not present in the input, this is a bug. When implementing this predicate, care has to be taken to handle some options for `grep`. E.g. the option `-n` configures `grep` to output line numbers, which need to be ignored when comparing the output to the input.

find.24bf33c0 `find` outputs file names of files which match the search query. `find` should never output a line which is not a (valid) file path, or is a file path to a non-existant file. Both properties can easily be checked for each output line. When implementing this, care has to be taken, because `find` may output error messages, which are not valid file path, but should not trigger the predicate either.

For reference

General-purpose Predicates of Interest
The following predicates of interest are reusable with different programs:

Crash Oracle Some error conditions, e.g. memory protection faults, are detected by the operating system, which, in turn, terminates the program. Crash Oracles consider this forceful termination as behavior of interest.

Timeout Oracle For many programs, a resonable maximal execution time can be defined. Timeout Oracles classify executions which exceed this maximum as buggy.

Error-message Oracle Many programs output detailed error messages. In this case, the presence (or absence) of a specific error message in the program output can be used as an oracle. This is especially useful for Java Programs, as the Java Runtime offers mechanisms for error checking, and outputs detailed stack traces if one of those checks fails. This allows a predicate of interest to check specifically for the line where an error occurs.

Regression Oracle As programs are being developed, new problems may be introduced in old code. A regression oracle checks whether the output of a program is the same as in an old version.

See Section 1.4.1

JavaScript

I also used three subjects which process program code written in JavaScript. JavaScript is a programming language that is widely used for web, full-stack and desktop applications. I used the JavaScript grammar that was used in Havrikov and Zeller [31]. This grammar was a manual translation of the JavaScript specification, done by one of the authors of Havrikov and Zeller [31]. I fixed several translation errors or imprecisions while working with the grammar. Most of those problems can be explained by the fact that Havrikov and Zeller [31] used the grammar for generating, while I also used it for parsing. A generator grammar can make due if all words it describes are in the language, whereas a parse grammar needs to describe the language as precisely as possible.

Rhino and JerryScript are implementations of a JavaScript interpreter, written in Java and C++ respectively. Rhino is developed at Mozilla, and was part of the Java Development Kit from version 6 to 11. JerryScript is a lightweight JavaScript engine that is optimized for usage in embedded systems. Closure was developed at Google. It transpiles JavaScript into optimized JavaScript. It is written in Java as well.

For Rhino and Closure, I targeted bugs that were discovered by the fuzzing campaign in Havrikov and Zeller [31]. The predicates of interest are based on recognizing the specific error message triggered by the bug.

For JerryScript, I targeted bugs that I found on the bug tracker. The predicates of interest are again based on the error message.

For reference

Definition 2: Parse Trees and Words in the Language
A *parse tree* for a context-free grammar (N, T, P, S) is a tree where each node is labeled with a control form. The root node is labeled with a control form $\langle s \rangle$ such that $S \rightarrow \langle s \rangle$ is in P. If a node is labeled with

1. a reference $\langle A \rangle$, it has one child which is labeled with a control form $\langle C \rangle$, such that $\langle A \rangle \rightarrow \langle C \rangle$ is a production rule in P.
2. a terminal symbol, it has no children (it is a leaf node).
3. an alternation $\langle C_1 \rangle$ | ...| $\langle C_n \rangle$, it has one child, labeled with one of the $\langle C_i \rangle$'s.
4. a concatenation $\langle C_1 \rangle$ ...$\langle C_n \rangle$, it has n children, where the i-th child is labeled with $\langle C_i \rangle$.
5. a quantification, all its children are labeled with the subject of the quantification. The node has one or more children if the annotation is +, zero or more children if the annotation is $*$ and zero or one children if the annotation is ?.

The sequence of terminal symbols that consists of the labels of the leaves labeled with a terminal symbol of the parse tree in preorder is the *leaf word* of this tree. It can also be said that the parse tree *derives* its leaf word. All words which have valid parse trees, that is, trees formed according to the rules above, are *words of the language*.

See Section 2.1

Genson

Genson is a Java Library for handling JSON input. JSON is file format that is based on storing key-value maps, where each value can be a primitive value, an array of values or another key-value map. JSON is widely used as a serialization format for data structures.

NetHack

NetHack[45] is an adventure game which was first published in 1987, and which, despite ASCII graphics and horribly complex gameplay, still receives attention (and software maintenance) from enthusiasts. This modernized version of NetHack had a software bug when reading its configuration file. This bug serves as a subject for Alhazen0. I used a generic grammar for property files, which I wrote myself.

2.5.3 Generating Test Data

In the evaluation, test data sets are used to see whether Alhazen0 recognizes behavior-triggering inputs. I generated this test data with an approach similar to Pavese et al. [55], using the enriched grammar as described in Section 2.4. This technique uses a probabilistic generator to generate new input samples.

Bug	Grammar Source	Predicate of Interest
grep c96b0f2c	hand written	subject-specific oracle
grep 55cf7b6a	hand written	Regression Oracle
grep 7aa698d3	hand written	subject-specific oracle
grep 2be0c659	hand written	Regression Oracle
grep 3c3bdace	hand written	Crash Oracle
grep 3220317a	hand written	Crash Oracle
grep c1cb19fe	hand written	Regression Oracle
grep 5fa8c7c9	hand written	Timeout Oracle
find 07b941b1	hand written	Crash Oracle
find 24bf33c0	hand written	subject-specific oracle
find 091557f6	hand written	error-message oracle
find b445af98	hand written	Regression Oracle
find dbcb10e9	hand written	Crash Oracle
find e1d0a991	hand written	Regression Oracle
find ff248a20	hand written	Timeout Oracle
closure 1978	[31], with adaptations	error-message oracle
closure 2808	[31], with adaptations	error-message oracle
closure 2842	[31], with adaptations	error-message oracle
closure 2937	[31], with adaptations	error-message oracle
closure 3178	[31], with adaptations	error-message oracle
closure 3379	[31], with adaptations	error-message oracle
rhino 385	[31], with adaptations	error-message oracle
rhino 386	[31], with adaptations	error-message oracle
genson 120	[31]	error-message oracle
jerryscript 3286	[31], with adaptations	error-message oracle
jerryscript 426	[31], with adaptations	error-message oracle
jerryscript 3297	[31], with adaptations	error-message oracle
nethack 5214	hand-written	error-message oracle

Table 2.1: All subjects used in the evaluation of ALHAZEN0.

The base algorithm for probabilistic generation from a grammar is shown in Listing 2.3. It is identical to the algorithm used in Pavese et al. [55][4]. However, Pavese et al. [55] use more than 2 seed inputs, which is why there probability distribution did not work for my use case. Therefore, I define a slightly different probability distribution at the end of this section.

The algorithm generates the nodes of a parse tree in preorder. This means that it can always use the definition of a parse tree and the grammar to determine what kind of node, and how many nodes are required below a given node. This is used to generate the appropriate child nodes for references and concatenations. For alternations and quantifications, the definition allows for different labels in child nodes, or number of child nodes respectively. For alternatives, the given probability distribution will be used to decide on the label of the child node

[4] I actually used the same source code.

Listing 2.3: Probabilistic algorithm for generating a parse tree for a given grammar, as presented in Pavese et al. [55].

```
1  // Input: A control form ⟨c⟩, a probability distribution P, a partial
       parse tree t and a depth cut-off value cut_off
2  Probabilistic-Generate(⟨c⟩, P, t):
3        Create a new node n in t and label it with ⟨c⟩
4        let current_depth be the depth of n in t
5        if current_depth >= cut_off:
6              remove n from t
7              return shortest-generate(⟨c⟩, t)
8        if ⟨c⟩ is a Reference:
9              find a production rule ⟨c⟩ → ⟨b⟩
10             add Probabilistic-Generate(⟨b⟩, P, t) as a child to n
11       if ⟨c⟩ is an alternation ⟨c1⟩ | …| ⟨cn⟩:
12             randomly select a ⟨cj⟩ with probability distribution P
13             add Probabilistic-Generate(⟨cj⟩, P, t) as a child to n
14       if ⟨c⟩ is a concatenation ⟨c1⟩ …⟨cn⟩:
15             for all ⟨cj⟩:
16                   add Probabilistic-Generate(⟨cj⟩, P, t) as a child to n
17       if ⟨c⟩ is a Quantification ⟨c′⟩*:
18             choose an integer v larger than 0 at random
19             while n has less than v children:
20                   add Probabilistic-Generate(⟨c′⟩, P, t) as a child to n
21       if ⟨c⟩ is a Quantification ⟨c′⟩+:
22             choose an integer v larger than 1 at random
23             while n has less than v children:
24                   add Probabilistic-Generate(⟨c′⟩, P, t) as a child to n
25       if ⟨c⟩ is a Quantification ⟨c′⟩?:
26             choose an integer v ∈ [0, 1] at random
27             while n has less than v children:
28                   add Probabilistic-Generate(⟨c′⟩, P, t) as a child to n
29       return n
```

(Line 11). For quantifications, the random decision how many children to create is not influenced by the probability distribution (Lines 17, 21 and 25). It was done like this in the source code of Pavese et al. [55], even if I cannot tell why. The algorithm uses a cut-off if the tree is too deep: It falls back to Shortest-Generate, as presented in Listing 2.2, if the tree becomes deeper than the depth limit.

Please mind that P is not the distribution of the leaf words in all generated parse trees. To see this, consider the grammar shown in Figure 2.12. This grammar has 4 leaf words: `"a1"`, `"a2"`, `"b1"` and `"b2"`. Assume I am using a probability distribution which has $P(\langle A1\rangle) = P(\langle A2\rangle) = P(\langle B1\rangle) = P(\langle B2\rangle) = 0.5$. Now, one would assume that each leaf word occurs with equal probability. However, the probability of generating `"a1"` is in fact $P(\langle A\rangle)P(\langle A1\rangle)$, so if $P(\langle A\rangle) \neq P(\langle B\rangle)$, leaf words would occur with different probabilities[5].

The probability distribution which was used in Pavese et al. [55] does not work well if there is just two seed inputs to learn from. They just count the occurrence of control forms, however, with just one sample to learn from, all

[5] And hence $P(\langle A1\rangle) + P(\langle A2\rangle) + P(\langle B1\rangle) + P(\langle B2\rangle) \neq 1$ is not a mistake, but necessary.

⟨*start*⟩	→	⟨*A*⟩ | ⟨*B*⟩
⟨*A*⟩	→	⟨*A1*⟩ | ⟨*A2*⟩
⟨*B*⟩	→	⟨*B1*⟩ | ⟨*B2*⟩
⟨*A1*⟩	→	`"a1"`
⟨*A2*⟩	→	`"a2"`
⟨*B1*⟩	→	`"b1"`
⟨*B2*⟩	→	`"b2"`

Figure 2.12: A grammar which illustrates why P is not the distribution of grammar words.

those counts are either 0 or 1, but then, all probabilities are either 0 or 1 just as well, a problem that I counteract by smoothing the probability distribution. My probability distribution is defined as follows:

I took the occurrence count of each control form in the seed inputs (see Section 2.5.2) and applied a standard Laplace Smoothing[63]:

Definition 12: Occurrence Count and Smoothed Occurence Count
For a control form $\langle C \rangle$ and a parse tree T, let the *occurence count* $\#_T(\langle C \rangle)$ be the number of nodes labeled with $\langle C \rangle$ in T. Then, I define the *smoothed occurence count*

$$s_T(\langle C \rangle) = \#_T(\langle C \rangle) + 1$$

.

For a control-form $\langle c \rangle$ with a production rule $\langle c \rangle \rightarrow \langle c_1 \rangle \mid ... \mid \langle c_n \rangle$, I define the probability as

$$P(\langle c_i \rangle) = \frac{s_T(\langle c_i \rangle)}{\sum_{i=0}^{n} s_T(\langle c_i \rangle)}$$

This leaves P undefined for all control-forms which are not part of the sequence of an alternation, however, the algorithm does not use those probabilities anyways.

Input Sets

Using the pre-existing behavior-triggering samples as seed inputs, I used the approach described in Section 2.5.3 to generate large input sets. I filtered those sets for duplicates, to make sure that I get different input samples. Then, I ran the approach several times, until I had 1000 unique, behavior-triggering inputs or a timeout of 1 hour was exhausted.

The inputs within those sets are similar, but not identical to the pre-existing inputs, so I have a high likelihood to generate behavior-triggering inputs. However, the proportion of behavior-triggering and non-behavior-triggering inputs is still not balanced.

Table 2.2 gives the number of bug-triggering and non bug-triggering samples for each subject.

subject	all samples		training samples		fraction
	benign	bug-triggering	benign	bug-triggering	
calculator	3809	1245	311	311	0.25
NetHack	2992	1155	289	289	0.28
closure 1978	3140	1128	282	282	0.26
closure 2808	4043	1157	289	289	0.22
closure 2842	9380	242	60	60	0.025
closure 2937	4058	1147	287	287	0.22
closure 3178	3078	1048	262	262	0.25
closure 3379	3944	1218	304	304	0.23
rhino 385	6077	1100	275	275	0.15
rhino 386	3996	1137	284	284	0.22
genson 120	6940	1174	293	293	0.14
find 07b941b1	3162	922	230	230	0.22
find 091557f6	3206	855	214	214	0.21
find dbcb10e9	3066	995	249	249	0.24
find ff248a20	3351	709	177	177	0.17
grep 3220317a	2862	1219	305	305	0.30
grep 3c3bdace	2787	1298	324	324	0.32
grep 5fa8c7c9	1665	363	91	91	0.18
grep 7aa698d3	3640	433	108	108	0.11
grep c96b0f2c	5625	507	127	127	0.08
jerryscript 3286	4094	1075	269	269	0.21
jerryscript 426	4316	813	203	203	0.16
jerryscript 3297	4176	6	1	1	0.00
jerryscript 3267	4050	1109	277	277	0.21
jerryscript 3276	5021	182	45	45	0.035
jerryscript 3389	4309	873	218	218	0.19

Table 2.2: Size of the sample sets.

It is clearly visible that some bugs are harder to trigger then others. For Jerryscript 3297, we have just 6 bug-triggering input samples. The initial, bug-triggering input for this is:

```
1    "98765".replace(76, function ( ) { return $ })
```

The important aspect for this bug is that the first argument to `replace`, in the sample `76`, is a substring of the string `replace` is invoked on (in the sample `98765`). Probabilities, as used with this generator, cannot model that. Therefore, getting an input which triggers this bug is pure luck.

Other bugs are simpler to trigger: For `grep 3c3bdace`, I generated 1298 bug-triggering samples. This bug is a segmentation fault when providing a specific regex to `grep`. This specific regex is added as an alternative to the grammar by the transformation described in Section 2.4, and therefore it is generated over

and over again.

All in all, roughly a fifth of all generated samples are bug-triggering. Therefore, the data sets contain enough benign and bug-triggering samples to evaluate the ALHAZEN0 approach.

In fact, a closer examination of the data revealed a potential problem: Many of the bug-triggering samples contain the (sub-)strings of the original samples, which was added as an alternative to the grammar in Section 2.4. The reason for this is first, that those are contained within the parse trees, and therefore get a high probability within my distribution. Second, the algorithm uses the shortest derivation as a cut-off if trees get too deep. The newly added alternatives are terminals, and have a shortest derivation of 1. Therefore, they are chosen as cut-off in many cases. This means that many of the behavior triggering samples, while not identical to the original samples, contain fragments of the original samples. We will see whether this leads to a bias in the evaluation later on.

Splitting into Training and Verification Sets I now proceed to split the generated input sets into smaller sets. I used $\frac{1}{4}$ of the behavior-triggering inputs as training set, and the remaining $\frac{3}{4}$ of them as verification set. To get a balanced training set, I randomly selected benign samples, such that the training set has the same number of benign and bug-triggering samples.

Then, I build verification sets: Each of those sets is supposed to be as large as the training set and contain the same number of benign and behavior-triggering samples. I achieve this by splitting the remaining samples into sets, and filling each of those sets with randomly selected samples from other sets until they have the same size, and same number of benign and behavior-triggering samples. This means that a sample may be contained in more than one verification sets.

None of those sets, and none of the contained samples, was used during the training phase of ALHAZEN0.

2.5.4 Generating Training Data

Within the evaluation, I use three different methods to generate training data:

The Sets Configuration In this configuration, I use the training set which was extracted from the data generated in the previous section as training data. The training data sets are relatively large, providing the decision trees with lots of observations to learn from. However, there is a clear threat to validity: The verification sets were derived by the same process as the training data. I already described how those sets contain subsets of the original samples quite often. If this is true for both, training data and verification data, ALHAZEN0 may describe such an implicit property of the inputs, rather than the actual cause of the error. I'll have to carefully check whether the values obtained with those training sets are representative for ALHAZEN0's performance.

The TwoInputs Configuration In this configuration, I use the behavior-triggering input from the bug report (see Section 2.5.2) and a randomly generated benign sample as training data. This approach has just two inputs to learn from. In actual software development, it is uncommon to know more than one input

which triggers a given bug. Therefore, this is a realistic scenario. On the other hand, it may simply be not enough data for a machine learning approach.

The 1-Path Configuration In this configuration, I use a set of inputs which satisfy the k-path criterion[31] with $k = 1$, and the behavior-triggering input from the bug report. The 1-path approach generates a set of inputs such that each control-form $\langle c \rangle$ is used in at least one parse tree for any of the inputs in the set. The 1-path metric was designed to create inputs which contain each control form $\langle c \rangle$ at least once. This means that there is good coverage of the input model: The decision tree was trained with diverse inputs, covering the entire input space. The machine learning approach does not suffer from uncertainty over some part of the input space. However, in many cases those sets contain no more than one failing input. This means that there is not a lot of information about the bug.

2.6 Creating Hypothesis

ALHAZEN0 uses a context-free grammar to parse program inputs, and derive numerical *features* from the parse tree. Based on those features, it trains a *decision tree*. This decision tree provides a set of constraints, which describe the conditions under which the behavior of interest occurs. I use the transformed grammar from Section 2.4 within this process.

2.6.1 Feature Extraction

The second step within ALHAZEN0 is to extract numeric feature vectors from input samples. This extraction is based on parsing. Each input sample is a word in the language of the input grammar. I use a parser to obtain the parse tree for this word, and extract features from the parse tree. I have to account for the fact that the grammars are ambiguous, so I use an Earley Parser[15], which generates all possible parse trees for a word.

In feature extraction, I consider all control forms $\langle C \rangle$ such that

1. there is a production rule $\langle A \rangle \rightarrow \langle C \rangle$
2. or there is a production rule $\langle A \rangle \rightarrow \langle C \rangle \mid \langle B \rangle \mid \ldots$

Let $\langle C \rangle$ be such a control form, and let W be the set of all words derived from a node labelled with $\langle C \rangle$ in any parse tree for the given input. I use the following features:

Existence This feature is 1 iff W is non-empty, 0 otherwise. It is written as exists($\langle C \rangle$).

Maximum Length This feature is the length of the longest word in W. It is written as max-char-length($\langle C \rangle$). The value of this feature is 0 if W is empty.

Maximum Quantification Length If $\langle C \rangle$ is a quantification, I use the maximal number of children for any node labelled with $\langle Q \rangle$ in the parse tree. It is written as max-qu-length($\langle C \rangle$). The value of this feature is 0 if W is empty.

Feature	Value
exists(⟨*start*⟩)	1
exists(⟨*start*⟩ → `"sqrt(9)"`)	1
exists(⟨*function*⟩ → `"sqrt"`)	1
exists(⟨*function*⟩ → `"tan"`)	0
exists(⟨*function*⟩ → `"sin"`)	0
exists(⟨*function*⟩ → `"cos"`)	0
exists(⟨*number*⟩)	1
exists(⟨*number*⟩ → `"9"`)	1
max-char-length(⟨*start*⟩)	11
max-char-length(⟨*number*⟩)	4
max-numeric(⟨*number*⟩)	9
max-char(⟨*start*⟩)	116
max-char(⟨*number*⟩)	57

Table 2.3: All features for the example input `"sqrt(9)"` and the grammar in Figure 2.11.

Maximum Code Point I use the maximum integer code point within any word in W as a feature. It is written as max-char(⟨C⟩). The value of this feature is $-\infty$ if W is empty, or contains only the empty word.

Maximum Numeric Interpretation If all terminal symbols reachable from ⟨C⟩ consist of the symbols `"-"`, `"."` and digits only, I use the maximum numeric interpretation of any word in W as a feature. It is written as max-numeric(⟨C⟩). The value of this feature is $-\infty$ if W is empty, or no word in W has a numeric interpretation.

The features and their values for the example input `"sqrt(9)"` and the grammar in Figure 2.11 are listed in Table 2.3. The root of the parse tree is labeled ⟨*start*⟩, so this production rule is used, and the corresponding exists(⟨*start*⟩)-Feature is 1. The alternative `"sqrt(9)"`, added to the grammar within the grammar transformation described in Section 2.4, is 1 as well. Those two productions, ⟨*start*⟩ and ⟨*start*⟩ → `"sqrt(9)"` form a complete parse tree. This illustrates the need for an earley parser. If I were looking at this parse tree exclusively, all other features would be 0 or ∞ respectively. However, I look at all possible parse trees. Therefore, the other features will be derived from other trees. The feature exists(⟨*function*⟩ → `"sqrt"`) is 1, because there is a parse tree, the one in Figure 2.3, which contains a `"sqrt"`. Likewise, max-numeric(⟨*number*⟩) is set to 9, because there is one parse tree which contains a ⟨*number*⟩ node with a value of 9.

2.6.2 Feature Selection

Feature selection, that is, the decision which features to use, is an important step in Alhazen0. First of all, the selected features define the capabilities of Alhazen0. If the condition for the behavior of interest cannot be expressed in

For reference

Definition 4: Grammar Graph
The *Grammar Graph* for a grammar G is a directed graph. Its nodes are the control forms of the grammar G. Two nodes $\langle C_i \rangle$ and $\langle C_j \rangle$ are connected with an edge if

1. $\langle C_i \rangle$ is a reference, and there is a production rule $\langle C_i \rangle \rightarrow \langle C_j \rangle$; or
2. $\langle C_i \rangle$ is an alternation, and $\langle C_j \rangle$ is part of the sequence; or
3. $\langle C_i \rangle$ is a concatenation, and $\langle C_j \rangle$ is part of the sequence; or
4. $\langle C_i \rangle$ is a quantification, and $\langle C_j \rangle$ is its subject

See Section 2.2

For reference

Definition 7: Number of leaf words $\#_d(\langle C \rangle)$
$\#_d(\langle C \rangle)$ is the number of valid parse trees rooted in $\langle C \rangle$. It is called the *number of leaf words*.

See Section 2.2

terms of the selected features, ALHAZEN0 will not be able to diagnose the problem correctly.

Another problem are structural correlations. Within the running example, the calculator grammar in Figure 2.1, there are four possible values for ⟨*function*⟩: `"sqrt"`, `"sin"`, `"cos"` and `"tan"`. There is an existence feature for each of those alternatives. However, there is also the max-char-length(⟨*function*⟩) feature. Note that max-char-length(⟨*function*⟩)$== 4$ if and only if exists(`"sqrt"`). Therefore, the decision tree can, later on, generate either the constraint exists(`"sqrt"`) or max-char-length(⟨*function*⟩)$== 4$. Both constraints are correct with respect to all available observations. However, as a human reading those constraints, exists(`"sqrt"`) is far more informative. I call this *structural correlation*.

Definition 13: Structural Correlation
If, due to the shape of the grammar, the values of two features always correlate, I call this *structural correlation*.

There are other cases of structural correlation, e.g. a max-numeric() feature may correlate with a chosen alternative just as well. In the following, I will present some heuristics which I use to rule out some of those cases.

Disabling Non-Informative Features

Looking at the features described in the previous section, one might notice that some of them do not give additional information in some situations. As an example, consider the production rule for ⟨*Function*⟩ in Figure 2.4. There are four exists(): exists(`"sqrt"`), exists(`"tan"`), exists(`"sin"`) and exists(`"cos"`), and in fact, those four each correspond to one of the four possible leaf words. But then, max-char-length(⟨*Function*⟩) $== 4$ means that exists(`"sqrt"`) $== 1$, there is no other leaf word with a length of 4. A similar observation can be

made for max-char(). If max-char(⟨*Function*⟩) == `"t"`, either exists(`"sqrt"`) or exists(`"tan"`) is 1.

Such situations occur quite often for control forms with a finite number of leaf words $\#_d$. If $\#_d(\langle C \rangle)$ is finite, this indicates that there are only alternations, concatenations and single-member quantifications reachable from this control form. But then, a combination of exists() already describes the entire (sub-)tree, and therefore, additional features are superfluous. Therefore, I do not use max-char-length(), max-char(), max-numeric() or max-qu-length() features for control forms with finite $\#_d$.

For a production rule $\langle A \rangle \rightarrow \langle C_1 \rangle \mid ... \mid \langle C_n \rangle$, many features can be determined by a combination of existence features and features of the $\langle C_i \rangle$. E.g. max-char($\langle A \rangle$) == max-char($\langle C_3 \rangle$) if exists($\langle C_3 \rangle$), because $\langle C_3 \rangle$ is a child of $\langle A \rangle$ within the parse tree. But then, the feature max-char($\langle A \rangle$) correlates with max-char($\langle C_3 \rangle$). Therefore, I do not use max-char-length(), max-qu-length() or max-char() for alternations.

The reasoning for maximum numeric interpretations is slightly different. Due to the rewrite in Section 2.4, my grammars often contain situations like ⟨*Literal*⟩ → `"9" | /[1-9][0-9]*/`. In this situation, max-numeric(⟨*Literal*⟩) == 9 can be due to exists(`"9"`), or due to `"9"` as a derivation of the regex. If, in an alternative, all options derive digits or the decimal dot only, I introduce a max-numeric() feature for the alternation, but not for individual members.

This does not remove all cases of structural correlation. For example, ambiguities always lead to structural correlation because exists() features for all possible interpretations are present. However, the heuristics in this chapter remove a far share of structural correlations.

2.6.3 Decision Trees

I use a decision tree learner [71] to learn associations between program behavior and input features. I use the implementation that is supplied by the scikit-learn[57] python library.

I decided to use a decision tree learner, because

- A decision tree learner can handle non-parametric data with arbitrary scale. The complex relationships between features mean that it is unclear how the values for individual features are distributed. Therefore, I need to be able to handle non-parameteric data. At the same time, features are not scaled equally. exists() features are essentially categorical, with the control from being either present, or not present. max-char-length(), max-qu-length() or max-numeric() features are numerical, and their upper and lower bounds depend on the grammar. Models like e.g. a linear regression require all features to have similar scale, and are therefore not directly applicable to our data.
- Decision trees are easy to interpret by a human. Explainability is an important property if I want to be able to use the generated hypothesis as an aid in manual debugging.

Within every (two-class) classification problem, the two classes generate a natural split of the input data: All observed data D is the union of the sets D^+, which contains only samples for the first class, and D^-, which contains only

samples for the second class. Within ALHAZEN0 those are the sets which contain only behavior-triggering samples, and the set which contains only non-behavior-triggering samples.

A decision tree classifier basically operates in the following way: The tree selects a feature f_0 and a boundary value v_0. The plane $f_0 < v_0$ also splits the set of all observed samples into two sets. The decision tree learner tries to chose a feature f_0 and a boundary value v_0 such that the plane $f_0 < v_0$ separates D^+ and D^- as good as possible. Then, the constraint $f_0 < v_0$ is used for the first node.

Afterwards, the decision tree attempts to split the two sets generated by the constraint recursively, until a predefined depth limit is reached, or the resulting sets contain samples with one label only.

Let's look at an example.

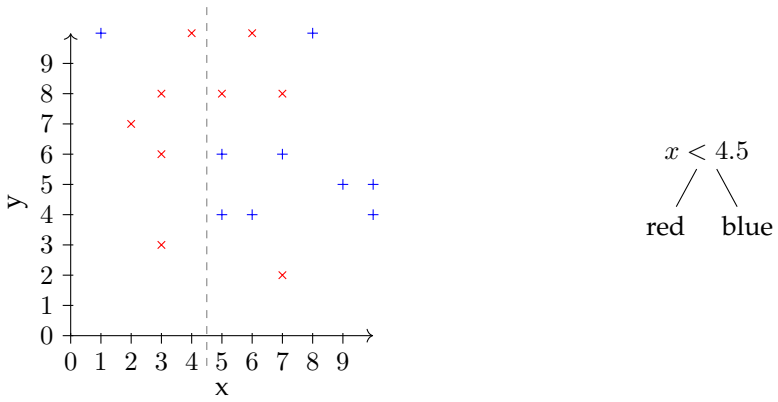


Figure 2.13: Step 1 of an example for the decision tree algorithm, using the x and y coordinates of a set of points as feature.

Figure 2.13 shows a cloud of random points. The decision tree learner already performed its first step. It found that $x < 4.5$ is a good splitting plane: There are 5 $\times$ and just one $+$ on the left, and 7 $+$ with 4 $\times$ on the right [6]. The corresponding tree is shown on the right of Figure 2.13. The learner proceeds to split both sets a second time.

[6] In fact, the algorithm to find a splitting plane is more complicated. Please look at Swain and Hauska [71] for the details.

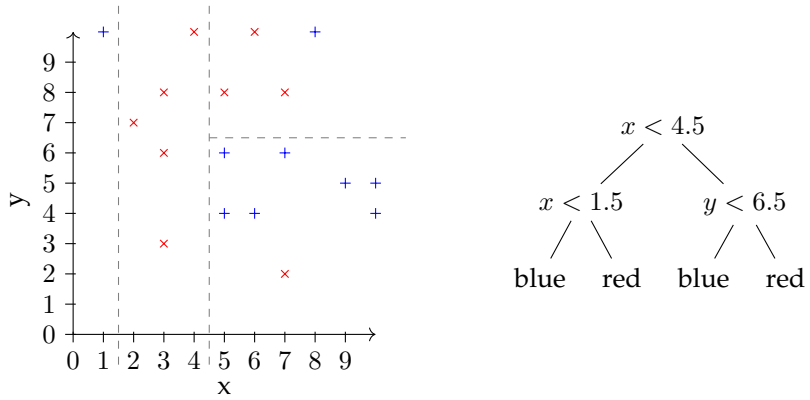


Figure 2.14: Step 2 of an example for the decision tree algorithm, using the x and y coordinates of a set of points as feature.

It decides to split the left compartment along the x-axis again, such that all subsets contain just one kind of points now. On the right-hand side, it split along the y-axis, which is the best possible split right now. Figure 2.14 shows the resulting splitting planes as dashed lines, and the resulting tree on the right. The algorithm introduced new internal nodes on both branches of the previous tree. In the next step, shown in Figure 2.15, the tree splits the two sets on the right again.

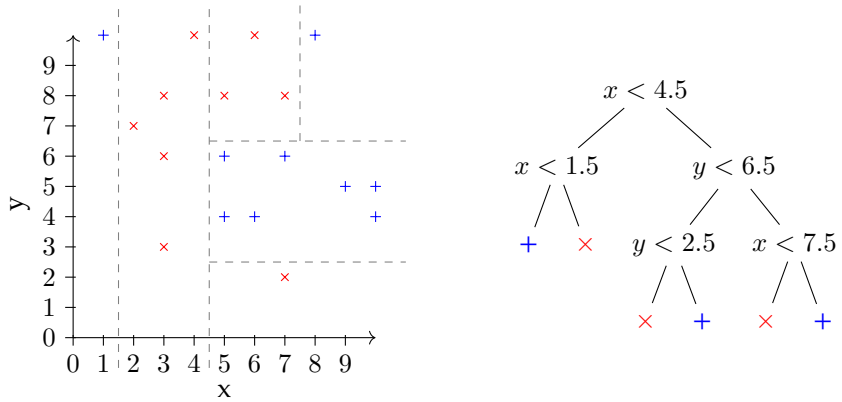


Figure 2.15: Step 3 of an example for the decision tree algorithm, using the x and y coordinates of a set of points as feature.

It decides to split along the x-axis for the set on top, and for the y-axis for the lower set. This means that two new nodes are created in the decision tree. All sets contain points from just one class now, so the algorithm terminates.

2.7 Evaluation

The evaluation follows the schema laid out in Section 2.5. I will look at the performance of Alhazen0as a predictor, and as a generator. After these quantitative

analysis, I will look at some of the generated trees qualitatively.

2.7.1 As Predictor

In this section, I evaluate whether ALHAZEN0 can be used to predict whether an input will trigger the behavior of interest. In order to do so, I ran ALHAZEN0 on all three training sets (see Section 2.5.3) and used the verification sets (see Section 2.5.3) to get precision and accuracy (see Section 2.5.1).

The results for accuracy can be seen in Figure 2.16, and Figure 2.17 gives the achieved precision. In both figures, the accumulated accuracy or precision respectively over all subjects is given in the top-most box.

From the accumulated results on top of the figures, you can see that ALHAZEN0 achieved its best accuracy (97.8%) and precision (96.5%) in the sets configuration. As described before, training data and verification data for this configuration are quite similar, so those results may not be reliable. However, the results for the twoInputs configuration (75.6% accuracy and 69.5% precision) and k-path configuration (85.5% accuracy, 81.9% precision) are remarkable as well. Especially for twoInputs, the configuration where ALHAZEN0 learns from just two input samples, the achieved results are stunning.

Many individual results are in the 9x range, and two subjects even achieve precison and accuracy of 100%. Looking at individual subjects, the worst precision is 58.9% for rhino 385 in the twoInputs configuration. The worst accuracy is 61.1%, for CLOSURE 2808 in the twoInputs configuration. It is not surprising to see both worse results in the two inputs configuration, as this configuration has the lowest number of samples to learn from.

RHINO 385 requires two function parameters with the same name. A property which cannot be described by ALHAZEN0's features. Therefore, it is not surprising to see the lowest precision here. CLOSURE 2808 requires a label to be defined, and used later. This is a context-sensitive property, which, again, cannot be expressed by ALHAZEN0, or the used grammar.

> ALHAZEN0 achieves an accuracy of 84.9% and a precision of 82.0% on average for all subjects, using the k-path training sets.

I used a Wilcoxon paired-samples hypothesis test to check whether the differences between the configurations are significant with a significance of 0.05. The test indicates that the sets configuration achieves statistically significantly different results from the k-path configuration, which, in turn, achieves different results than the twoInputs configuration. I can therefore claim that sets indeed achieves better results than k-path, and k-path achieves better results than two inputs.

Those results are good news with respect to the first potential use case of ALHAZEN0. Within this use case, I attempt to protect programs against malicious inputs. The word malicious is misleading here: While cyber attackers may use behavior-triggering inputs to trigger behavior that is beneficial for them, and harmful for others, a behavior triggering input may also be send to the program due to human error, or simply because it is not known that it would trigger misbehavior. ALHAZEN0's models serve as a filter in front of the program under test, and are supposed to recognize inputs that trigger ill-behavior, before any actual damage occurs. A precision of 95% means that this would be successful:

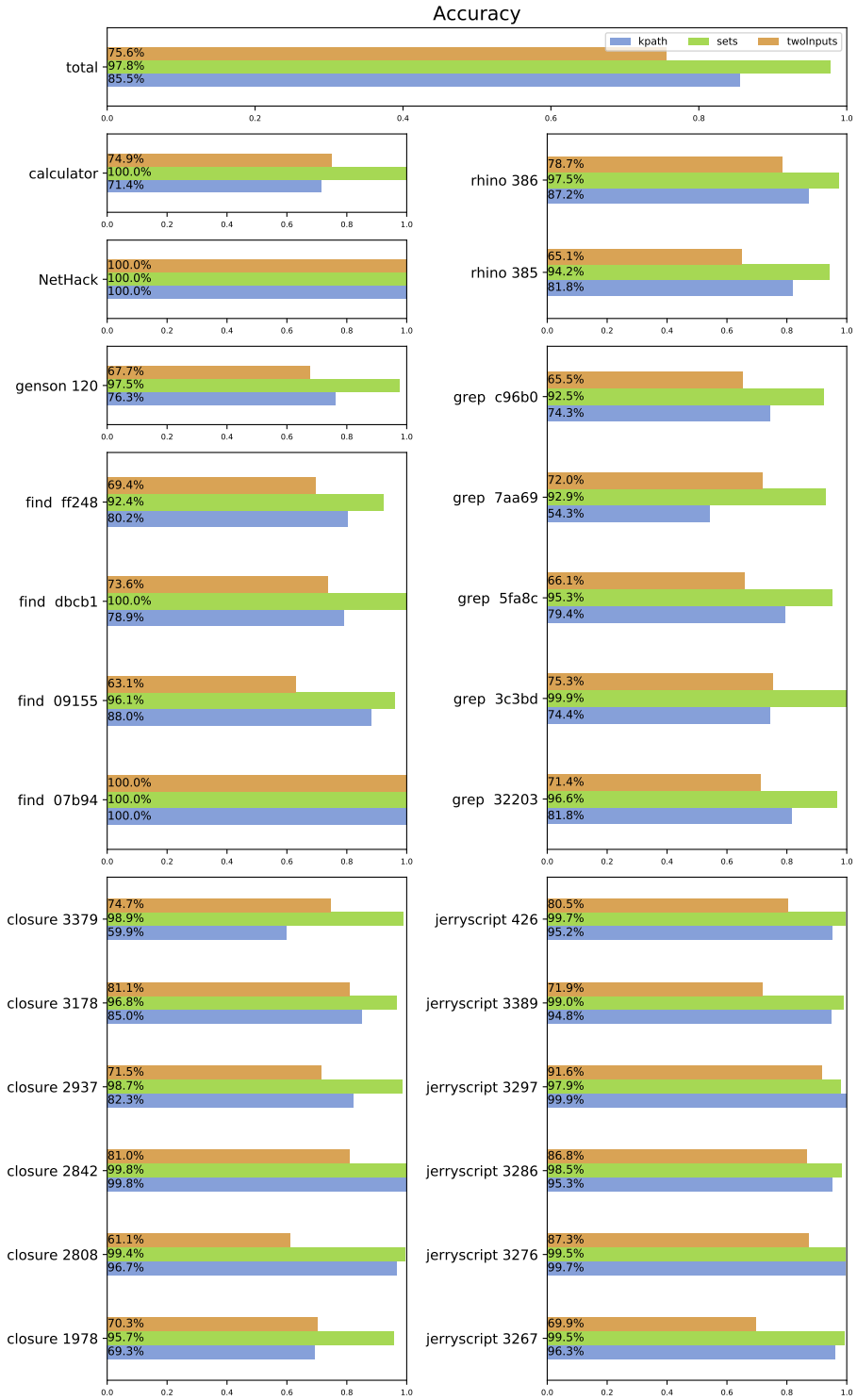


Figure 2.16: Accuracy for ALHAZEN0 as a predictor, on all subjects and all configurations.

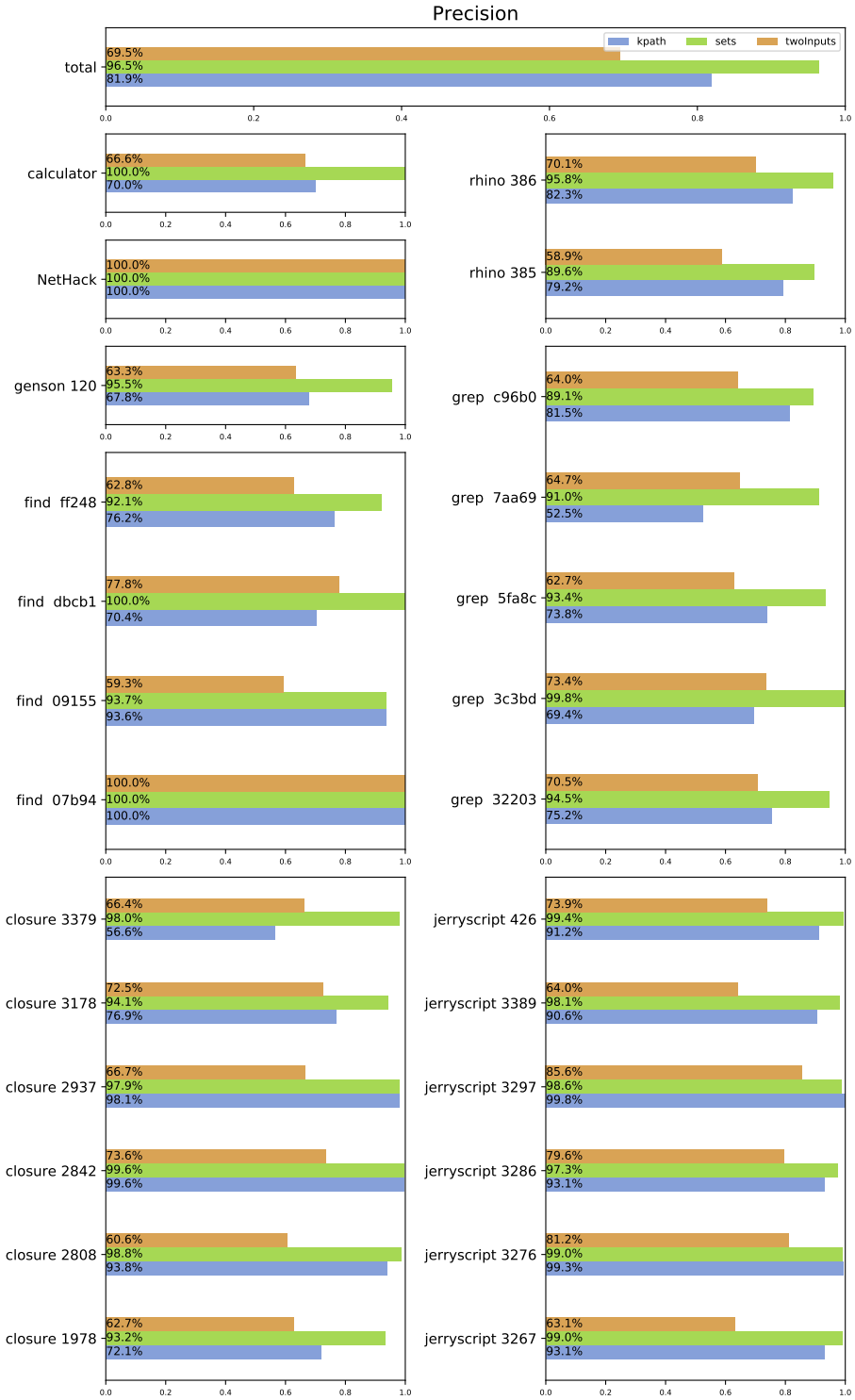


Figure 2.17: Precision for ALHAZEN0 as a predictor, on all subjects and all configurations.

A large number of behavior-triggering inputs can be recognized and prevented. At the same time, the accuracy of 96.6% means that only few legitimate requests will be denied.

Training Set Size At last, let's have a look at the size of the training sets. Figure 2.18 gives those numbers. As with accuracy and precision, those numbers are averages over 4 runs, which explains why the number of training samples can be a fractional.

The numbers for twoInputs are not surprising: This configuration supplied two input samples for each subject, so the number of training samples is always 2. However, it is surprising to see that the sets training sets are on average smaller than the k-path training sets. After all, one naively assumes that machine learning approaches work better with larger input sets, and the sets configuration has better precision and accuracy than the k-path configuration. I can therefore assume that the inputs within the sets configuration are more informative than the inputs generated by k-path.

> Carefully choosing informative inputs is more important than the absolute number of inputs.

2.7.2 As Generator

In this section, I evaluate whether Alhazen0 can be used to generate more inputs which trigger the behavior in question. So far, I did not describe how to create new samples from the decision tree. The required algorithm will be introduced in Chapter 3, as it is a main aspect of the work presented there. In this evaluation, I will just assume that I have the ability to do so.

Concretely, I assessed accuracy and precision of Alhazen0 as a generator in the following way: I extracted the path constraints for each path in the decision tree (see Section 3.1). Then I generated inputs for each of those paths/predicate sets (see Section 3.2). After that, I evaluated precision and accuracy for the set of inputs that were generated in this process.

Low precision is associated with a high false positive rate, meaning that Alhazen0 often generates behavior-triggering samples when it was asked to generate non-triggering inputs.

On the other hand, high accuracy indicates a low number of wrong predictions in general, or, within this part of the evaluation, indicates that Alhazen0 often generates inputs which are in the requested class.

Figure 2.19 gives the accuracy for this evaluation. Precision can be found in Figure 2.20.

Again, the topmost box in both figures gives accumulated data over all subjects. The best performing configuration is again the sets configuration. It achieves an accuracy of 79.8%, and a precision of 46.5%. The huge gap between precision and accuracy, however, gives reasons for despair. It indicates that while Alhazen0 generates quite some non-behavior triggering samples, it cannot generate behavior triggering samples reliably. This is confirmed by the data for individual subjects. Precision is below 50% for 18 of the 26 subjects. Two subjects could not even generate a single behavior-triggering samples.

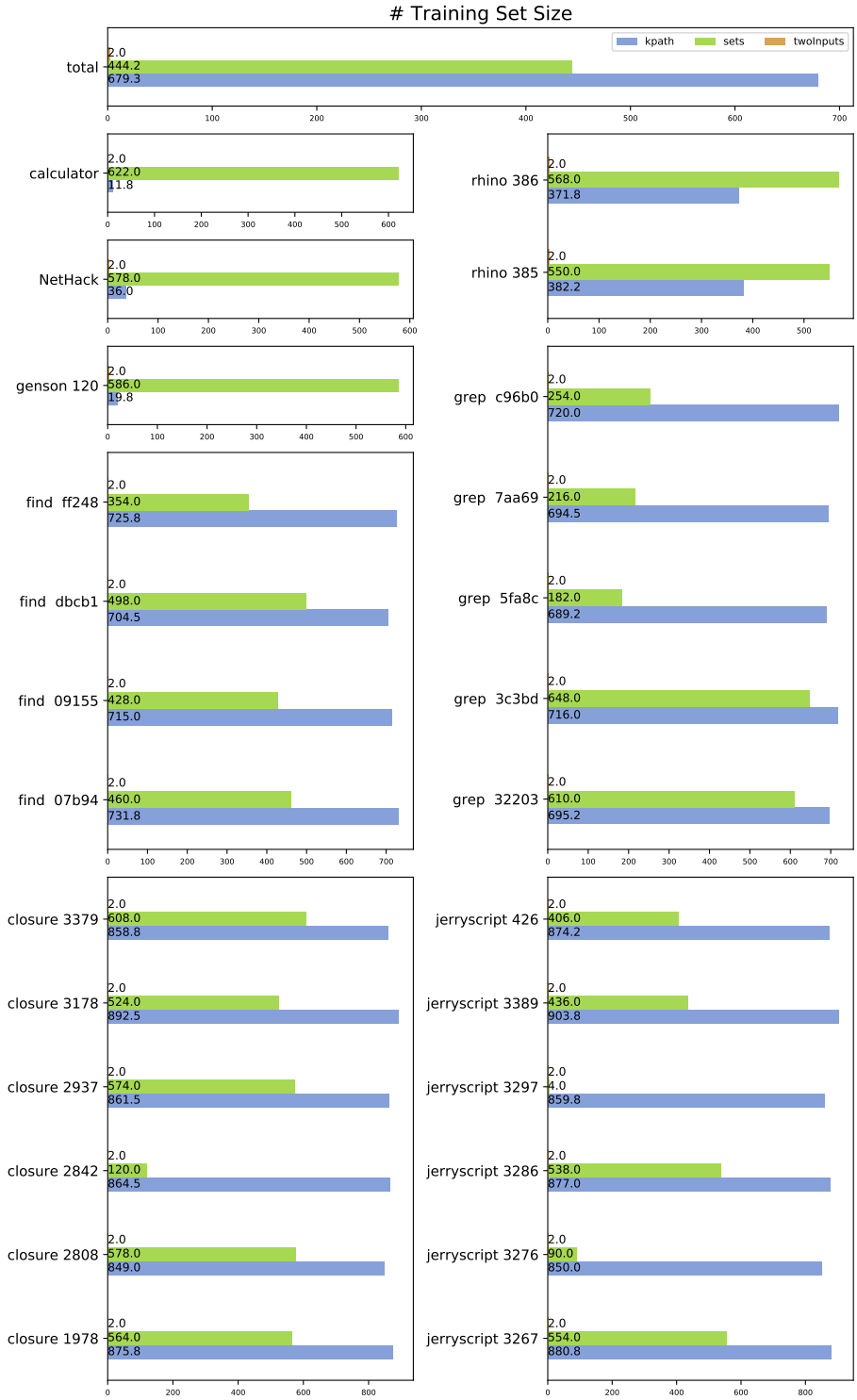


Figure 2.18: Training set sizes for ALHAZEN0 as a predictor, on all subjects and all configurations.

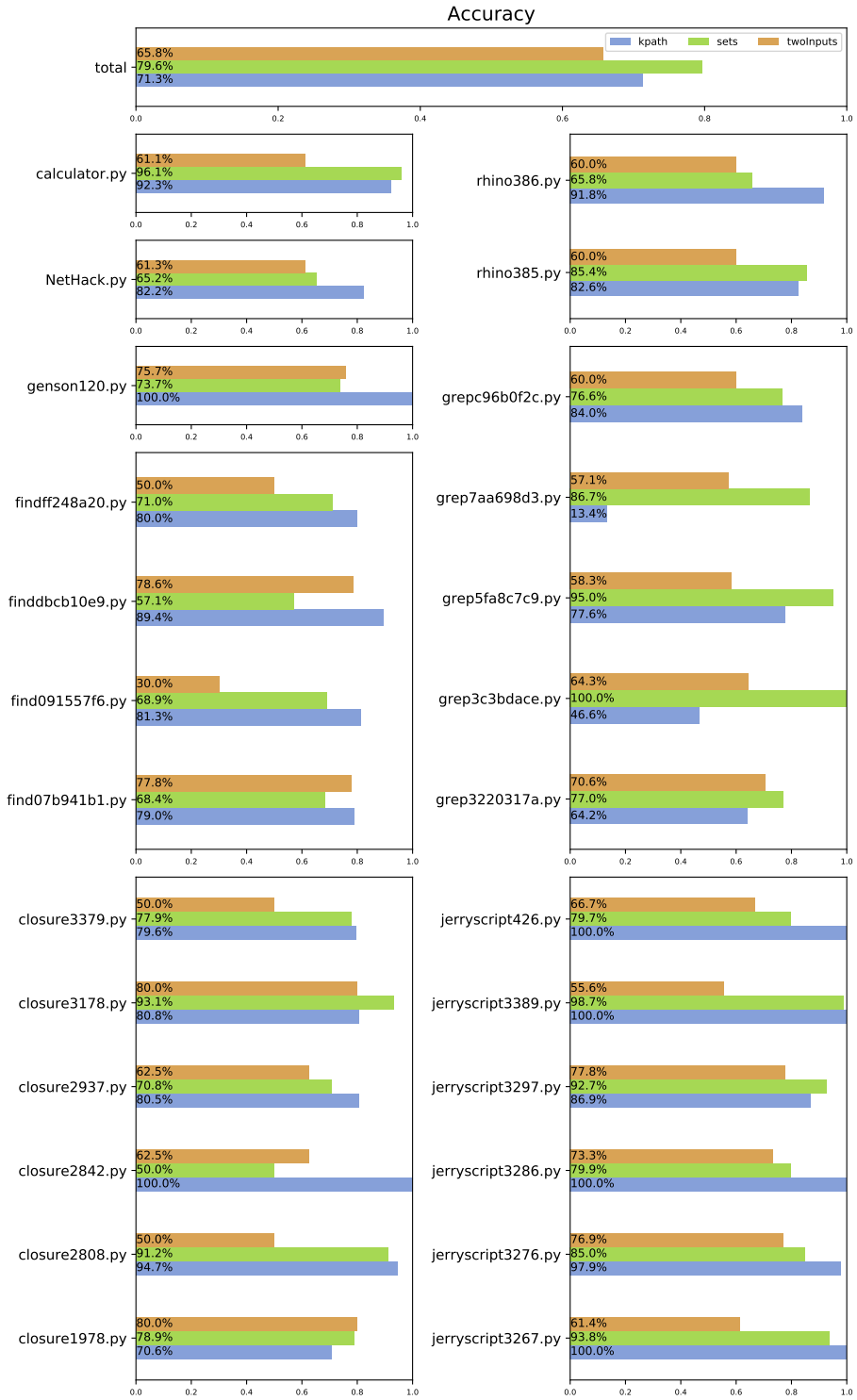


Figure 2.19: Accuracy for ALHAZEN0 as a generator, on all subjects and all configurations.

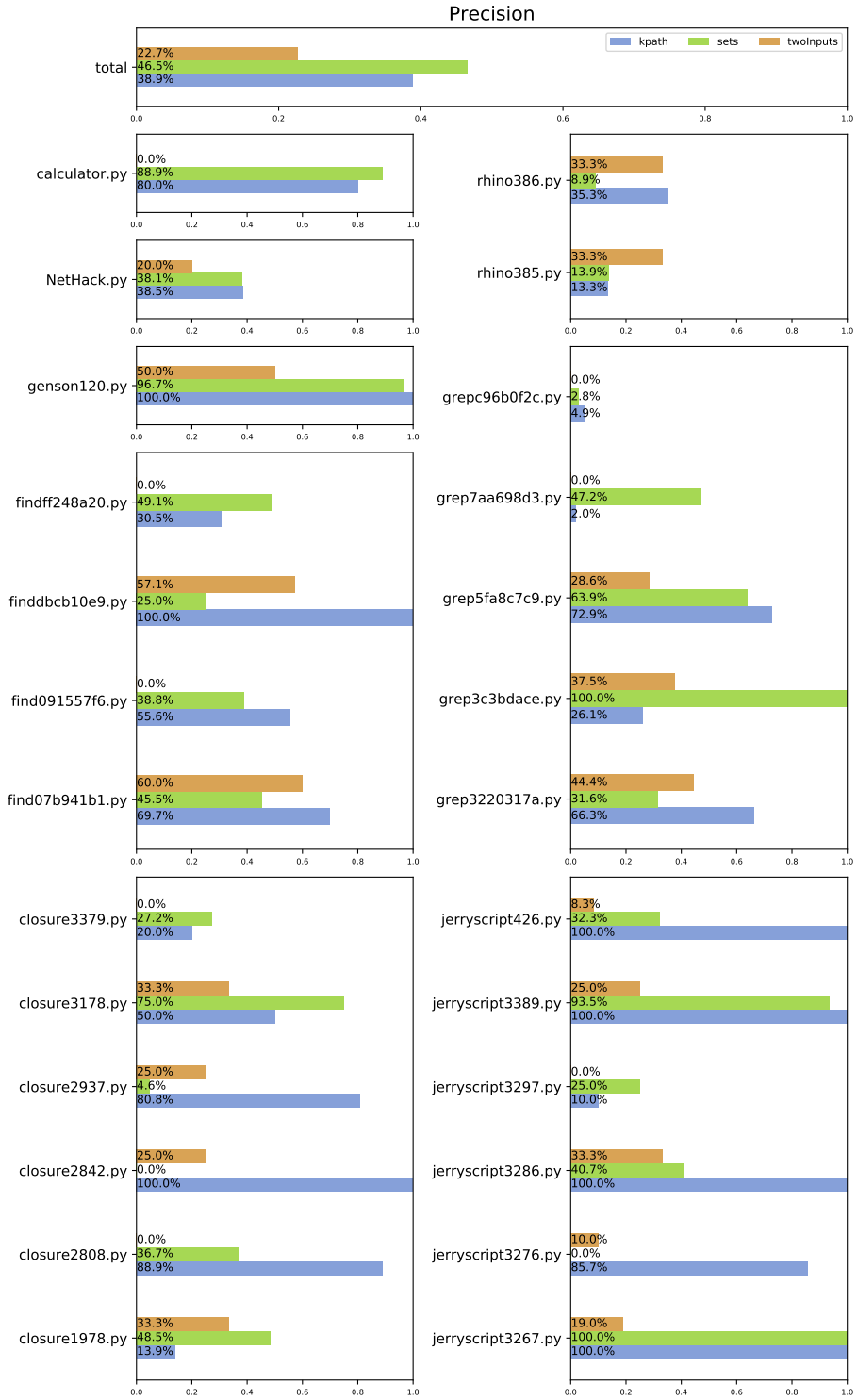


Figure 2.20: Precision for ALHAZEN0 as a generator, on all subjects and all configurations.

The reason for this might again be in the bias in the sets configuration. The probabilistic generator used for the training and verification data uses shortest derivation as a cut-off. However, many short derivations were added in grammar enrichment (see Section 2.4). So ALHAZEN0 learned to recognise samples which used the cut-off, rather than bug-triggering samples. ALHAZEN0's own generator does not use shortest derivation at all. Therefore, the generator does not re-create this bias, and the generative power is much lower than the predictive power. I will discuss this aspect further in Section 2.7.3. Despite this bias, the sets configuration is still more successful than the twoInputs and k-path configurations, as both configurations achieve worse precision and accuracy. There seems to be some usable information in the trees after all.

> When generative power and predictive power differ a lot, it is likely that there is some bias in the verification set.

The gap between accuracy and precision indicates that the non-failing class is much easier to generate than the failing class. This is not surprising: I observed as much when I generated the training data in Section 2.5.4 already.

Finally, I can only conclude that:

> ALHAZEN0 is not usable as a generator.

2.7.3 Analysis of Individual Cases

Within this section, I am going to look at the decision trees for two of the subjects. While the shape of the trees is irrelevant for filtering behavior-triggering inputs and input generation, it does matter if the trees are supposed to be looked at by humans. Also, observations in this section may point to shortcomings of the approach or reasons for the underwhelming performance of ALHAZEN0 as a generator.

Grep 3c3bdace

Bug 3c3bdace in `grep`, as found within Böhme et al. [7], is a segmentation fault which occurs if `grep` is invoked with the regex `"(^| )*"`, and the extended regex option.

ALHAZEN0 provides the decision tree shown in Figure 2.21. The diagnosis correctly states that the extended regex option is required to trigger the bug. Then, it says there has to be an expression `"^| "`. This is one of the alternatives that were added in the grammar rewrite in Section 2.4. Samples which contain this string are indeed likely to trigger the bug, but in fact, `"(^| )*"` is required. ALHAZEN0 missed the requirement for braces and an asterisk. The expression `"(^| )*"` was added by the grammar rewrite as well. ALHAZEN0 does not use it in the tree due to structural correlation : When `"(^| )*"` is part of the input sample, `"^| "` is as well. It is a substring. Within the training data, ALHAZEN0 only ever saw examples where both strings were included, and therefore any of them leads to the same precision. The training cannot decide which one to use, and arbitrarily chooses one.

However, ALHAZEN0's tree contains more information. It also reports that the bug does not occur any more if the word matching option, which make `grep`

For reference

Definition 13: Structural Correlation
If, due to the shape of the grammar, the values of two features always correlate, I call this *structural correlation*.

See Section 2.6.2

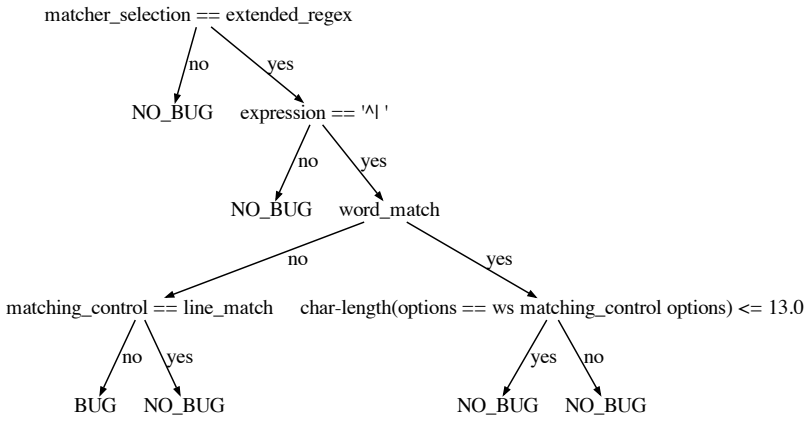


Figure 2.21: The decision tree given by ALHAZEN0 for Grep 3c3bdace.

match complete words only, is used. This is correct, and was not part of the original bug report.

ALHAZEN0 can discover information that is not included in the original bug report.

An Insight about Structural Correlation In Section 2.6.2, some work was done to eliminate structural correlation for max-char-length(), max-qu-length(), max-char() and max-numeric() features. However, I did not remove structural correlation for exists() features, even if there are some obvious cases. For example, in a derivation rule $\langle C \rangle \rightarrow \langle A \rangle \mid \langle B \rangle$, there would be exists($\langle A \rangle$), exists($\langle B \rangle$) and exists($\langle C \rangle$), even if exists($\langle A \rangle$) implies exists($\langle C \rangle$), as does exists($\langle B \rangle$). The reason for not eliminating those structural correlations can be observed in this example.

Figure 2.22 shows a fragment of the grammar used for `grep`. The feature exists($\langle$*extended_regex*$\rangle$) structurally correlates with the features exists(`"-E"`) and exists(`"--extended-regex"`), for exactly the reason shown above. There are two options to avoid structural correlation:

- Remove the exists($\langle$*extended_regex*$\rangle$) feature
- Remove the exists(`"-E"`) and exists(`"--extended-regex"`) features

⟨*extended_regex*⟩ → `"-E"` | `"--extended-regexp"`

Figure 2.22: An excerpt from the grammar used for `grep`.

Listing 2.4: JavaScript code which triggers Rhino 386.

```
1 const [y,y] = [];
```

Within the tree, exists(⟨*extended_regex*⟩) is used. This is because for this bug, it does not matter whether the long or the short form of the option is used to activate extended regex processing. So for this bug, the structural correlation should be avoided by removing the exists(`"-E"`) and exists(`"--extended-regex"`) features. It is quite possible that for another bug or another program with a similar situation in it's grammar, removing the features below an alternation is wrong, and removing the exists() feature for the alternation itself would be right. However, Alhazen0 is supposed to be general and work with any grammar and any program. So I cannot make this decision in general. This is why I decided not to fight cases of structural correlation beyond what I explained in Section 2.6.2.

Rhino 386

Rhino is a JavaScript interpreter that is written in Java. Bug 386[29] occurs if within a destructuring assignment two target variables have the same name. Listing 2.4 shows a JavaScript snippet which triggers this bug.

The first observation is that Alhazen0 cannot express this. No feature expresses the name of a variable, and even if there was such a feature, the tree cannot compare features, meaning that it could not express that two variables need to have the same name. Still, Alhazen0's sets configuration achieves a accuracy of 97.5% as a predictor and 60.8% as a generator.

The decision tree for this subject can be seen in Figure 2.23. The first observation is the size of the tree. With 8 internal nodes and 8 leaf nodes, it is rather large, and can be confusing. Second, none of the predicates within the nodes have an obvious connection to the bug. A closer look reveals what is going on:

In Section 2.5.3, I explained how the verification sets contain many substrings of the samples I generated from. For Rhino 386, the sample I generated from is 17 characters long. The predicate within the root node of the tree restricts the length of ⟨*Declaration*⟩s to 16.5 characters. Therefore, it rejects everything which is shorter than the original sample, and therefore does not contain the original sample.

In the right part, the tree checks for the existence of a break statement, and in the lower left corner, there is a check for a continue statement. `"break"` and `"continue"` are only valid within a loop, this is, however, not modeled in the grammar. If Rhino encounters a break or continue outside a loop, it terminates without executing the code. This means that the bug Alhazen0 is diagnosing does not occur. Therefore Alhazen0 tells its user that the bug only occurs in absence of a break or continue statement, or if the break statement is paired with a loop statement.

In the left part of the tree, Alhazen0 next checks for ⟨*ClassDeclaration*⟩ and

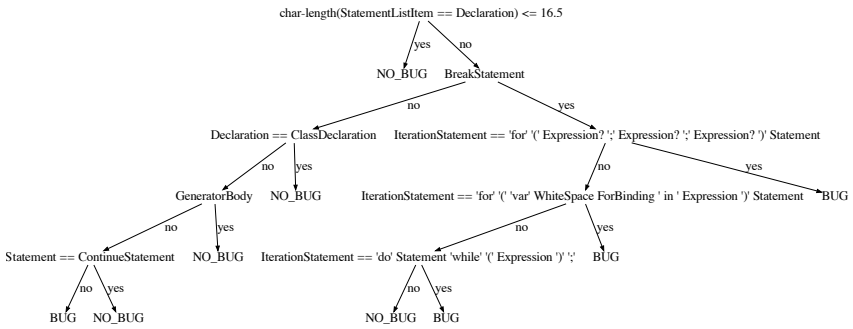


Figure 2.23: The decision tree for Rhino 386.

⟨*GeneratorBody*⟩. Both refer to parts of the grammar that contain a fair share of long keywords: Those productions are an excellent way to get to 17 characters without using a substring of the original sample. Alhazen0 rules out all samples where this happened. Therefore, Alhazen0 tells its user that the bug occurs if the sample is long enough, and the length was not caused by keywords.

This only holds for the data generated in Section 2.5.3. It is the properties of this generator, its tendency to build short samples unless the snippets from the original sample are used, which makes this true. Alhazen0 can only learn those properties if they also hold for the training data. Therefore, this bias does not exist if k-path or preexisting inputs (as in the twoInputs configuration) where used for training.

> The results for the sets configuration cannot be trusted, as Alhazen0 learned properties of the generator algorithm, rather than properties of the behavior of interest.

This leads to a second class of unwanted correlations, after structural correlation.

> **Definition 14: Coincidental Correlation**
> If, due to properties of the generator algorithm, the values of two features always correlate, I call this *coincidental correlation*.

Coincidental correlation depends on the generator algorithm. A different algorithm has a different bias, and therefore other coincidental correlations within the data. This can be seen by looking at the generator evaluation for Rhino 386. Alhazen0 achieved a precision of 8.9%, apparently it generated just very few behavior-triggering instances. The twoInputs configuration achieved a precision of 33.3%, and the k-path configuration achieved 35.3%. Both results are not good, bit better than the one for the sets configuration. The reason is that Alhazen0's internal generator, which will be described in the next chapter, has different biases. Therefore, it does not re-create the coincidental correlations that were observed in the training data, and a tree which relies on those correlations is

worthless for the generator algorithm. The twoInputs configuration and k-path configuration were not affected by this bias in generation, and therefore achieve better results as a generator.

2.8 Conclusion

ALHAZEN0, as presented in this chapter, shows promise for explaining program behavior with decision trees. The precision value, 82% for ALHAZEN0 as a predictor, indicate that ALHAZEN0 can be deployed as a filter to reject behavior-triggering inputs before they actually trigger an unwanted behavior. This may be used as a first line of defence against attacks. It may take some time to come up with a proper fix for a bug, and ALHAZEN0 can be used to protect the system in the meantime, therefore buying the developers time to implement a proper fix.

However, coincidental correlations are a huge problem. ALHAZEN0 may learn properties of the training data, rather than properties of the behavior of interest. Care needs to be taken to avoid this.

As a generator, ALHAZEN0 cannot generate sufficient numbers of behavior triggering inputs. This indicates further that ALHAZEN0 may be mislead by biased training data.

Still, examining the data brought some insights:

- ALHAZEN0 needs a mechanism to fight coincidental correlations.
- Comparing the different configurations used in this chapter, it is obvious that more inputs are not necessarily good. The sets configuration performs better as a predictor, but worse as a generator. At the same time, the two inputs configuration does the opposite. This points to over-specialisation in the two inputs configuration, and over-generalization in the sets configuration.

In the next chapter, I will explore whether a specialized generator, built to create exactly those samples that benefit the tree most, can help to improve ALHAZEN0.

Chapter 3

Refining Hypothesis with a Feedback Loop

In Section 2.7.1, I observed that carefully choosing the inputs is more important than the sheer number of inputs. In this chapter, I will present a method to systematically generate inputs that are well-suited for learning.

This can be seen as a way to automate scientific inquiry. The new tool developed in this chapter, Alhazen, postulates an initial hypothesis, using the same method as Alhazen0 in the previous chapter. This initial hypothesis suffers from the problems observed previously: Lack of precision and sensitivity to structural and coincidental correlations. A scientist now designs an experiment which can test this hypothesis, and uses the data collected in the experiment to refine it. Alhazen does the same: A custom generator produces inputs which, according to the hypothesis, trigger or do not trigger the behavior of interest. Then, these are given to the program, and the hypothesis is tested. Using the newly generated samples – and observed outcomes of the program execution – as additional data, a new hypothesis is formed.

The process can also be described as a feedback loop between decision tree learner and input generator: The decision tree learner provides a hypothesis, and the generator attempts to refute this hypothesis.

As this work aims at improving Alhazen0, it will be evaluated in the same way as Alhazen0.

3.1 Generating Predicate Sets

Within the feedback loop, Alhazen always attempts to *refute* the current hypothesis of the tree. That means, Alhazen generates samples which have the features described by one path in the tree, and checks whether the prediction of the tree is correct for those samples. This requires a representation of the current hypothesis which can be consumed by a solver. In other words, the decision tree needs to be translated into a representation that is more suitable for solving those constraints.

3.1.1 Extracting Predicates from the Trees

The predicates in the nodes of a decision tree are always of the form $f < t$, where f is a feature and $t \in \mathbb{Q}$ is a threshold value. Let p_j be a node on a path $p = p_1, \ldots p_n$. Let *predicate*(p_j) be the predicate in p_j. I form a predicate set

$$P(p) = \left\{ \begin{array}{ll} \textit{predicate}(p_j) & \text{if } p_{j+1} \text{ is the left child of } p_j \\ \neg\textit{predicate}(p_j) & \text{if } p_{j+1} \text{ is the right child of } p_j \end{array} \;\middle|\; j \in [1 \ldots n-1] \right\}$$

.

This definition reflects the semantics of the decision tree: When classifying an input, the tree follows a path from the root to a leaf. It takes the left child if the predicate in a node is fulfilled, and the right child otherwise. So if for a path p in a decision tree T, an input i fulfills all predicates in $P(p)$, classifying the input i with T would follow the path p. Due to the way they are derived, predicates are of the form $f < t$, or, if they are negated, $\neg f < t = f \geq t$. Within the feedback loop, I generate such a predicate set for each path in the tree.

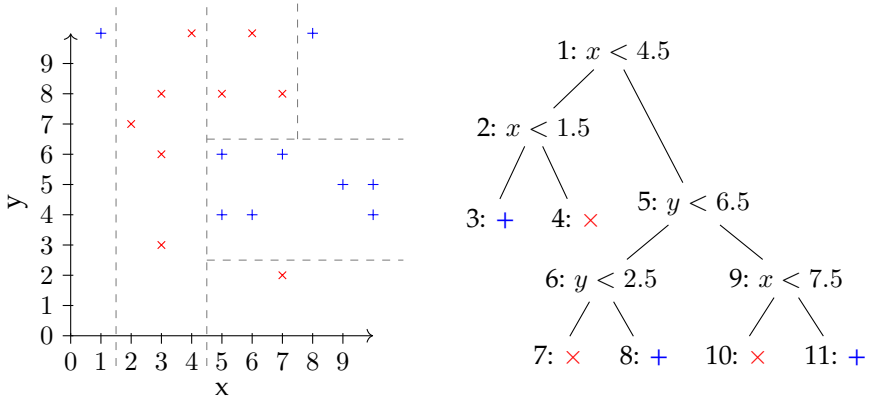


Figure 3.1: A decision tree which splits a set of points in a 2-dimensional space. The same tree can be found in Figure 2.15. The numbers before the colon in the nodes are an index.

To give an example, I use a decision tree which splits a set of points in a 2-dimensional space. The same example was used in Section 2.6.3. The decision tree can be seen on the right of Figure 3.1.

The extraction process considers all path in the tree one after another. The left-most path, which ends in +, can be given by the indexes of its nodes. It is $p_1 = 1, 2, 3$. All of the branches are left children, so the predicate set for this path is

$$P(1, 2, 3) = \{\textit{predicate}(1), \textit{predicate}(2)\} = \{x < 4.5, x < 1.5\}$$

The second path takes the left branch at the root node and the right branch in its child. It is $p_2 = 1, 2, 4$. Within the predicate set, the second constraint is negated.

$$P(1, 2, 4) = \{\textit{predicate}(1), \neg\textit{predicate}(2)\} = \{x < 4.5, x \geq 1.5\}$$

the predicate set $\{x < 4.5, x \geq 1.5\}$ is generated.

For the third child node, the predicate set is $\{x \geq 4.5, y < 6.5, y < 2.5\}$. This time, the predicate that was generated from the root node is negated. The remaining predicate sets are $\{x \geq 4.5, y < 6.5, y \geq 2.5\}$, $\{x \geq 4.5, y \geq 6.5, x < 7.5\}$ and $\{x \geq 4.5, y \geq 6.5, x \geq 7.5\}$. It can be observed that each of those predicate sets corresponds to one of the rectangular areas that are given by the splitting hyperplanes shown on the left of Figure 3.1.

3.1.2 Simplifying the Predicate Sets

The predicate sets obtained in the previous section are quite verbose. Therefore, I apply some simple steps to remove abundant predicates. If there is more than one predicate for the same feature, the predicates can be simplified. For predicates $f < h_1$ and $f < h_2$, I can replace both by a predicate $f < \min(h_1, h_2)$. For predicates $f \geq h_1$ and $f \geq h_2$, I can replace both by a predicate $f \geq \max(h_1, h_2)$. For predicates $f < h_1$ and $f \geq h_2$, I can replace both by a predicate $f \in [h_2, h_1]$.

Also, the decision tree treats everything as a continuous feature. However, within ALHAZEN, the predicates of the form exists($\langle c \rangle$)opv are in fact binary: exists() features are always 0 or 1. Still, the decision tree contains predicates of the form exists($\langle c \rangle$)< 0.5 or exists($\langle c \rangle$)≥ 0.5. I rewrite those to exists($\langle c \rangle$) or $\neg$exists($\langle c \rangle$) respectively.

Within the example, this means that the predicate set $\{x < 4.5, x < 1.5\}$ can be simplified to $\{x < 1.5\}$. The example contains no binary feature, and therefore the second simplification cannot be illustrated.

All in all, this simplification is much more effectful on the example than on real-world data, as the example has just two features, and therefore the third predicate in a set necessarily uses a feature that appeared in another predicate before.

3.1.3 Exploring Beyond known Search Space Areas

If I solve the predicate sets obtained in the previous sections, I get predicates which exercise one decision path in the tree. Looking at Figure 3.1, those samples are in an area of the input space that already contains samples. They only refute the current hypothesis, and therefore provide additional information, if the prediction of the tree happens to be wrong. While this happens sometimes, new combinations of predicates are more likely to yield this result. In the example, a point in the lower left corner, for example $(1, 1)$ is certainly more interesting than yet another point within the area defined by $\{x \geq 4.5, y < 6.5, y \geq 2.5\}$, which contains 7 + already.

Therefore, I generate additional predicate sets. For each existing predicate set P, I take all subsets P_{sub}, and for each of them, I build a new predicate set

$$P' = \{p | p \in P \wedge p \notin P_{sub}\} \cup \{\neg p | p \in P_{sub}\}$$

For instance, the predicate set $\{x \geq 4.5, y < 6.5, y \geq 2.5\}$, I would generate 7 additional sets:

1. $\{x \geq 4.5, y < 6.5, y < 2.5\}$,
2. $\{x \geq 4.5, y \geq 6.5, y \geq 2.5\}$,

For reference

Definition 13: Structural Correlation
If, due to the shape of the grammar, the values of two features always correlate, I call this *structural correlation*.

See Section 2.6.2

For reference

Definition 14: Coincidental Correlation
If, due to properties of the generator algorithm, the values of two features always correlate, I call this *coincidental correlation*.

See Section 2.7.3

3. $\{x \geq 4.5, y \geq 6.5, y < 2.5\}$,
4. $\{x < 4.5, y < 6.5, y \geq 2.5\}$,
5. $\{x < 4.5, y < 6.5, y < 2.5\}$,
6. $\{x < 4.5, y \geq 6.5, y \geq 2.5\}$ and
7. $\{x < 4.5, y \geq 6.5, y < 2.5\}$,

Some of those were present already, as they are the same as extracted from other parts of the tree, but some describe regions of the input space which were not yet explored.

3.1.4 Breaking Correlations

Within Section 2.6.2 and Section 2.7.3, I explained the problem of structural correlation and coincidental correlation. As a reminder, a coincidental correlation is if, due to some property of the generator algorithm, all samples which solve the predicate $f_1 \leq v_1$ also solve $f_2 \leq v_2$. A structural correlation is if all samples which solve $f_1 \leq v_1$ also solve $f_2 \leq v_2$, because other solutions are ruled out by the grammar.

In Section 2.7.3, I showed an example of why structural correlation cannot be avoided entirely. Still, I would like to have no coincidental correlations in my data, as they support wrong believes within the tree. This section shows how to use the feedback loop to counteract coincidental correlations. I implemented this for Alhazen's feedback loop, but did not use it for the evaluations in Section 2.7.2 and Section 3.3.2. The reason for this decision is that while coincidental correlations are problematic within the feedback loop, they are irrelevant to the creation of additional samples.

Coincidental correlation exist, because the generator may be biased towards specific control forms or grammar structures. One reason for this may be that the generator tries to generate short samples, and those control forms or structures are especially short. I believe that other generators would also have some coincidental bias, even if it may not be for short samples.

However, the generator is still capable of providing samples which do not. But then, if a predicate $f_1 < v_1$ coincidentally leads to $f_2 < v_2$, solving a predicate set $P = \{f_1 < v_1, f_2 \geq v_2\}$ provides a sample which does not exhibit

this coincidental correlation. However, if the correlation between $f_1 < v_1$ and $f_2 < v_2$ is structural, the predicate set P is infeasible.

This can be applied to counteract coincidental correlations within ALHAZEN. If I observe that two features f_1 and f_2 always correlate within the available training data, I instruct the generator to generate samples where this is not the case.

In order to implement this idea, I need to find correlations within the training data. For this purpose, I use Spearman's correlation coefficient. This measure handles ranked data, which means that it can be used for continuous data, such as the values for max-numeric() features, and ordinal data, such as the values for exists() features. Also, it is non-parametric, so the distribution of feature values within my data is of lesser importance.

I select pairs of features f_1 and f_2 such that, in all existing data, the absolute correlation coefficient between those features is at least 0.6. Then, I say that f_1 enables f_2 if is smaller than -0.6, and f_1 disables f_2 if the correlation coefficient is larger than 0.6. I chose this value, because Cohen [11] suggests to call a correlation strong as soon as the correlation coefficient is larger than 0.6.

Then, I generate additional predicate sets according to the following rules:

1. For every predicate set P that contains a predicate which uses a feature f,
   - I add an additional predicate set $P \cap \neg f'$ for all exists() features f' that are enabled by f.
   - Then, I add an additional predicate set $P \cap f' < v$ for all features f' that are enabled by f.

   v is the mean of the values for feature f' in all data observed so far.
2. For every predicate set P that contains a predicate which uses a feature f,
   - I add an additional predicate set $P \cap f'$ for all exists() features f' that are disabled by f.
   - Then, I add an additional predicate set $P \cap f' > v$ for all features f' that are disabled by f.

   v is the mean of the values for feature f' in all data observed so far.

It is important to add additional feature sets, rather than replacing the existing predicate sets. The algorithm cannot tell apart structural and coincidental correlation. However, if the additional predicates are added for cases of structural correlation, they are infeasible, as the predicates demand something that is not possible within the grammar. Therefore, this step adds several infeasible predicate sets. The generator algorithm will detect and filter them out later.

3.2 Generating Grammar Words

After extracting predicate sets from decision trees in the previous section, this section describes how to generate more inputs from the predicate sets.

Those inputs need to fulfill two sets of constraints: The first one is given by the predicate set, and expressed in $\leq$, $>$ and range conditions over the features

⟨*Expression*⟩	→	⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
⟨*UnaryExpression*⟩	→	⟨*Literal*⟩ | ⟨*Invocation*⟩
⟨*Invocation*⟩	→	⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"`
⟨*Function*⟩	→	`"sqrt"` | `"cos"` | `"tan"` | `"sin"`
⟨*Literal*⟩	→	`/[1-9][0-9]*/`

Figure 3.2: A more complex grammar for a calculator.

used by Alhazen. The second one is given by the grammar: Inputs need to be valid grammar words.

This is difficult, because the predicates often use a different interpretation than the grammar: If one of the predicate is max-numeric(⟨*number*⟩)≤ 9, and the grammar specifies that ⟨*number*⟩ → `/9[1-9]+/` | `"5.32"`, it is obvious that `"5.32"` is the only solution which fulfills both, the restrictions posed by the grammar, and the predicate. However, numeric constraint solvers would not be able to interpret the constraint given by the grammar, and grammar-based generators argue about individual characters, they cannot know that the sequence `5`, `.`, `3`, `2` is a number, and smaller than 9.

The algorithm I developed works in three steps:

1. It starts with a grammar rewrite, which removes parts of the grammar which are prohibited by the constraints. The details will be described in Section 3.2.2.

2. It performs a feasibility check, to avoid running the costly search on predicate sets which are obviously infeasible. The details will be described in Section 3.2.3.

3. The generator itself consists of two intertwined search algorithms. The inner search is a greedy search which searches for one candidate parse tree within the grammar. The outer search is a heuristic search within the space of all possible trees. Within the outer search, the inner search is used to generate candidate solutions, the outer search modifies the inner searches starting conditions, until a candidate which fulfills the predicate set is obtained. The details of the outer search can be found in Section 3.2.5, and the inner search will be described in Section 3.2.4.

Before jumping into the details of the algorithms, I present some definitions and observations that will help in understanding the algorithm.

3.2.1 More Properties of Control Forms

This section presents additional properties of grammars, which will be used in the algorithm in the following sections. Impatient readers may skip it, and come back when the definitions are used in the following sections.

Unfortunately, the running example, the calculator grammar, is not complex enough to serve as an example for most of these observations. Therefore, I am going to use the grammar in Figure 3.2 throughout this section. This grammar

For reference

Definition 4: Grammar Graph
The *Grammar Graph* for a grammar G is a directed graph. Its nodes are the control forms of the grammar G. Two nodes $\langle C_i \rangle$ and $\langle C_j \rangle$ are connected with an edge if

1. $\langle C_i \rangle$ is a reference, and there is a production rule $\langle C_i \rangle \rightarrow \langle C_j \rangle$; or
2. $\langle C_i \rangle$ is an alternation, and $\langle C_j \rangle$ is part of the sequence; or
3. $\langle C_i \rangle$ is a concatenation, and $\langle C_j \rangle$ is part of the sequence; or
4. $\langle C_i \rangle$ is a quantification, and $\langle C_j \rangle$ is its subject

See Section 2.2

For reference

Theorem 1:
Let G be a grammar with start symbol $\langle S \rangle$, and a production rule $\langle S \rangle \rightarrow \langle s \rangle$.
For every path in the grammar graph that starts in $\langle s \rangle$, a parse tree which contains a path with the same node labels can be constructed.

See Section 2.2

has been used in Section 2.2 already, and the corresponding grammar graph can be found in Figure 2.5.

Reachability

Definition 15: Reachability and Distance
A control form $\langle C_i \rangle$ *is reachable* from a control form $\langle C_j \rangle$, iff there is a path from $\langle C_j \rangle$ to $\langle C_i \rangle$ within the grammar graph.
The length of the shortest path from $\langle C_j \rangle$ to $\langle C_i \rangle$ is the *distance* from $\langle C_j \rangle$ to $\langle C_i \rangle$. If there is no such path, the distance is infinite.

If there is a path from $\langle C_j \rangle$ to $\langle C_i \rangle$, as required by the definition, Theorem 1 says that there is a parse tree which contains this path as well. Due to the fact that paths are directed, $\langle C_i \rangle$ is in the subtree rooted at $\langle C_j \rangle$ in this parse tree. Therefore, reachability expresses whether it is possible to derive a control form, starting at a specific other control form. Moreover, the distance expresses how much work is required to derive a node labeled with $\langle C_i \rangle$, starting at a node labeled with $\langle C_j \rangle$. That is because the length of the path has an impact on the size of the subtree.

Table 3.1 gives some examples of distances within the grammar in Figure 3.2, which is a replication of the grammar graph in Figure 2.5. The first column and second column in the table swap start and end node, but have different length, due to the fact that the graph is directed. As is obvious from this example, distance is not symmetric.

From	To	Distance
⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩	⟨*Expression*⟩ `"+"` ⟨*Expression*⟩	1
⟨*Expression*⟩ `"+"` ⟨*Expression*⟩	⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩	2
⟨*Expression*⟩	`"sqrt"`	8
⟨*Literal*⟩	⟨*UnaryExpression*⟩	not reachable

Table 3.1: Some examples of distance and reachability within Figure 3.2.

Minimal Length

Definition 16: Minimal Length
The *minimal length* of a control form ⟨*C*⟩ is the length of the shortest word that can be derived from ⟨*C*⟩.

The minimal length can be computed with the same algorithm as the shortest derivation. This algorithm was given in Listing 2.1. Only a few cases need to be changed. For terminals, the minimal length is the length of the terminal symbol. For references and alternations, the minimal length of the child node is used directly, and not incremented (Line 10). The same is true for quantifications annotated with `+` (Line 15). Quantifications with `?` or `*` have a minimal length of 0 (Line 19). For concatenations, the +1 in Line 21 must be omitted.

The algorithm can handle strongly connected components, just as the algorithm for `SHORTEST_DERIVATION` can.

Within the example, the minimal length for ⟨*Function*⟩ is 3, which is the length of the `"tan"`, `"cos"` and `"sin"` terminal symbols. The minimal length for ⟨*Expression*⟩ is 1. This length can be achieved by using a ⟨*Literal*⟩ as an ⟨*Expression*⟩, via ⟨*UnaryExpression*⟩. ⟨*Expression*⟩ belongs to a strongly connected component, and therefore a real exit from this component is required for the minimal length. This real exit is ⟨*Literal*⟩. Furthermore, the concatenation ⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"` has a minimal length of 6, the sum of the minimal length of all its members.

Maximal Length

Definition 17: Maximal Length
The *maximal length* of a control form ⟨*C*⟩ is the length of the longest word that can be derived from ⟨*C*⟩.

The calculation of the maximal length is somewhat easier than that of the minimal length. The reason is that for every node within a strongly connected component, the maximal length is infinite.Therefore, the maximal length can be calculated bottom-up:

For concatenations, the maximal length is the sum of the maximal length of all

children.

For alternations, the maximal length is the longest maximal length of all options.

For quantifications with + or *, the maximal length is infinite.

For quantifications with ?, the maximal length is the maximal length of the subject.

Within the example in Figure 3.2, the maximal length for ⟨*Function*⟩ is 4, as the largest reachable terminal symbol, which is `"sqrt"`, is relevant here. For ⟨*Expression*⟩, the maximal length is infinite, as ⟨*Expression*⟩ is in a strongly connected component. Whenever there is an ⟨*Expression*⟩, one can generate a longer ⟨*Expression*⟩ by adding a `"+"` and another expression in the end. Due to the presence of an element with infinite maximal length in the concatenation, ⟨*Invocation*⟩ has infinite maximal length just as well.

3.2.2 Rewriting the Grammar for Excludes

The grammar rewrite presented in this section is the first step of the search for grammar words that fulfill the predicate sets from Section 3.1. In this step, everything that is prohibited by the predicates will be removed from the grammar. As a general remark, removing more control forms helps the algorithm later on: A smaller grammar is obviously easier to search.

There are three ways how the predicates can prohibit something:

1. A ¬exists(⟨C⟩) constraint means that ⟨C⟩ is prohibited.
2. A constraint max-char-length(⟨C⟩) $\leq v$ prohibits ⟨C⟩ if the minimal length of ⟨C⟩ is larger than v.
3. A constraint max-char(⟨C⟩) $\leq v$ prohibits ⟨C⟩ if no terminal symbol which contains a codepoint smaller than v is reachable from max-char(⟨C⟩).

In all three cases, ⟨C⟩ is removed from the grammar.

Removing control forms from the grammar needs to be done recursively: If a control form ⟨C⟩ is removed, and there is a production rule ⟨A⟩ $\rightarrow$ ⟨C⟩, ⟨A⟩ should be removed as well, and all references to ⟨A⟩ should be removed. If there is a concatenation which includes ⟨C⟩, the entire concatenation is to be removed. If there is a quantification with the subject ⟨C⟩, the entire quantification is to be removed. If there is an alternation which includes ⟨C⟩, ⟨C⟩ is to be removed from the alternation, while the alternation itself and the other members of the sequence are preserved.

3.2.3 Checking Feasibility

The search algorithm, described in the next sections, will not terminate on infeasible cases. However, the algorithm in Section 3.1, especially the mechanism to break coincidental correlations, generates quite a few infeasible predicate sets. Therefore, checking predicate sets for feasibility before attempting to solve them is an important optimization. In practice, I cannot detect all infeasible

For reference

Definition 2: Parse Trees and Words in the Language
A *parse tree* for a context-free grammar (N, T, P, S) is a tree where each node is labeled with a control form. The root node is labeled with a control form $\langle s \rangle$ such that $S \rightarrow \langle s \rangle$ is in P. If a node is labeled with

1. a reference $\langle A \rangle$, it has one child which is labeled with a control form $\langle C \rangle$, such that $\langle A \rangle \rightarrow \langle C \rangle$ is a production rule in P.
2. a terminal symbol, it has no children (it is a leaf node).
3. an alternation $\langle C_1 \rangle$ | ...| $\langle C_n \rangle$, it has one child, labeled with one of the $\langle C_i \rangle$'s.
4. a concatenation $\langle C_1 \rangle$ ...$\langle C_n \rangle$, it has n children, where the i-th child is labeled with $\langle C_i \rangle$.
5. a quantification, all its children are labeled with the subject of the quantification. The node has one or more children if the annotation is +, zero or more children if the annotation is $*$ and zero or one children if the annotation is ?.

The sequence of terminal symbols that consists of the labels of the leaves labeled with a terminal symbol of the parse tree in preorder is the *leaf word* of this tree. It can also be said that the parse tree *derives* its leaf word. All words which have valid parse trees, that is, trees formed according to the rules above, are *words of the language*.

See Section 2.1

cases, which is why I run the search algorithm with a timeout. But as the feasibility check is usually faster than the timeout, it improves performance if I detect infeasibility earlier, and avoid an invocation of the search algorithm. I implemented the following feasibility checks:

Reachability Check Almost all predicates require a specific control from to be used. This is obviously true for exists() predicates, but holds for max-numeric(), max-char(), max-char-length() and max-qu-length() just as well: Numeric interpretations as well as length have a default value if there is no node labeled with the required control form.

In the previous step, I performed a grammar rewrite, which means that control forms were removed, and others are unreachable now. But then, all predicates which require one of those control forms are infeasible. Therefore, I implemented a feasibility check which considers a predicate set infeasible if for any predicate within the set, the control form used within the predicate is unreachable or has been removed from the rewritten grammar.

Feasible Interval Check max-char-length() predicates require the leaf word below a given control form to have a specific length. Within Section 3.2.1 and Section 3.2.1, I gave definitions and showed how to calculate the minimal and maximal length of the leaf word for a control form. Now, I use those values to

check whether it is possible to derive a leaf word within the required length, and consider the predicate as infeasible otherwise.

Feasible Quantification Check A predicate max-qu-length$(\langle C \rangle) < 0$ is infeasible if $\langle C \rangle$ is a quantification annotated with +, as those need at least one child within the parse tree.

3.2.4 Greedily Searching for Candidate Trees

In this section, I will describe an algorithm, which, given a grammar and a set of predicates over features, generates a parse tree which has features which fulfil the predicates. Along with the parse tree, it generates a decision sequence, which will be used in the following section. The algorithm is going to be a *best-effort algorithm*, which means that it may, occasionally, give a result which does not fulfill all predicates.

The Rating Function

The behavior of the algorithm can be controlled by the choice of a rating function r. I will first describe the algorithm, without providing the details of the rating function. Afterwards, I define a rating function which is based on a set of predicates, and which steers the algorithm towards generating parse trees which fulfil the predicates.

> **Definition 18: Rating Function**
> Let $\mathbb{Q}'$ be a set with a total order and three values MIN $\in \mathbb{Q}'$, MAX $\in \mathbb{Q}'$ and NO $\in \mathbb{Q}'$ such that for all $q \in \mathbb{Q}'$, MIN $\leq q$, MAX $<$ NO and $q \leq$ NO, and there is no $q \in \mathbb{Q}'$ such that MAX $< q$ and $q <$ NO.
> For a partial parse tree t, a node n in t and a control form $\langle c \rangle$, a *rating function* r is a function $r(t, n, \langle c \rangle)$ such that for all t, n and $\langle c \rangle$, $r(t, n, \langle c \rangle) \in \mathbb{Q}'$.

Unless specified otherwise, I will choose $\mathbb{Q}'$ as a subset of $\mathbb{Q}$. This can be done by choosing values for MIN $\in \mathbb{Q}$, MAX $\in \mathbb{Q}$ and NO $\in \mathbb{Q}$ and setting $\mathbb{Q}'$ as an interval $[\text{MIN} \ldots \text{MAX}] \cup \{\text{NO}\}$. This resonates well with modern computers: Floating point numbers, as defined in IEEE 754, represent just an interval of $\mathbb{Q}$ anyways. There are three options on how to implement NO:

- Choose a larger value than the one chosen for MAX. Mind that, due to floating-point imprecision, equality comparisons to this value may be problematic.
- Use the special value `NaN`, which is provided by IEEE 754. This requires custom logic for ordering, as comparisons to `NaN` usually do not work as desired.
- Implement functions $r_{\backslash p}$ and p, such that

$$r_{\backslash p}(t, n, \langle c \rangle) = \begin{cases} r(t, n, \langle c \rangle) & \text{if } r(t, n, \langle c \rangle) \neq \text{NO} \\ \text{MAX} & \text{else} \end{cases}$$

  and $p(t, n, \langle c \rangle) =$ true if and only if $r(t, n, \langle c \rangle) =$ NO.

The last option works for ordering, as all values are within the range defined by IEEE 754, and it can do equality checks versus NO, as an equality check can be implemented by calling p. Therefore it is the option that I choose in my implementation.

Generating Trees based on a Rating Function

The algorithm for the greedy generator is given in Listing 3.1. When invoked on a control form, this algorithm generates a parse tree node for the new control form (Line 3). It then proceeds to recursively generate child nodes as demanded for the production rule for $\langle c \rangle$. This essentially generates the nodes of the tree in pre-order. If the node to be generated corresponds to an alternation (Line 7) or quantification (Lines 14, 18 and 23), the rating function is used to decide which alternative should be used, or whether the subject shall be instantiated respectively. This decision is added to the decision sequence.

The algorithm prefers control forms with smaller r, and never uses control forms with $r = \text{NO}$. Therefore, the difference between $r = \text{MAX}$ and $r = \text{NO}$ is that control forms with a rating of MAX may be used, even if the rating function indicates that it is not beneficial to do so. For some decisions, it may happen that all control forms have a rating of MAX, in which case the algorithm invokes a close-off procedure to take a decision. The close-off procedure will be discussed in Section 3.2.4. Control forms with a rating of NO will never be used. If in Line 11 or Line 7 all control forms have a rating of NO, the algorithm fails. However, all rating functions used within this thesis are built such that the rating of an alternation $\langle A_1 \rangle | \ldots | \langle A_j \rangle$ is NO if the rating for all $\langle A_i \rangle$ is NO, and the rating of a concatenation $\langle A_1 \rangle \ldots \langle A_j \rangle$ is NO, if the rating for any $\langle A_i \rangle$ is NO.

Let's look at how the algorithm generates `"sqrt(9)"`, using the grammar in Figure 2.1. In order to do so, I define a rating function r_T for an existing tree T.

Definition 19: Reconstructing rating function r_T
For a parse tree T, the *reconstructing rating function* r_T is defined as:

$$r_T(t, n, \langle c \rangle) = \begin{cases} \text{MIN} & \text{if } t \text{ is a subtree of } T, \text{ and there is a child labeled} \\ & \text{with } \langle c \rangle \text{ below } n \text{ in the existing tree } T. \\ \text{MAX} & \text{otherwise} \end{cases}$$

As we will see in the example, this rating function has the algorithm reconstruct an existing tree. It always leads the algorithm to generating the nodes which exist within the pre-existing T. I will use this rating function in all examples, until I introduce the function ALHAZEN actually uses.

When reconstructing `"sqrt(9)"`, the first call is to `Greedy-Generate(`$\langle$*start*$\rangle$, ϵ, ϵ`)`, where ϵ is an empty tree or a non-existent node respectively. $\langle$*start*$\rangle$ is a reference, so after creating a node `start` labeled with $\langle$*start*$\rangle$ (Line 3), the first case of the algorithm is triggered (Line 4), and `Greedy-Generate(`$\langle$*function*$\rangle$ `"("` $\langle$*number*$\rangle$ `")"`, `start`, `start)` is invoked. This generates a node `concat` (Line 3), and then the concatenation triggers the third case (Line 11). The members of the concatenation are ordered by r_T. As r_T is defined to mimic the existing tree in this example, r_T is MIN for all of them, and therefore the order remains

Listing 3.1: Greedy search algorithm for generating a parse tree for a given predicate set.

```
1  // Input: A control form ⟨c⟩, a node n' in a partial parse tree t
2  Greedy-Generate(⟨c⟩, t, n_p):
3        Create a new node n in t as a child of n_p and label it with ⟨c⟩
4        if ⟨c⟩ is a Reference:
5              find the production rule ⟨c⟩ → ⟨b⟩
6              Greedy-Generate(⟨b⟩, t, n)
7        if ⟨c⟩ is an alternation ⟨c_1⟩ | …| ⟨c_m⟩:
8              add   ⟨c⟩ to the decision sequence
9              choose ⟨c_j⟩ such that r(t, n, ⟨c_j⟩) is not NO and minimal for
                   j ∈ [1, . . . , m]
10             Greedy-Generate(⟨c_j⟩, t, n)
11       if ⟨c⟩ is a concatenation ⟨c_1⟩ …⟨c_m⟩:
12             for all ⟨c_j⟩, ordered by r(t, n, ⟨c_j⟩):
13                   Greedy-Generate(⟨c_j⟩, t, n)
14       if ⟨c⟩ is a Quantification ⟨c'⟩*:
15             while r(t, n, ⟨c_j⟩) ∉ [MAX, NO]:
16                   add   ⟨c_j⟩ to the decision sequence
17                   Greedy-Generate(⟨c'⟩, t, n)
18       if ⟨c⟩ is a Quantification ⟨c'⟩+:
19             Greedy-Generate(⟨c'⟩, t, n)
20             while r(t, ⟨c_j⟩) ∉ [MAX, NO]:
21                   add   ⟨c_j⟩ to the decision sequence
22                   Greedy-Generate(⟨c'⟩, t, n)
23       if ⟨c⟩ is a Quantification ⟨c'⟩?:
24             if r(t, n, ⟨c'⟩) ∉ [MAX, NO]:
25                   add   ⟨c'⟩ to the decision sequence
26                   Greedy-Generate(⟨c'⟩, t, n)
27       return n
```

unchanged [1]. The algorithm therefore makes four recursive calls:

1. `Greedy-Generate(`⟨*function*⟩`, {start, concat}, concat);`
2. `Greedy-Generate("(", {start, concat}, concat);`
3. `Greedy-Generate(`⟨*number*⟩`, {start, concat}, concat);` and
4. `Greedy-Generate(")", {start, concat}, concat)`

The two calls for the terminal symbols `"("` and `")"` just generate a node labeled with `"("` and `")"` respectively (Line 3), and return afterwards. From here on, I will give the full tree as `t`, it contains all nodes generated thus far. The invocation for ⟨*function*⟩ generates a node labeled with ⟨*function*⟩ (Line 3), and invokes `Greedy-Generate("sqrt" | "cos" | "sin" | "tan", t, n)`. This is an alternation, so the second case is triggered (Line 7). The algorithm checks which option of the alternation gives the minimum r. In the example, it is $r_T(t, \texttt{"sqrt"}) = \text{MIN}$, and $r = \text{MAX}$ for all other options. There-

[1] Within this example, it is not obvious why I need the reordering. There will be a more complex example to showcase in Section 3.2.4.

For reference

Theorem 0:
For every path in a parse tree, there is a path with the same node labels in the grammar graph.

See Section 2.2

For reference

Theorem 1:
Let G be a grammar with start symbol $\langle S \rangle$, and a production rule $\langle S \rangle \rightarrow \langle s \rangle$.
For every path in the grammar graph that starts in $\langle s \rangle$, a parse tree which contains a path with the same node labels can be constructed.

See Section 2.2

fore, the algorithm chooses `"sqrt"`. Calling `Greedy-Generate` for `"sqrt"` terminates immediately, so the recursive calls of the algorithm return, and `Greedy-Generate(`⟨*number*⟩`, {start, concat}, concat)` is the next invocation to be handled. It generates `"9"` in the same way as the call for ⟨*function*⟩ generated `"sqrt"`.

At this point, I would like to make an observation about the grammar graph. The path from ⟨*start*⟩ to `"sqrt"` in the grammar graph has exactly 7 nodes. This is the minimal number of calls to `Greedy-Generate` which is required to generate a node labeled with `"sqrt"` after generating a node labeled with ⟨*start*⟩. As the order of calls within the concatenation is not fixed, there may be more than 7 calls. This leads to the following theorem:

Theorem 6:
If there is a path between two control forms $\langle C \rangle$ and $\langle D \rangle$ in the grammar graph, the distance of those control forms, incremented by 1, is always the minimal number of `Greedy-Generate` invocations required to create a node labeled with $\langle D \rangle$ after creating a node labeled with $\langle C \rangle$.

Let's assume we call `Greedy-Generate` with a control form $\langle C \rangle$. If $\langle C \rangle$ is a terminal symbol, the distance from $\langle C \rangle$ to $\langle C \rangle$ is 0, and the theorem holds trivially, as there is just one invocation to `Greedy-Generate`.

If the control from $\langle C \rangle$ is either:

- an alternation $\langle C \rangle \rightarrow$ ...| $\langle C' \rangle$ | ...;
- a concatenation $\langle C \rangle \rightarrow$ ...$\langle C' \rangle$ ...;
- a quantification $\langle C \rangle \rightarrow \langle C' \rangle$*;
- a quantification $\langle C \rangle \rightarrow \langle C' \rangle$+;
- a quantification $\langle C \rangle \rightarrow \langle C' \rangle$?; or
- a reference such that $\langle C \rangle \rightarrow \langle C' \rangle$ is in the grammar

`Greedy-Generate` generates exactly one new node n, labeled with $\langle C \rangle$. In those cases, there is a recursive call to `Greedy-Generate` with a control form $\langle C' \rangle$. This generates a child node n' labeled with $\langle C' \rangle$ below n. But then, $\langle C \rangle \rightarrow \langle C' \rangle$ is a

For reference

Definition 15: Reachability and Distance
A control form $\langle C_i \rangle$ *is reachable* from a control form $\langle C_j \rangle$, iff there is a path from $\langle C_j \rangle$ to $\langle C_i \rangle$ within the grammar graph.
The length of the shortest path from $\langle C_j \rangle$ to $\langle C_i \rangle$ is the *distance* from $\langle C_j \rangle$ to $\langle C_i \rangle$. If there is no such path, the distance is infinite.

See Section 3.2.1

path in the parse tree, and due to Theorem 0 also a path in the grammar graph. Therefore, the distance from $\langle C \rangle$ to $\langle C' \rangle$ is 1, and there were two calls to `Greedy-Generate`. For longer path, the same argument can be applied iteratively.

It is only the minimal number, not the exact number, as there may be a concatenation on the path from $\langle C \rangle$ to $\langle D \rangle$. But then, siblings of a node on the path could be created before the next node on the path is created, and therefore more calls to `Greedy-Generate` may happen in between.

Lastly, it remains to show that there cannot be a shorter path. Assume that there is a path from $\langle C \rangle$ to $\langle D \rangle$ within the grammar graph, that is shorter than the number of `Greedy-Generate` invocations required incremented by 1. I prepend this path with a prefix that starts in the start symbol, and then, according to Theorem 1, there is a parse tree which contains such a path, and the execution of `Greedy-Generate` which generates this parse tree, has a number of `Greedy-Generate` invocations which is the path length incremented by 1 (assuming an optimal order of the recursive invocations for concatenations). This contradicts my assumption, and therefore Theorem 6 is proven.

Rating Function r_P

The algorithm I introduced in the previous section can generate trees, and the rating function r can be used to control the shape of those trees. In this section, I introduce a rating function r_P which considers a predicate set P, and leads the algorithm to generate a tree which fulfills many of the predicates in P.

I start by defining a rating function $r'(p', t, n, \langle C \rangle)$, which considers a single predicate p', then I use this function to define a rating function which considers all predicates p in a predicate set P.

For all predicates $r'(p, t, n, \langle C \rangle) = \text{MAX}$ if t fulfills p. This ensures that the algorithm does not try to reach a goal that is already fulfilled. As long as a predicate is not fulfilled, the value of r' depends on the predicate type.

Existence and Maximal Codepoint Predicates An existence predicate has the form exists($\langle D \rangle$) or ¬exists($\langle D \rangle$). A predicate of the form exists($\langle D \rangle$) is fulfilled if there is at least one node labeled with $\langle D \rangle$. A predicate of the form ¬exists($\langle D \rangle$) is fulfilled if there is no node labeled with $\langle D \rangle$.

Predicates of the form ¬exists($\langle E \rangle$) were handled in the grammar rewrite, and can be ignored in this step. For predicates of the form exists($\langle D \rangle$), if n is not labeled with $\langle D \rangle$, and not a subnode of a node labeled with $\langle D \rangle$, $r'(p, t, n, \langle C \rangle)$ is the distance from $\langle C \rangle$ to $\langle D \rangle$.

Theorem 6 says that, the distance is the minimal number of calls to `Greedy-Generate` that is required until a node labeled with $\langle D \rangle$ is generated. Going

For reference

Definition 19: Reconstructing rating function r_T
For a parse tree T, the *reconstructing rating function* r_T is defined as:

$$r_T(t, n, \langle c \rangle) = \begin{cases} \text{MIN} & \text{if } t \text{ is a subtree of } T, \text{ and there is a child labeled} \\ & \text{with } \langle c \rangle \text{ below } n \text{ in the existing tree } T. \\ \text{MAX} & \text{otherwise} \end{cases}$$

See Section 3.2.4

for the control form with the smallest distance to $\langle D \rangle$ therefore gets `GreedyGenerate` to generate a node labeled with $\langle D \rangle$ eventually. This is similar to a path search algorithm, which always expands the node which is closest to its target first. For existence features this is, in fact, sufficient: The predicate is fulfilled as soon as $\langle D \rangle$ was generated.

A maximal codepoint predicate has the form max-char(D) $< v$. It is fulfilled if all characters in the leaf word for all subtrees of nodes labeled with $\langle D \rangle$ have a code point smaller than v.

It is quite simple to check whether a terminal symbol fulfills this requirement. Afterwards, the sum of the distance to $\langle D \rangle$ and the minimum distance from $\langle D \rangle$ to any of those terminal symbols can be used as r'.

This has another advantage: If there was an $\langle E \rangle$, which is prohibited by the predicate set on the path to a $\langle D \rangle$, which is required, this $\langle E \rangle$ was removed from the grammar, and therefore distance calculations will not consider the path that contained $\langle E \rangle$.

Maximal Numeric Interpretation Predicates A maximal numeric interpretation predicates has the form max-numeric($\langle D \rangle$) $< v_1$. It is fulfilled if for all occurrences of $\langle D \rangle$, an interpretation of the leaf word below $\langle D \rangle$ is a decimal number smaller than v_1.

The first step in defining r' for max-numeric() predicates is to determine the valid interval:

- For a predicate $v_0 \leq$ max-numeric($\langle D \rangle$) $\in [v_0, v_1]$, this is $[v_0, v_1]$.
- For a predicate max-numeric($\langle D \rangle$) $< v_0$, this is $[v_0 - 1000, v_0]$.
- For a predicate max-numeric($\langle D \rangle$) $\geq v_0$, this is $[v_0, v_0 + 1000]$.

Technically, the valid interval for the last two cases would be $[-\infty, v_0]$ or $[v_0, \infty]$, but this proves to be impractical: The generator tries to generate extremely large numbers, which, in most cases, is not desired.

After determining the valid interval, I sample 5 equally-spaced values from the interval. Then, I try to parse each of those values with $\langle D \rangle$. This gives me a set of parse trees T, each representing the parse tree for one of the numbers. $r'(p, t, n, \langle c \rangle)$ is the distance from $\langle c \rangle$ to $\langle D \rangle$, as long as n is not in a subtree of a node labeled with $\langle D \rangle$. Afterwards, r' is the same as the minimal reconstructing rating function r_T for the parse trees T.

For reference

Definition 2: Parse Trees and Words in the Language
A *parse tree* for a context-free grammar (N, T, P, S) is a tree where each node is labeled with a control form. The root node is labeled with a control form $\langle s \rangle$ such that $S \rightarrow \langle s \rangle$ is in P. If a node is labeled with

1. a reference $\langle A \rangle$, it has one child which is labeled with a control form $\langle C \rangle$, such that $\langle A \rangle \rightarrow \langle C \rangle$ is a production rule in P.
2. a terminal symbol, it has no children (it is a leaf node).
3. an alternation $\langle C_1 \rangle$ | ...| $\langle C_n \rangle$, it has one child, labeled with one of the $\langle C_i \rangle$'s.
4. a concatenation $\langle C_1 \rangle$ ...$\langle C_n \rangle$, it has n children, where the i-th child is labeled with $\langle C_i \rangle$.
5. a quantification, all its children are labeled with the subject of the quantification. The node has one or more children if the annotation is `+`, zero or more children if the annotation is `*` and zero or one children if the annotation is `?`.

The sequence of terminal symbols that consists of the labels of the leaves labeled with a terminal symbol of the parse tree in preorder is the *leaf word* of this tree. It can also be said that the parse tree *derives* its leaf word. All words which have valid parse trees, that is, trees formed according to the rules above, are *words of the language*.

See Section 2.1

For reference

Definition 16: Minimal Length
The *minimal length* of a control form $\langle C \rangle$ is the length of the shortest word that can be derived from $\langle C \rangle$.

See Section 3.2.1

Max-QA-Length Predicates Predicate of the form $v_0 <$ max-qu-length$(\langle D \rangle) < v_1$ are fulfilled if all occurrences of $\langle D \rangle$, which is a quantification, have less than v_1 child nodes, and at least one of them has more than v_0 child nodes. To see why it is all occurrences for the upper bound, and at least one occurrence for the lower bound, remember that there is a *max* in the definition of max-qu-length().

First of all, predicates of the form max-qu-length$(\langle D \rangle)$ also require $\langle D \rangle$ to be reached. Therefore $r'(p, t, n, \langle c \rangle)$ is the distance from $\langle c \rangle$ to $\langle D \rangle$, as long as n is not in a subtree of a node labeled with $\langle D \rangle$.

As soon as n is in a subtree of a node labeled with $\langle D \rangle$, the rating function needs to ensure that the correct number of child nodes is generated. I use:

- $r'(p, t, n, \langle c \rangle) =$ MIN as long as $\langle c \rangle = \langle E \rangle$, and the number of children of `n` is smaller than v_0. This encourages the generation of child nodes.
- $r'(p, t, n, \langle c \rangle) =$ MAX if $\langle c \rangle = \langle E \rangle$, and the number of children of `n` is larger than v_0.

- $r'(p,t,n,\langle c\rangle) =$ NO if $\langle c\rangle = \langle E\rangle$, and the number of children of `n` is larger than v_1. This means that the algorithm will not generate more child nodes.

Maximal Length Predicates A maximal length predicate of the form $v_0 <$ max-char-length($\langle D\rangle$) $< v_1$ requires all subtrees with a root labeled with $\langle D\rangle$ to have a leaf word which has a length smaller than v_1, and at least one of them to have a leaf word which has a length larger than v_0.I will call v_1 the maximal allowed length of $\langle D\rangle$.

In order to define a rating function r' for those predicates, I need to argue about the length of the leaf words below a node. More precisely, I need to be able to argue about the length that a leaf word may have, once the partial parse tree is completed. This comes in two parts: For nodes which already exist in the partial parse tree, the leaf word can be examined, and its length can be calculated. If a node does not yet exist, I can still use the definition of the parse tree to predict which label it is going to have, and use the minimal length of this control form as an estimate of length.

Definition 20: Current Length of a Node in a Parse Tree
Let n be a node labeled with $\langle A\rangle$ in a (partial) parse tree. Assume that, according to the definition of a parse tree, n has a child node $n_{\langle B\rangle}$ labeled with $\langle B\rangle$. Observe that n is in a *partial* parse tree, so there may not actually be such a child node.
The *current length of* $\langle B\rangle$ *below* n is

$$curr(\langle B\rangle, n) = \begin{cases} curr(n_{\langle B\rangle}) & \text{if } n_{\langle B\rangle} \text{ exists} \\ \text{the minimal length of } \langle B\rangle & \text{otherwise} \end{cases}$$

.
The *current length of* n is

$$curr(n) = \begin{cases} curr(\langle B\rangle, n) & \text{if } n \text{ is labeled with a reference } \langle A\rangle, \text{ and there is a production rule } \langle A\rangle \rightarrow \langle B\rangle \\ \text{the minimal length of } \langle A\rangle & \text{if } n \text{ is labeled with a terminal symbol } \langle A\rangle \\ curr(n') & \text{if } n \text{ is labeled with an alternation } \langle A\rangle, \text{ and there is a child node } n' \\ \min_{j=1,\ldots,j} curr(\langle B_i\rangle, n) & \text{if } n \text{ is labeled with an alternation } \langle B_1\rangle \text{ .. } \langle B_j\rangle, \text{ and has no child node} \\ \sum_{i=1}^{j} curr(\langle B_i\rangle, n) & \text{if } n \text{ is labeled with a concatenation } \langle B_1\rangle \text{ .. } \langle B_j\rangle \\ curr(\langle B\rangle, n) & \text{if } n \text{ is labeled with a quantification } \langle B\rangle + \text{ and has no children} \\ \sum_{i=1}^{j} curr(n_i) & \text{if } n \text{ is labeled with a quantification, and has the child nodes } n_1, \ldots, n_j \\ 0 & \text{if } n \text{ is labeled with a quantification } \langle B\rangle * \text{ or } \langle B\rangle ?, \text{ and has no child nodes} \end{cases}$$

Intuitively, the current length expresses the best estimate on the length in a completion of a partial parse tree: If a node exists already, the length of its leaf word is used. Otherwise, the minimal length for this control form is used as an estimate. Therefore, during the execution of the algorithm, the current length is an underestimation of the length of this node in the parse tree that will be obtained when the algorithm terminates. As indicated by the definition, the current length for a node can be found recursively.

Assume that there is a production rule $\langle D \rangle \rightarrow \langle D_1 \rangle \ldots \langle D_m \rangle$, and a maximal length predicate $p = v_0 \leq$ max-char-length$(\langle D \rangle) < v_1$. When $r'(p, t, n, \langle D_j \rangle))$ is calculated, the maximal allowed length for $\langle D_j \rangle$ can be found: It is the maximal allowed length for $\langle D \rangle$, v_1 in the example, minus the current length of n. This can be used to propagate the maximal length requirement down the parse tree, and have a maximal allowed length for each node below a node labeled with $\langle D \rangle$.

If the minimal length of a derivation $\langle D_i \rangle$ is larger than its maximal allowed length, r' is NO, otherwise it is MAX.

Let's illustrate this with an example: Assume the algorithm is supposed to solve the predicate max-char-length$(\langle$*Expression*$\rangle) < 8$ within the grammar in Figure 3.2. It already created the partial tree shown in Figure 3.3. The algorithm needs to choose between `"sqrt"`, `"cos"`, `"sin"` and `"tan"`, to be added below $\langle$*Function*$\rangle$.

The maximal allowed length for the root node is 8, due to the predicate. This number is indicated with a subscript in Figure 3.3. The root node has one child, `"+"` with a length of 1, and a child $\langle$*Expression*$\rangle$ with a minimal length of 1. This means that the maximal allowed length for the $\langle$*Expression*$\rangle$ on the left is 6. This propagates to the $\langle$*UnaryExpression*$\rangle$, because it is the only child below $\langle$*Expression*$\rangle$, and likewise to $\langle$*Invocation*$\rangle$. The current length of the node labeled with $\langle$*Invocation*$\rangle$ is 3 (2 because of the existing children `"("` and `")"`, plus 1 for the minimal length of $\langle$*Expression*$\rangle$). Therefore, the maximal allowed length for $\langle$*Function*$\rangle$ is 3. This means that $r'(p, t, n,$ `"sqrt"`$) =$ NO, as the minimal length of `"sqrt"` is 4, and therefore more than 3. The other 3 options, `"sin"`, `"cos"` and `"tan"`, all receive a rating of MAX.

I am using MAX here on purpose: When I combine the r' into a combined rating function for a predicate set, rather than an individual predicate, I will make sure that a rating of MAX means that another r' can decide which option to choose. So by rating all options as MAX, this r' basically indicates that it does not care, and one of the other r' should make the decision.

Defining r_P, based on r' Using the rating function r' for individual predicates, I define a rating function r for a predicate set P as

$$r_P(t, n, \langle c \rangle) = \min_{p \in P} r'(p, t, n, \langle c \rangle)$$

Using min in the definition means that the algorithm is lead by the predicate where r' gives the smallest value. Intuitively, the algorithm always goes for the currently easiest problem.

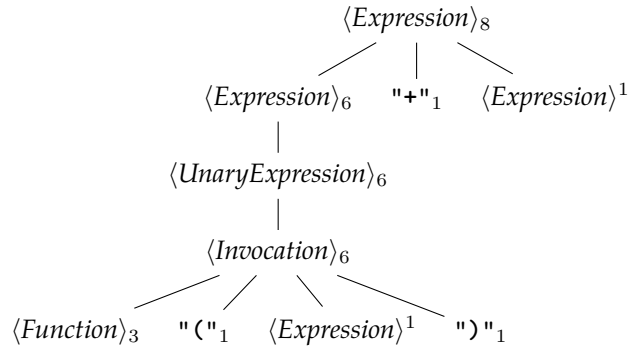


Figure 3.3: A partial tree for the grammar in Figure 2.4, generated while solving max-char-length($\langle$*Expression*$\rangle$) < 8. Subscripts in the node labels indicate the maximal allowed length of a node, while superscripts indicate the minimal length for this production.

The Close-Off procedure

As mentioned in Section 3.2.4, the algorithm falls back to a close-off procedure when multiple alternatives have the same value for the rating function r. Most ties occur when all rating functions rate a decision as MAX. That is, no predicate can be fulfilled or violated by a decision. In this case, it basically does not matter which alternative is choosen.

I implemented the close-off procedure as a secondary rating function. If the first rating function produces a tie, I get the minimal length for all involved control forms, and choose the one with the smallest minimal length. This is different from tribble and the generator in Section 2.5, which use the shortest derivation for tie-breaking. If smallest minimal length still leads to a tie, I choose an alternative randomly.

Examples

In this section, I present four examples of how the algorithm behaves in different situations. Each of the first three examples is chosen to justify one decision in the design of the algorithm. Those design decisions are:

1. The re-ordering of members in a concatenation[2].,
2. the usage of min in the definition of r, and
3. the difference between MAX and NO.

The last example shows a situation where the algorithm is unable to fulfill all predicates, and therefore motivates why I add the outer search in the next section.

Example 1: The Re-Ordering of Members in a Concatenation While the child nodes of a concatenation in the parse tree are ordered, as defined by the grammar,

[2]Thanks to Havrikov and Zeller [31]. I found this trick in their source code

⟨*Start*⟩ → ⟨*map*⟩ | ⟨*value*⟩
⟨*map*⟩ → ⟨*map*⟩ `","` ⟨*pair*⟩ | `""`
⟨*pair*⟩ → ⟨*key*⟩ `":"` ⟨*value*⟩
⟨*key*⟩ → `"key"`
⟨*value*⟩ → `"value"`

Figure 3.4: A grammar for a language that consists of a comma-separated list of key-value pairs, or a single value.

the algorithm generates them in a different order. The following example will explain why this is necessary.

The grammar in Figure 3.4 describes a format which allows either a single value, or a comma-separated list of key-value pairs. For the example, assume we want to fulfill the predicates max-char-length(⟨*map*⟩) $>= 10 \wedge$ exists(⟨*pair*⟩).

The algorithm starts by calling `Greedy-Generate`(⟨*Start*⟩, ϵ, ϵ), where ϵ is the empty tree. It generates a node labeled with ⟨*Start*⟩, and finds the derivation rule ⟨*Start*⟩ → ⟨*map*⟩ | ⟨*value*⟩. At this point, it needs to decide between ⟨*map*⟩ and ⟨*value*⟩. The algorithm calculates $r(P, \epsilon, \epsilon, \langle value \rangle) = \text{MAX}$, as no predicate uses ⟨*value*⟩. It is $r(P, \epsilon, \epsilon, \langle map \rangle) = 0$, because r(max-char-length(⟨*map*⟩) $>=$ $10, \epsilon, \epsilon, \langle map \rangle) = 0$, as this is the distance from ⟨*map*⟩ to ⟨*map*⟩. The algorithm therefore chooses ⟨*map*⟩. It generates a node labeled with ⟨*map*⟩ and finds the derivation rule ⟨*map*⟩ → ⟨*map*⟩ `","` ⟨*pair*⟩ | `""`. It generates a node labeled with ⟨*map*⟩ `","` ⟨*pair*⟩ | `""`, and needs to choose between ⟨*map*⟩ `","` ⟨*pair*⟩ and `""`. It is $r(P, t, n, \texttt{""}) = \text{MAX}$, as this does not fulfill any of the predicates, so the algorithm generates ⟨*map*⟩ `","` ⟨*pair*⟩. This is where the re-ordering of members of a concatenation takes effect. If the algorithm were to invoke `Greedy-Generate(P, t, n,` ⟨*map*⟩`)`, it would repeat the same steps as before, and, as the tree does not contain new node which are relevant for any of the predicates, take the same decisions as before. This would lead to an endless loop. Reordering means that the algorithm generates ⟨*pair*⟩ first, and when the call to `Greedy-Generate(P, t, n,` ⟨*map*⟩`)` occurs, t already contains a ⟨*pair*⟩ node, so the value of the rating function will depend on r(max-char-length(⟨*map*⟩) $\geq 10, t, \langle map \rangle)$ instead of r(exists(⟨*pair*⟩)$, t, \langle map \rangle)$, and the algorithm does not go into an endless loop.

As the length of the leaf word below ⟨*map*⟩ is 10 at this point (the leaf word below ⟨*map*⟩ is `",key:value"`), r(max-char-length(⟨*map*⟩) $\geq 10, t, n, \langle map \rangle) <$ MAX, and so the algorithm decides to create another ⟨*map*⟩ `":"` ⟨*pair*⟩. Again, the pair is generated first, and with a length of 20, r(max-char-length(⟨*map*⟩) $\geq$ $10, t, \langle map \rangle) ==$ MAX for the next invocation of r(max-char-length(⟨*map*⟩) $\geq$ $10, t, \langle map \rangle)$, so the algorithm chooses `""`. This means that there are no more decisions to be taken, and the algorithm outputs the obtained parse tree.

Example 2: min in the Definition of r Within the definition of r, I am using min to combine the value of r' for individual predicates. The following example is going to illustrate why I have to use min, rather than $\sum$ or a mean. Consider the grammar graph in Figure 3.5. I am working with two predicates exists(`"a"`) and exists(`"d"`). Within Figure 3.5, the values for r'(exists(`"a"`)$, ...)$ are given

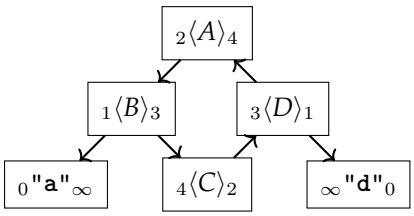


Figure 3.5: A grammar graph with a cycle. The left subscript gives the distance of a node to "a", and the right subscript gives the distance to "d".

$$
\begin{array}{lcl}
\langle \textit{number} \rangle & \rightarrow & \texttt{"-"}?\ \langle \textit{regex.1} \rangle +\ \langle \textit{regex.2} \rangle ? \\
\langle \textit{regex.1} \rangle & \rightarrow & \texttt{"0"} \mid ... \mid \texttt{"9"} \\
\langle \textit{regex.2} \rangle & \rightarrow & \texttt{"."}\ \langle \textit{regex.1} \rangle +
\end{array}
$$

Figure 3.6: A context-free grammar for numbers. Same as in Figure 2.2.

in the left subscript of a node, and the values for $r'(\text{exists}(\texttt{"d"}), ...)$ are given in the right subscript.

`Greedy-Generate` starts by generating a node labeled with $\langle A \rangle$. The recursive call is invoked with $\langle B \rangle$, and the algorithm needs to choose "a" or $\langle C \rangle$. With min in the rating function r_P, it chooses "a", and therefore fulfills one predicate. If a sum where used within the definition of r_P, the value for "a" would be MAX, as "d" is not reachable from "a". The value for $\langle C \rangle$, however, would be 6. The algorithm would therefore choose $\langle C \rangle$. Afterwards it needs to generate a $\langle D \rangle$, and then it can choose between "d" and $\langle A \rangle$. Again, for "d" the rating function, if it where defined with a sum, gave MAX. The algorithm would choose $\langle A \rangle$, therefore entering a loop.

In essence, the rating function needs to be defined such that it *monotonically decreases* with every step. This is true for each r', and can only be guaranteed for their combination, if the r' are combined with a min.

Example 3: The difference between MAX and NO In this example, I will work with the $\langle \textit{number} \rangle$ non-terminal from the grammar in Figure 3.6. I solve the predicate set $P = \{\text{max-char-length(number)} < 2, \text{max-numeric(number)} < 60\}$.

For max-numeric() predicates the algorithm starts by determining the valid interval. For the predicate max-numeric(number) < -9, it is $[-1009, -9]$. The algorithm samples 5 values from this interval, they are -1009, -759, -509, -259 and -9.

Then, it starts by expanding $\langle \textit{number} \rangle$. For sake of simplicity, I skip the reordering of the concatenation in the example. For the quantification "-"?, the first predicate gives MAX. The reason is that the maximal allowed length for $\langle \textit{number} \rangle$ is 2, and the minimal length for the quantifications $\langle \textit{regex.1} \rangle +$ and $\langle \textit{regex.2} \rangle ?$ are 1 and 0 respectively. Therefore, the maximal allowed length of "-"? is 1, which is the same as the minimal length of this control form. The second predicate gives MIN, as all five parse trees, start with a "-", such that the reconstructing rating function for those parse trees gives MIN. The value of r_P is therefore MIN, and the algorithm decides to instantiate "-". Now, $\langle \textit{regex.1} \rangle$ is instantiated.

This is not a decision, as the quantification is annotated with +, and therefore has to be instantiated at least once. Within this production, the algorithm has to decide which option of the alternative to use. The second predicate gives MIN for 1, 7, 5 and 9, as each of those occurs in one of the parse trees. The first predicate gives MAX for all of those, as non violates the allowed maximal length of 1. For the sake of the example, assume that the algorithm chooses 9. Now, it has to decide whether it wants to instantiate ⟨*regex.1*⟩ a second time, which would be allowed by the quantification. The max-numeric() predicate would encourage it as all five samples are longer than 2. It gives MIN. However, the max-char-length() gives NO, as this would violate the maximal allowed length. Therefore, no additional digit is added. This motivates why NO is needed, and I cannot rely on MAX: If the first predicate gave MAX, the algorithm would still choose the smaller value, which is MIN, and therefore it would chose to expand the quantification once more. Further on, the quantification ⟨*regex.2*⟩* is not instantiated, as no of the samples for the max-numeric() predicate demand it, and it would violate by the maximal allowed length, which is 0 for this control form now. The algorithm terminates with a leaf word of `"-9"`, which is the only option that fulfills both predicates.

As the astute reader may have noticed, this result is not guaranteed. When the algorithm instantiated ⟨*regex.1*⟩, it could have chosen 1, 7, 5 and 9. 9 is the only option which leads to a satisfying result.

The next section describes a heuristic search which solves this problem, using the greedy search as a fitness function.

Example 4: Incompleteness For this example, I am solving the predicate set $P = [\text{exists}(\langle \textit{value} \rangle), \text{exists}(\langle \textit{key} \rangle)]$ for the grammar in Figure 3.4. The algorithm starts by deriving ⟨*Start*⟩. As before, the first decision is whether ⟨*map*⟩ or ⟨*value*⟩ shall be expanded. This depends on which option has the smaller rating. It is $r'(\text{exists}(\langle \textit{value} \rangle), t, n, \langle \textit{value} \rangle) = 0$, and $r'(\text{exists}(\langle \textit{key} \rangle), t, n, \langle \textit{map} \rangle) = 4$. Likewise, $r'(\text{exists}(\langle \textit{value} \rangle), t, n, \langle \textit{map} \rangle) = 4$. So the algorithm decides for ⟨*value*⟩. There are no more decisions to be taken, and the final result is a tree with just 3 nodes, going from ⟨*Start*⟩ directly to ⟨*value*⟩. The predicate exists(⟨*key*⟩) is not fulfilled.

The next section describes a heuristic search which solves this problem, using the greedy search as a fitness function.

3.2.5 Searching the space of all possible trees

This section describes a search process which explores the space of all possible trees for a tree that fulfills a given set of predicates.

The main insight for this algorithm is that the decision sequences, as generated by `Greedy-Generate`, form a tree. Whenever something is added to the decision sequence, there are alternatives. Each decision becomes a node in the tree, with its alternatives as siblings. Each path in the obtained tree is a decision sequence, and each decision sequence corresponds to a parse tree.

To illustrate this, consider the decision sequences for obtaining `"sqrt(9)"` and `"tan(0)"` within Figure 2.1. Figure 3.7 illustrates how those can be viewed as part of the same tree.

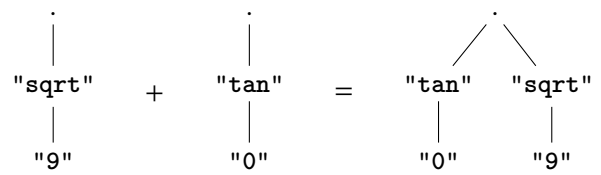


Figure 3.7: The decision sequences for obtaining `"sqrt(9)"` and `"tan(0)"` within Figure 2.1 illustrate how all decision sequences form a tree.

Within this decision tree, a heuristic tree search can be used to identify the parse tree, or decision sequence, which fulfills the predicate set.

In general, a heuristic tree search starts at the root node of a tree. It then generates all children of this node (this process is called expansion), and assigns a heuristic value to each child. Afterwards, the child with the lowest heuristic value is chosen, and expanded. In doing so, more nodes are created. In each step the node with the lowest heuristic value, among all known nodes, is expanded, until there is no unexpanded node left, or a solution is found. By always expanding the node with the lowest heuristic value (among all known nodes), the search systematically explores the entire tree. If heuristic values are low within an area of the tree that has a high likelihood to contain the goal of the search, it will be found sooner than with a depth-first or breadth-first search. This kind of heuristic tree search algorithms – or its generalization to graphs – are used in different fields of computer science: The famous A* algorithm for path search is a heuristic graph search[28], the FF$^+$[34] algorithm in automated planning is a heuristic graph search, and games like chess or Nine men's morris can be played with this kind of algorithm. The key to the application of this kind of algorithm is to describe the search space as a graph or tree structure, which I just did, and define a heuristic which allows me to decide where to look for a solution first.

Each internal node in the tree of decision sequences corresponds to the prefix of a decision sequence, only leaf nodes correspond to complete decision sequences. However, prefixes of decision sequences yield only partial parse trees, and no complete leaf words. The grammars I am working with are ambiguous, and all features are defined such that they consider all possible parse trees for a word: An existance feature is 0 if there is a parse tree which contains the corresponding control form, a maximal character length feature considers the maximal character length with respect to all parse trees and so on. This means that I can calculate the heuristic for words only, and not for partial parse trees. This is where the *inner search*, defined in Section 3.2.4 comes in: I use it to complete a partial parse tree. In order to do so, I define a rating function r_{seq} such that the algorithm follows the pre-existing decision sequence for all decisions that are part of the prefix. As soon as I reach the end of the prefix, I use r_P for the predicate set I am attempting to solve. Now, the inner search yields a complete parse tree, which I can use to calculate a heuristic that I can use in the outer search.

Since my goal is to fulfil all predicates within a given predicate set, I just count the number of predicates within the predicate set which are not fulfilled by a parse tree, and divide by the number of predicates in the predicate set. If this number is lower, the parse tree is more similar to a parse tree which fulfills

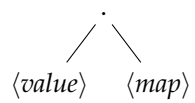


Figure 3.8: The already expanded parts of the tree of decision sequences after the first iteration of the outer search in the example.

all predicates. Therefore, the corresponding decision sequence is in an area of the tree which is likely to contain the correct solution.

In practice, this can be further refined: Not all predicates are necessarily binary decisions. For numeric predicates, I can use an idea similar to branch distance[48]. For a predicate $v_0 <$ max-char-length($\langle C \rangle$) $< v_1$, I can use a value

$$h = \begin{cases} 1 & \text{iff } v_0 < \text{max-char-length}(\langle C \rangle) < v_1 \\ \frac{v_0 - \text{max-char-length}(\langle C \rangle)}{v_0} & \text{iff max-char-length}(\langle C \rangle) < v_0 \\ \frac{\text{max-char-length}(\langle C \rangle) - v_1}{v_1} & \text{iff max-char-length}(\langle C \rangle) > v_1 \end{cases}$$

This definition ensures that the value is always between 0 and 1, and is 0 iff the predicate is fulfilled. If I sum up those values, and divide by the number of predicates, I get a value between 0 and 1, which is 0 if all predicates are fulfilled. However, trees which have a max-char-length($\langle C \rangle$) closer to the required range have lower values, therefore leading the search into the right branch of the tree.

Examples

In Section 3.2.4, I presented a case where the inner search cannot find a solution. The goal is to solve the predicate set $P = [\text{exists}(\langle \textit{value} \rangle), \text{exists}(\langle \textit{key} \rangle)]$ for the grammar in Figure 3.4. The algorithm initially knows nothing about the tree of decision sequences. So it invokes the inner search with r_P as its rating function. This invocation is identical to example 3 in Section 3.2.4. The inner search returns a tree with exactly 3 nodes. This tree fulfills exists($\langle \textit{value} \rangle$), but not exists($\langle \textit{key} \rangle$). So the heuristic value is 0.5. The decision sequence is rather simple. It has just one element: $\langle \textit{value} \rangle$. However, the tree of decision sequences, as shown in Figure 3.8, shows that there is another decision possible: The algorithm could choose $\langle \textit{map} \rangle$ first.

The outer search proceeds to start the inner search with a rating function r_1 such that

$$r_1(t, n, \langle c \rangle) = \begin{cases} r_{seq}(t, n, \langle c \rangle) & \text{for the first decision} \\ r_P(t, n, \langle c \rangle) & \text{else} \end{cases}$$

Here, r_{seq} is defined with respect to the decision sequence $\langle \textit{map} \rangle$.

Therefore, the first decision taken by the inner search is $\langle \textit{map} \rangle$. After that, the inner search has to decide between $\langle \textit{map} \rangle$ `","` $\langle \textit{pair} \rangle$ and `""`. At this point, r just returns r_P, and the first first option gets a value, as it contains a $\langle \textit{pair} \rangle$, and there are paths from $\langle \textit{pair} \rangle$ to both $\langle \textit{value} \rangle$ and $\langle \textit{key} \rangle$. All in all, this execution of the inner search is very similar (but not identical), to the invocation in Example 1 in Section 3.2.4. It yields a tree which contains a pair, and fulfills both predicates. The outer search finds a solution by invoking the inner search twice.

By Catch

Within ALHAZEN, each predicate set corresponds to a path in the decision tree, and each predicate corresponds to a node on a path. However, nodes appear in multiple paths, and therefore predicates appear in multiple predicate sets. This leads to a situation where all predicate sets that are derived within one iteration of ALHAZEN are quite similar.

If a decision tree yields the predicate set $P_1 = [\text{exists}(\langle value \rangle), \text{exists}(\langle key \rangle)]$, it likely also yields the predicate set $P_2 = [\text{exists}(\langle value \rangle), \neg\text{exists}(\langle key \rangle)]$. This is just the other branch in the node which has the predicate $\text{exists}(\langle key \rangle)$. But when I attempted a solution for P_1 within the example in Section 3.2.5, the first invocation of the inner search gave a sample which fulfills P_2. It is no longer necessary to invoke the search for this predicate set, as the solution was discovered accidentally. This happens so frequently, that I began to call this *by catch* of the outer search. Therefore, it is beneficial to check each intermediate result against all other predicate sets that are to be solved in the same iteration of ALHAZEN's feedback loop. In some cases, all predicate sets can be solved by a single invocation of the outer search.

This is especially true for the mechanism which avoids coincidental correlation: It generates predicate sets P' such that there is a predicate set P in the same iteration of ALHAZEN's feedback loop such that P is a subset of P'. But then, a solution for P' is a solution for P just as well.

When the generator algorithm is invoked with more than one predicate set, it will attempt to solve those sets one after another, but report solutions that are discovered as by catch. The search will not be started for a predicate set if a solution is known already, due to by catch.

3.3 Evaluation

In this chapter, I presented ALHAZEN, which added a feedback loop to ALHAZEN0. Therefore, the objective of this evaluation is compare the performance of ALHAZEN0, with the improved ALHAZEN. ALHAZEN0 was presented and evaluated in Chapter 2, and, as ALHAZEN shares the same objective and use cases, I am going to use the same evaluation scheme. Results are thus directly comparable. The evaluation scheme was presented in Section 2.5.

3.3.1 As Predictor

This section evaluates whether ALHAZEN can be used to determine whether an input triggers the behavior of interest, without actually running the program under test with this input. As in Section 2.7.1, I'll look at precision and accuracy as the relevant metrics. The results were determined with the same methodology as for Section 2.7.1.

ALHAZEN received the same training samples as ALHAZEN0, but generated additional samples in its feedback loop. Figure 3.9 shows the number of samples initially provided, and additionally created.

For the twoInputs configuration, where ALHAZEN gets just two inputs to learn from, it does not generate as many samples as were generated for the sets

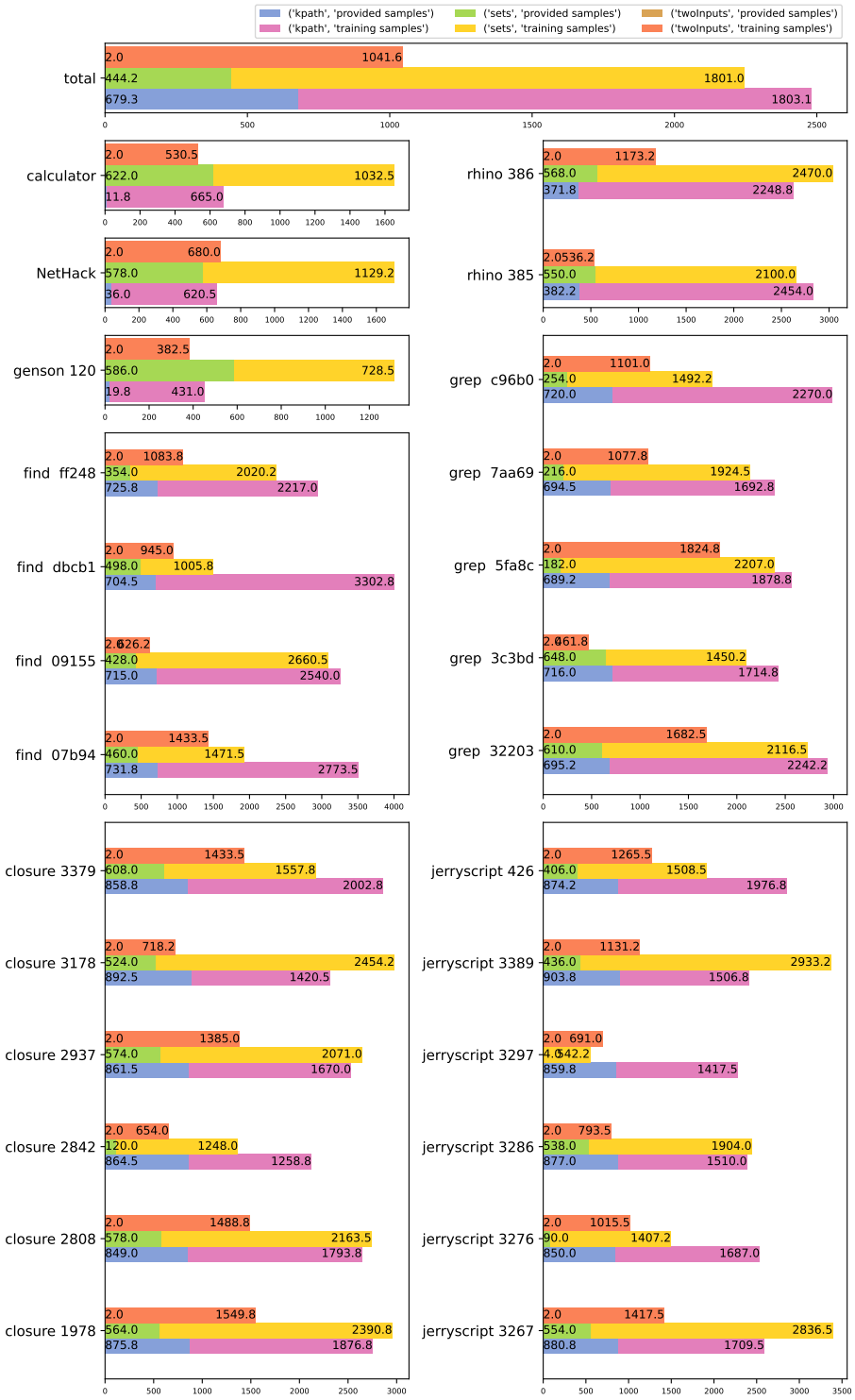


Figure 3.9: Number of training samples for ALHAZEN as a predictor, on all subjects and all configurations.

configuration in 23 out of 26 cases. ALHAZEN achieves comparable precision and accuracy, with a lower number of samples.

> The samples generated for the twoInputs configuration are more informative than those generated in Section 2.5.4.

With this observation, let's look at accuracy in Figure 3.10, and precision in Figure 3.11.

The twoInputs configuration still archives better precision than the sets configuration on 6 of the 26 subjects, and better accuracy on 2 subjects. Overall, the precision of the twoInputs configuration is 90.8%, and the precision of the sets configuration is 94.8%. This difference is, however, not statistically significant (tested with a wilcoxon paired-sample test with $p < 0.05$). I can therefore conclude that, with respect to precision, ALHAZEN trained on just two inputs with the feedback loop is as good as ALHAZEN with its feedback loop trained on the sets dataset. As observed before, the number of samples does not coincide with the quality of the classifier directly. Also keep in mind that the bias for the sets configuration still exists: The twoInputs configuration was good enough to offset the bias.

The accuracy data looks slightly different: The overall accuracy of sets is 96.2%, and twoInputs achieves 88.6%. While not a huge difference, this is statistically significant.

> In terms of precision, ALHAZEN trained on just two inputs with the feedback loop is as good as ALHAZEN with its feedback loop trained on the sets dataset. Accuracy is slightly higher for the sets configuration.

The k-path configuration achieves better precision than sets in 5 subjects. The hypothesis test shows the results to be statistically significantly different from those obtained with the sets configuration, but similar to the ones obtained with twoInputs.

In conclusion, the feedback loop has the effect of a larger initial input set when it is used with a small input set, but does not help as much with a large initial input set.

> Using the feedback loop, differences between different starting conditions are negligible.

The choice of a configuration should therefore be based on other factors: If, a huge number of samples exists already, there is no reason why this data should not be used, and ALHAZEN0 may indeed be sufficient. On the other hand, the twoInputs configuration has the lowest number of samples, and therefore needs the smallest number of invocations of the program under test in the training phase. If program executions are expensive, it is the best choice.

> Learning from just two inputs, ALHAZEN achieves a precision of 90.8% and an accuracy of 88.6% over all subjects.

This outperforms the precision I reported in Section 2.7.1.

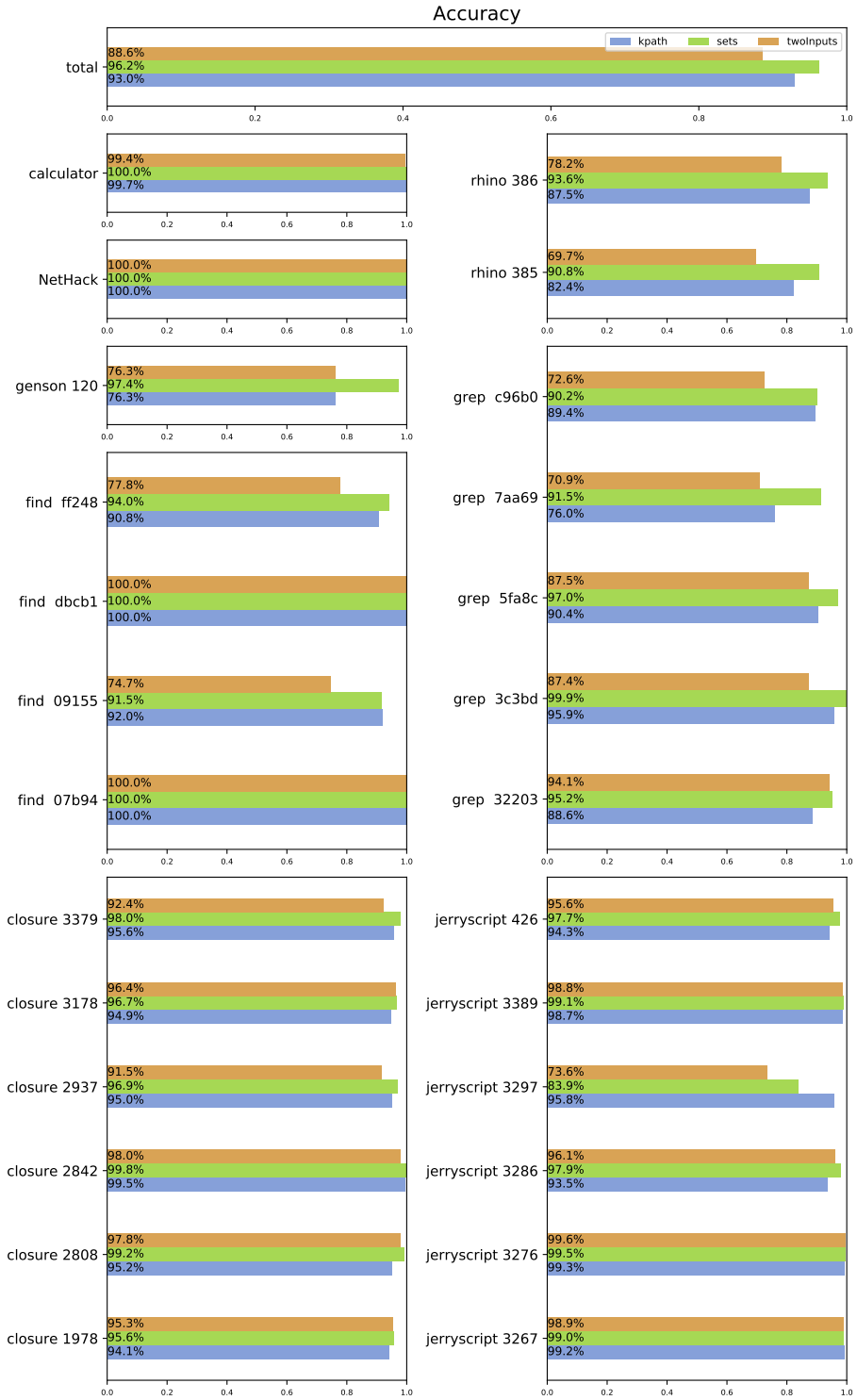


Figure 3.10: Accuracy for ALHAZEN as a predictor, on all subjects and all configurations.

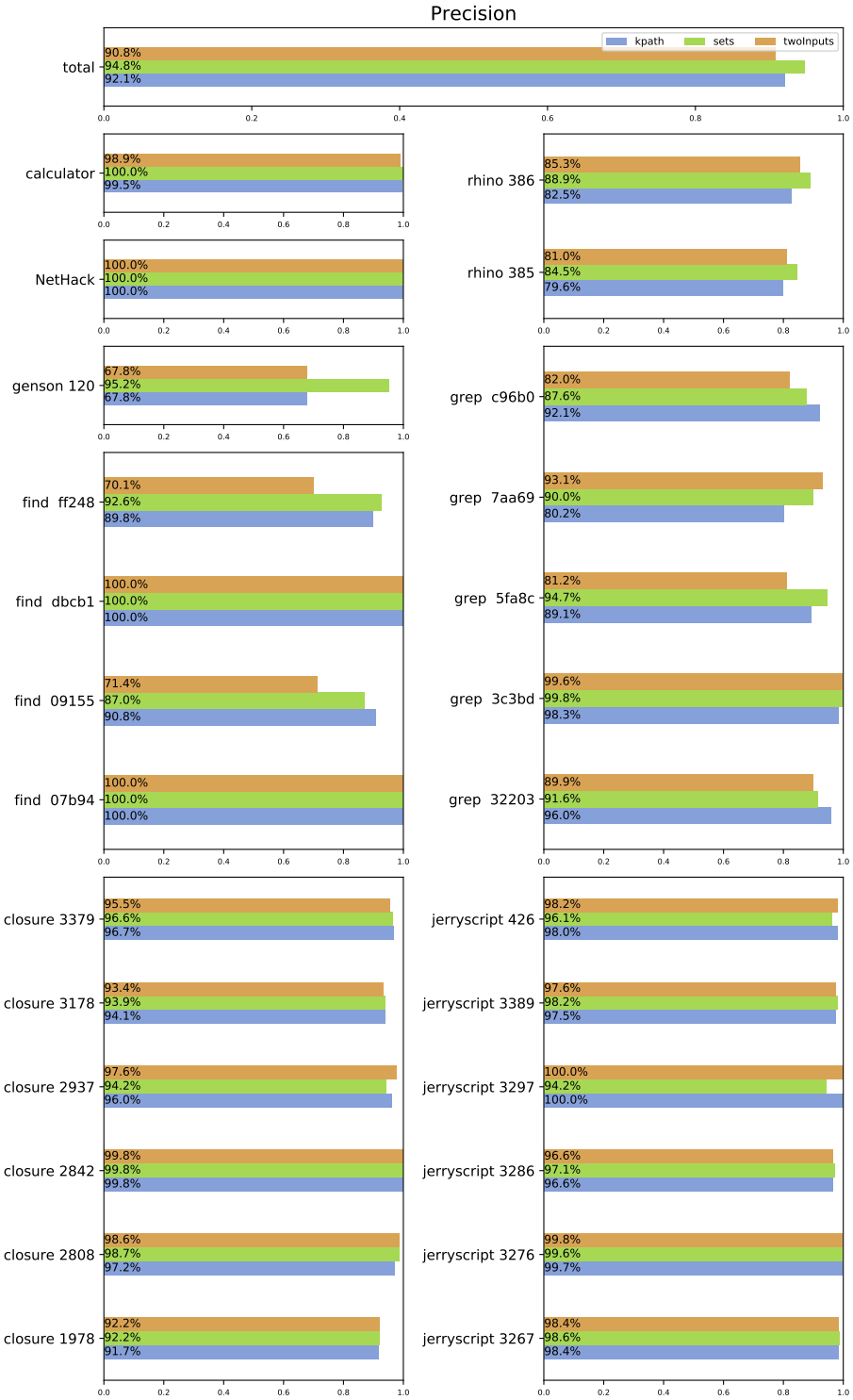


Figure 3.11: Precision for ALHAZEN as a predictor, on all subjects and all configurations.

For reference

The results for the sets configuration cannot be trusted, as ALHAZEN0 learned properties of the generator algorithm, rather than properties of the behavior of interest.

See Section 2.7.3

For reference

ALHAZEN0 achieves an accuracy of 84.9% and a precision of 82.0% on average for all subjects, using the k-path training sets.

See Section 2.7.1

3.3.2 As Generator

When I evaluated ALHAZEN0 as a generator in Section 2.7.2, the problems with the approach became evident: The generator use case seems to be much more affected by coincidental or structural correlations. So, it will be interesting to see how the feedback loop improves the situation.

Let's look at the number of generated samples first. This data can be found in Figure 3.12.

All configurations generated more samples than for ALHAZEN0 in total.

Let's look at the data for precision in Figure 3.14, and accuracy in Figure 3.13.

The precision overall, which was 22.8% for ALHAZEN0, improved to 65.2% for the twoInputs configuration. The improvement is statistically significant. This proves that the feedback loop indeed helps to improve the trees, and eliminate coincidental correlations. The accurracy is still higher than the precision (89.2% for the twoInputs configuration), indicating that it is still easier to generate non-failing than failing samples.

89.2% of the samples generated trigger or don't trigger the behavior as requested. 65.2% of the samples generated trigger the behavior, if requested.

In Chapter 2, I explained that there is a bias in the sets configuration. This bias is still present, so twoInputs is at the disadvantage. Nevertheless, the twoInputs configuration outperforms the sets configuration on both metrics. However, as for the predictor evaluation, the differences between the three configurations are not statistically significant. I can confirm that the feedback loop equalizes differences in input data.

3.3.3 Analysis of Individual Cases

In this section, I describe some of the generated trees. Within Section 2.7.3, this was the step that made shortcomings of ALHAZEN0 obvious, so I'll look at the same examples again, and check whether the situation improved.

In Section 2.7.3, I analyzed trees obtained in the sets configuration, and one of the main observations was a bias with this configuration. In this section, the improved precision and accuracy for ALHAZEN in the twoInputs configuration

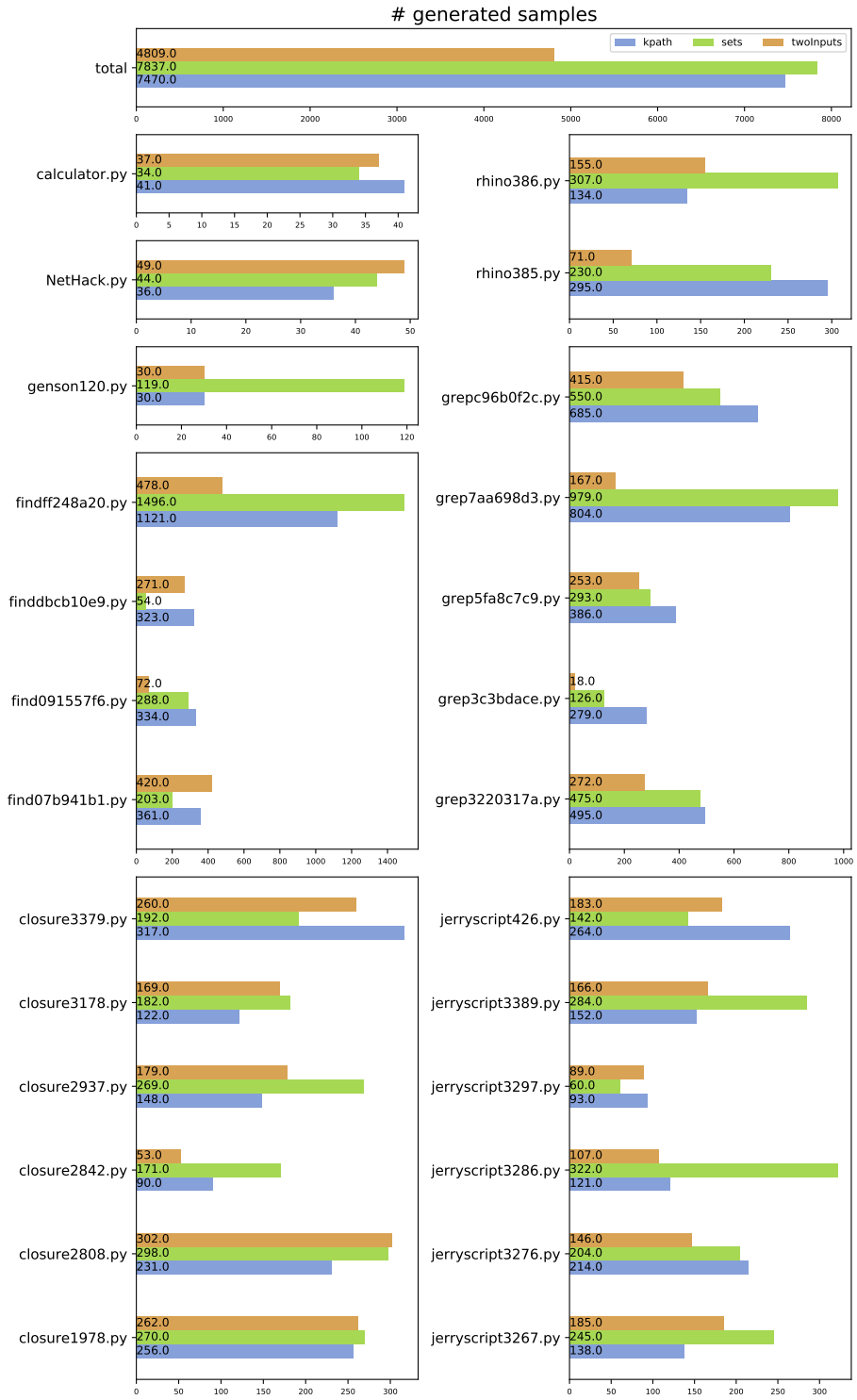


Figure 3.12: Number of samples generated by ALHAZEN, on all subjects and all configurations.

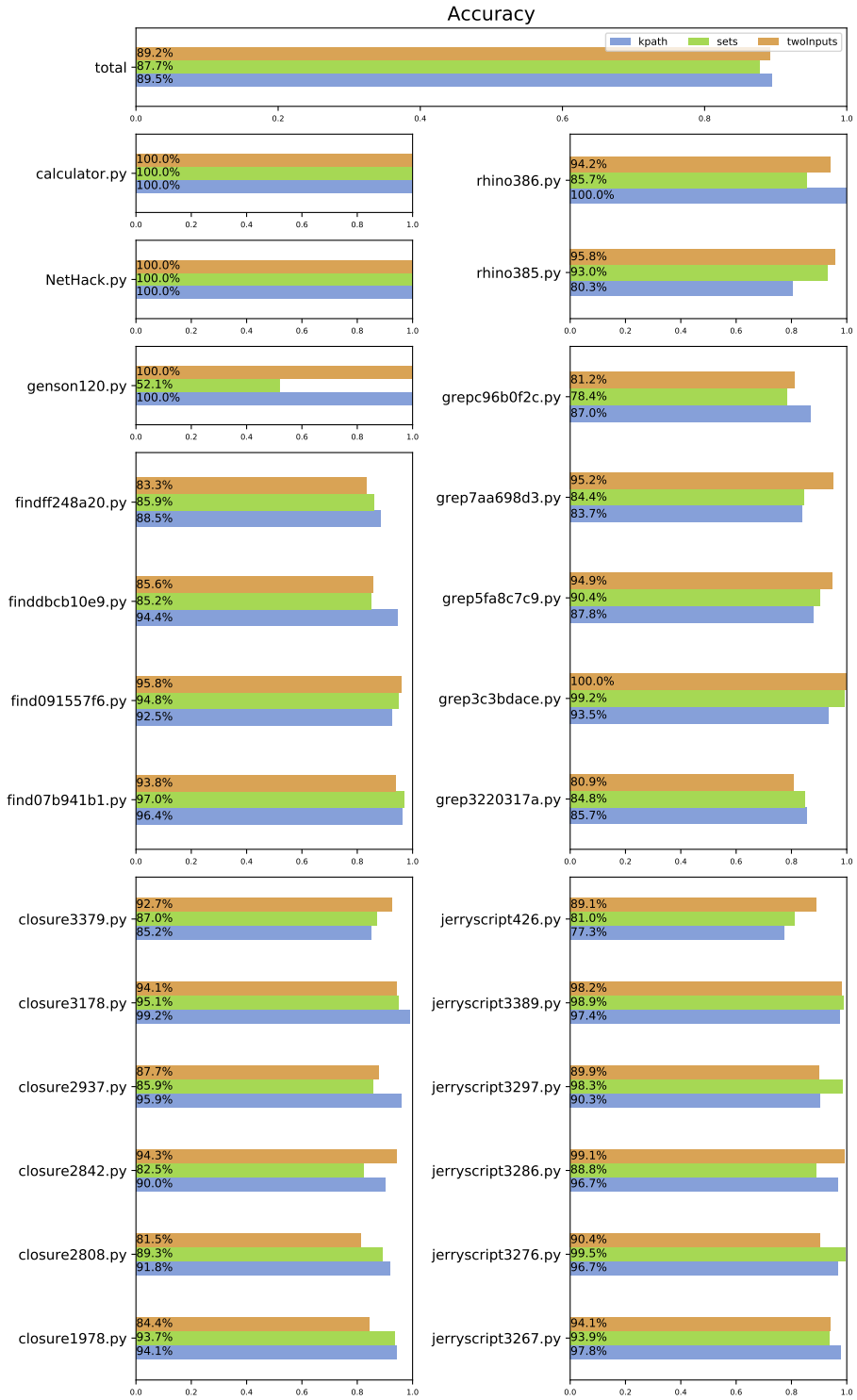


Figure 3.13: Accuracy for ALHAZEN as a generator, on all subjects and all configurations.

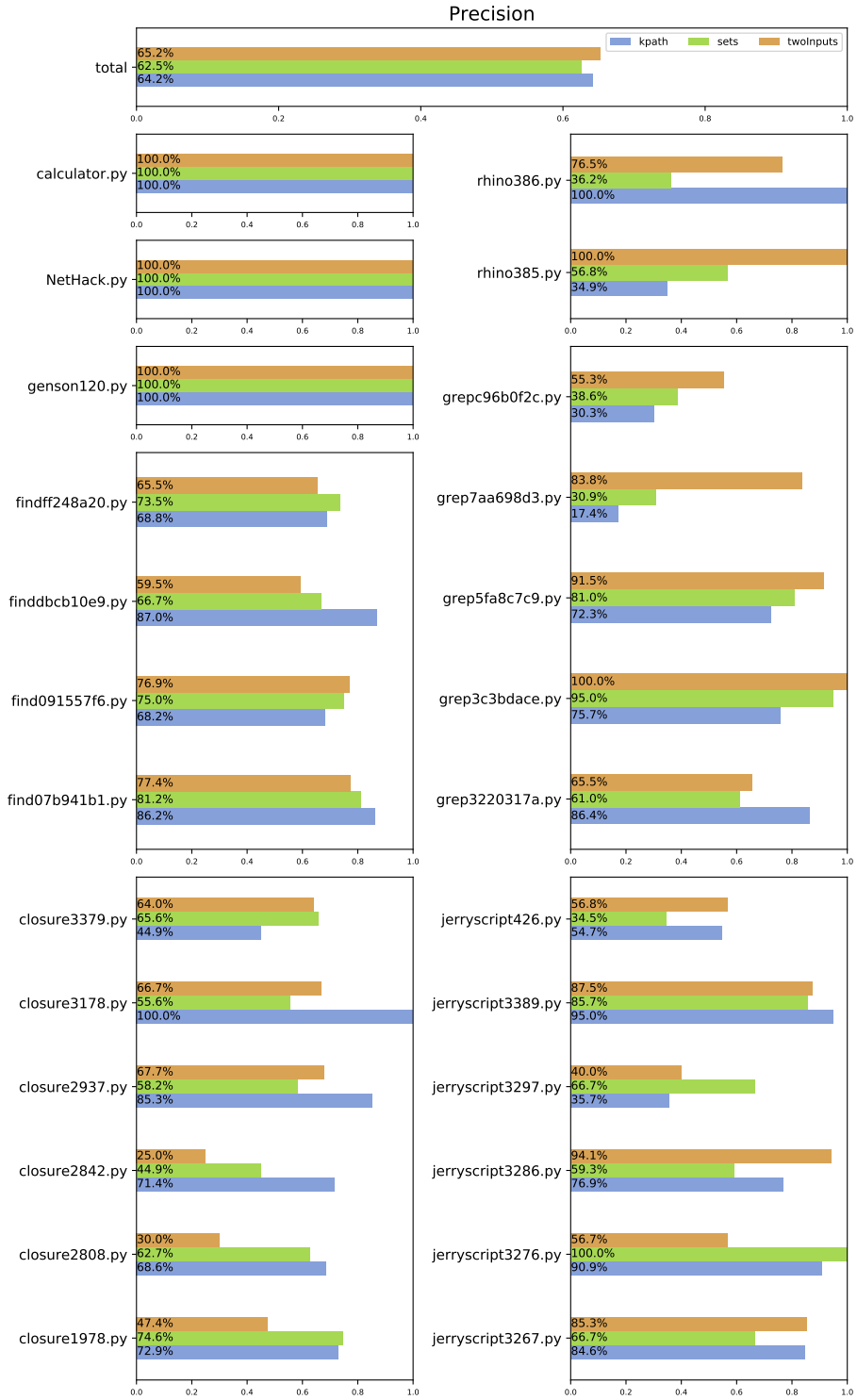


Figure 3.14: Precision for ALHAZEN as a generator, on all subjects and all configurations.

For reference

Using the feedback loop, differences between different starting conditions are negligible.

See Section 3.3.1

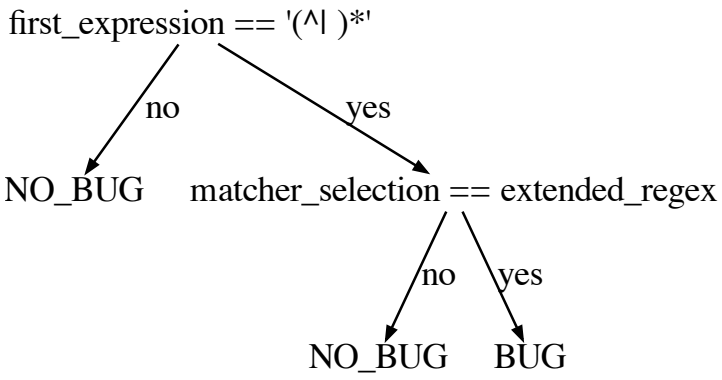


Figure 3.15: The decision tree for Grep 3c3bdace with ALHAZEN and the twoInputs configuration.

mean that I can analyze trees obtained in this configuration, which do not suffer from the bias.

Grep 3c3bdace

Bug 3c3bdace in `grep` occurs if grep is invoked with the regex `"(^| )*"` and the extended regex option. In Section 2.7.3, I observed that ALHAZEN0 gave a diagnosis which even included some information that was not included within the original bug report. Despite this success, ALHAZEN0 did not identify the regex correctly.

The tree obtained with ALHAZEN can be found in Figure 3.15. In contrast to the tree obtained by ALHAZEN0(presented in Figure 2.21), it identifies the required regex correctly. However, it does not contain the information that activating the word match option hides the bug. This was discovered by ALHAZEN0.

In this experiment, I learned from the twoInputs configuration. None of the seed inputs for this bug contains the word match option, therefore the decision tree never uses this option in any decision, and thus it is never included in any of the generated samples. But then, the decision tree cannot know that it hides the bug.

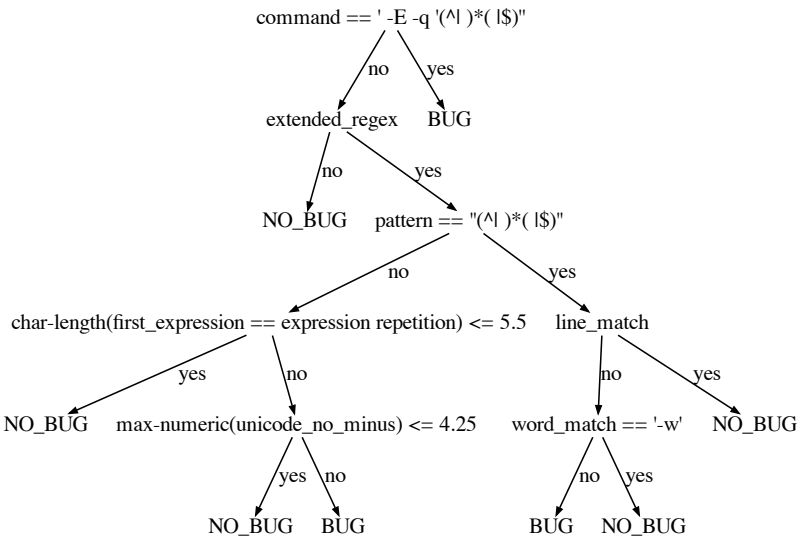


Figure 3.16: The decision tree for Grep 3c3bdace with ALHAZEN and the kPath configuration.

> With the feedback loop, ALHAZEN concentrates on a subspace of the input space, and may miss effects in other subspaces.

Randomly generated samples are distributed among the entire search space, so combining the feedback loop with some randomly generated samples may be beneficial. In Figure 3.16, the tree for the kPath configuration is shown.

This tree is a lot bigger than the tree from the twoInputs configuration. But does it also contain more information? This tree starts by pointing out that using the original sample always fails. Obviously, that is correct, but not helpful. Afterwards, it reports that the extended regex option is required (correct), and looks at the pattern used. The tree claims that `"(^| )*( |$)"` is the required regex, which is wrong. The first part, `"(^| )*"`, is sufficient already.

Going further down on the right hand side, the tree indicates that using the line match and word match options hides the bug, as ALHAZEN0 did. Therefore, more randomized data actually brings this additional information back.

Looking at the left hand side of the pattern node, the tree talks about the length of ⟨*expression*⟩ ⟨*repetition*⟩. This production rule would be used to parse `"(^| )*"`. So there is some hint at the correct pattern, even if it is unlikely that a software developer would notice it. The last node talks about a max-numeric() for ⟨*unicode_no_minus*⟩: The training data by accident contained non-behavior triggering samples with a small numeric value for this non-terminal only.

ALHAZEN does not examine parts of the search space that are not in the seed samples, so more randomized data, provided together with the seed samples,

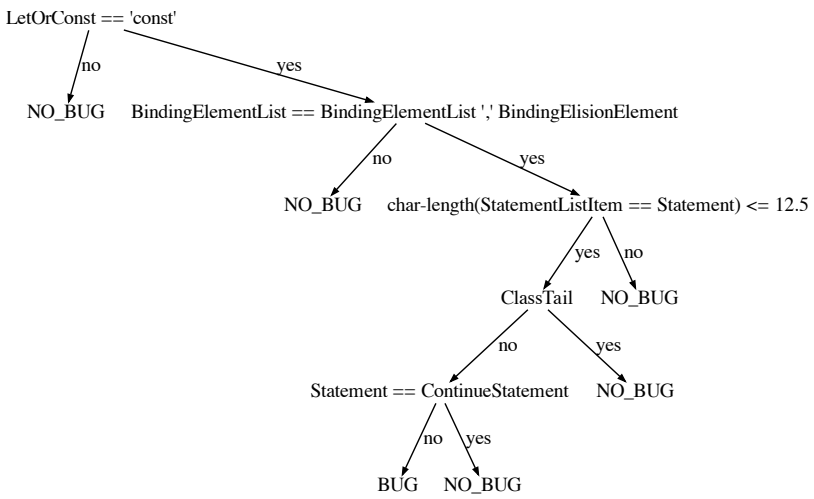


Figure 3.17: The decision tree for Rhino 386.

helps Alhazen to analyse the entire search space, and discover more information. But then, it also means that there is more random correlations which may distract Alhazen. While I did not run this experiment, it stands to reason that more time would help Alhazen to clarify or eradicate wrong beliefs within the tree.

Rhino 386

Bug 386 in Rhino occurs if two variables have the same name within a destructuring assignment. As I already explained in Section 2.7.3, neither Alhazen0 nor Alhazen can express this: My features do not allow me to reason that two parts of the input, in this example variable names, need to be the same.

Within Section 2.7.3, this example showed how Alhazen0 was biased by the training data: The tree expressed that samples of a certain length were very likely to trigger the bug. Figure 3.17 shows the decision tree obtained by Alhazen.

In the root node, Alhazen correctly reports that the `"const"` keyword needs to be used. This keyword starts a destructuring assignment. In the second node, Alhazen found that the ⟨*BindingElementList*⟩ needs to be at least 2 elements long. If it were shorter, there could not be two elements with the same name, so this is true just as well. Afterwards Alhazen restricts the length of a ⟨*statement*⟩. This restriction leads to shorter variable names, which increases the probability of two variables with the same name. Then, Alhazen excludes a ⟨*ClassTail*⟩, which, as in Section 2.7.3 would be a way to get a sufficiently long statement without using the required expressions. In the last node, Alhazen rules out a `"continue"`. `"continue"` is invalid in all contexts except for a loop, and violating this means that Rhino will never execute the code, and the bug will never be triggered. So this just rules out one factor which makes Rhino fail earlier than the bug I am

For reference

Definition 0: Predicate of Interest
The *predicate of interest* is a predicate over observable program behavior. The program behavior which is recognized by the predicate of interest is the *behavior of interest*. If the program exhibits the behavior of interest while processing a specific input, this input is said to be *behavior-triggering*.

See Section 1.4

looking for.

All in all, the tree does contain some distracting nodes, that a user of Alhazen may not be able to interpret. However, given that this bug exceeds Alhazen's abilities, it is remarkable that the two top-most nodes are correct. A software developer who understands that they have to ignore the remaining nodes may even profit from looking at this tree.

Alhazen versus Imprecise Oracles

So far, I assumed that the predicates of interest given to Alhazen are *perfect* and *pure*. Here, perfect means that all program runs which exhibit the bug are recognized as such, while pure means that only program runs which exhibit the bug are recognized as such. However, writing predicates of interest can be challenging. Therefore, I will examine what happens with an unpure predicate of interest in this section.

Bug 3c3bdace in grep is going to serve as an example once more. Alhazen's results for this bug were already discussed in Section 2.7.3 and Section 3.3.3. The bug is a segmentation fault which occurs if grep is invoked with the regex `"(^| )*"`, and the extended regex option.

For this section, I ran Alhazen with a grammar which would allow for invalid inputs. As described in Section 2.5.2, the grammar for grep was set up to generate full shell commands like

```
1    printf 'X' | timeout 0.5s grep -E -q '[a-z]'
```

However, the grammar is not perfect. It can also generate inputs like this:

```
1    printf 'X"'' | timeout 0.5s grep -E -q '(^| )*( |$)'
```

Within this input, the argument to `printf` is not properly escaped: The quotation marks are unbalanced. This means that the shell, upon execution of this command, will terminate with a syntax error, rather than invoking grep.

For the experiments, the predicate of interest was set up to recognize this problem, and Alhazen ignored such cases. Within this section, I modified the predicate of interest to judge those cases as not bug-triggering. As a result, Alhazen generated the tree in Figure 3.18.

The first observation is that Alhazen uses almost the entire seed input in the root node. This means that it recognizes samples which are mostly identical to the seed input. But the remainder of the tree is interesting nevertheless. The substring used in the root node contains the `"-q"` option, because it was used in the original bug report. On the left side, Alhazen describes that in fact the ⟨*extended_regex*⟩ option and the regex `"(^| )*"` are sufficient. This part

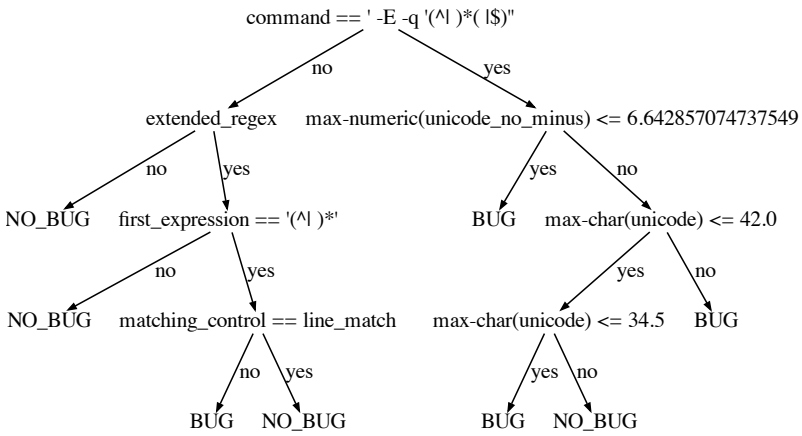


Figure 3.18: The decision tree for Grep 3c3bdace with an imprecise predicate of interest.

of the tree also indicates that the ⟨*line_match*⟩ option means that the bug does not appear any longer. Basically, the left side of the tree is identical to what ALHAZEN0 gave.

On the right hand side, there is a lot more going on. One would assume that all inputs which contain the original behavior-triggering sample trigger the bug. But ALHAZEN tells us that this is not the case: The bug does not occur if specific characters are in the argument to `printf`. Those characters happen to be quotation marks. Therefore, this part of the tree points at the problem with the grammar.

Software developers who employ ALHAZEN should be aware of this kind of effects. When interpreting the trees, they should examine one sample for each leaf of the tree, and check whether this sample points to a shortcoming with the predicate of interest or the grammar (or any other part of the setup, for that matter). However, as it is quite difficult to write such predicates and grammars, I assume that the grammar would be written once, and used to diagnose multiple bugs. The cost for fixing problems in the grammar would be high for the first couple of bugs analyzed, but quickly decrease as ALHAZEN and the grammar are used more often.

If none of the general-purpose predicates of interest can be used, one predicates of interest needs to be written for each bug. However, developers would quickly build up expertise in how to design those. Also, predicates of interest are usually just a few lines of code. Even the problem in the predicate of interest in this section, recognizing a syntax error as a non-behavior triggering input rather than dismissing the sample, is a problem with the test scaffolding, rather than the predicate of interest.

3.4 Conclusion

Within this chapter, ALHAZEN0 was extended with a feedback loop. Using ALHAZEN instead of ALHAZEN0, two inputs, a behavior-triggering one and a non-behavior triggering one, are sufficient to obtain a hypothesis with good results for precision and accuracy. This means that, in a practical application of ALHAZEN, it is no longer necessary to generate a huge set of input data. ALHAZEN can be run with minimal initial input data, and generates the required training data itself. Users should be aware, however, that the initially supplied samples also define which part of the input space is examined by ALHAZEN. Therefore, a low number of initial samples may compromise ALHAZEN's ability to uncover additional information about the behavior of interest.

In this configuration, ALHAZEN can be used to predict if a given input triggers a bug or not. As an example, it would be possible to filter inputs from an external source before feeding them into the program, and prevent a bug triggering input, which, in the context of a system reachable remotely, may be an attempt to perform a cyberattack, before it even reaches the program. Given that ALHAZEN cannot provide a guarantee, this filter should only be used as long as the bug is not yet properly fixed.

Also, ALHAZEN can be used to generate more behavior-triggering inputs. Those inputs may be used as regression tests, within automated repair approaches or for statistical debugging.

This chapter did not attempt to provide an in-depth analysis of whether ALHAZEN is useful as a debugging aid. This evaluation will be performed in Chapter 4. Still, the analysis of individual cases in Section 3.3.3 provides some insights. Effects within the training data, imprecise predicates or problems with the grammar and test scaffolding can lead to trees that are hard to interpret. In most cases, it is possible to build a deeper understanding of the underlying problem when one looking at the test cases. Still, a developer who wants to use ALHAZEN needs a good understanding of how ALHAZEN works, in order to be able to recognize and interpret this kind of effects.

Chapter 4

Debugging with Input Features

Chapter 2 and Chapter 3 presented ALHAZEN, a tool which generates a bug diagnosis automatically. They did not, however, evaluate whether these diagnoses are useful for software developers.

Previous attempts to help developers with debugging[78, 50], which will be discussed in Section 6.1, tried to minimize the context a developer needs to analyze. Intuitively, with a good idea where to look, the developer can concentrate on a few lines of code, rather than search the entire program for a flaw. This also applies to input minimization techniques: A smaller input often means a faster program run, which simplifies the analysis of program states, simply because there is a smaller number of (intermediate) program states.

The same argument can be used for ALHAZEN. The tool identifies those control forms in the grammar, and potentially even properties of the associated substrings of the input, that are connected to the bug. In Section 4.1,I will analyze which fraction of the program input space is deemed irrelevant by ALHAZEN.

However, this is not the full story of fixing a bug: When a software developer attempts to fix a bug, it is important that they understand the root cause of the bug. If the developer attempts a fix without deep understanding of the problem, the fix rarely meets quality standards. Common problems are:

Treating Symptoms A bug fix which treats a symptom only hides the consequences of a bug, rather than fixing it.

Incomplete Fixes An incomplete fix solves one instance of the problem, but other inputs which trigger the same problem still do so.

Regression Bugs A regression bug is a bug which is introduced during a fix. While the fix may solve the original problem, it introduces a new problem at the same time.

When considering the usefulness of ALHAZEN in the context of debugging, a natural question is whether ALHAZEN's diagnosis helps in identifying this root cause. This question can only be answered in a user study. In the remainder of this chapter, I will discuss the design of such a user study, and perform two pilot studies to refine the proposed design. The general design was developed within

Sven Fackert's master thesis, which I advised. I then refined the design based on his findings, and conducted a larger pilot study, which uncovered further design problems. In this chapter, I use 'we' to report about findings and actions I performed together with Sven Fackert, and 'I' for work that I performed myself.

This chapter concludes by describing an improved design, which can be used to conduct a user study in future work.

4.1 Focusing on Small Parts of the Input Space

As described in the introduction, previous attempts at automated debugging focused on minimizing the search space for the software developer. In this section, I will apply the same idea to Alhazen. My research question for this section is thus:

1. Which part of the program input space is part of Alhazen's diagnosis?

This is relevant, as developers can ignore every part of the program input space which is not part of the diagnosis. Alhazen deemed it irrelevant.

Obviously, this evaluation does not really indicate that Alhazen is useful in debugging. Many more factors are relevant. As an example, the decision trees may be hard to interpret, or the relationship between input structure and inner workings of the program may be too complex for the information provided by Alhazen to be useful. However, if one accepts the assumption that debugging is easier if the program input space that needs to be considered is smaller, this evaluation gives a first indication of usefulness.

4.1.1 Characterizing the Search Space

If I want to show that Alhazen reduces the search space, I need to be able to measure the size of this search space. Previous approaches to minimize the search space in debugging actually minimized an existing input. They can be evaluated by reporting the number of characters that are not in the input any more. Alhazen uses a grammar to describe which parts of the input structure are important. Therefore, counting characters would not do my approach justice. Even if Alhazen is used as a generator, it generates more than one test. So, the length of which of those tests should be considered?

Instead, I will characterize the search space in terms of the features: A large grammar belongs to a program with very diverse inputs, and lots of variation in possible program runs. Such a large grammar also uses lots of features. Therefore, I will use the number of features as an approximation of search space size.

Table 4.1 shows the size of the grammars for all subjects. JerryScript and Closure both accept JavaScript as inputs, still, I list sizes per subject, and not per grammar. The reason is that even subjects with the same grammar may have different numbers of features. This is caused by the rewrite in Section 2.4, which adds additional alternatives, and therefore enlarges the grammar. In generel, the grammars are quite different:

Ignoring the calculator example (17 features), NetHack (62 features) and genson 120 (63 features) are smallest. The largest grammar is the one for Closure 3178, which has 1939 features.

subject	total # of features	# of max-char()	# of max-char-length()	# of max-qu-length()	# of max-numeric()	# of exists()
NetHack	62	2	9	1	1	49
calculator	17	2	3	0	1	11
closure 1978	1919	218	614	4	25	1058
closure 2808	1881	218	612	6	25	1020
closure 2842	1899	218	612	6	25	1038
closure 2937	1890	218	612	5	26	1029
closure 3178	1939	218	613	5	25	1078
closure 3379	1886	218	612	6	25	1025
find 07b941b1	1172	0	100	0	8	1064
find 091557f6	1179	0	100	0	8	1071
find dbcb10e9	1168	0	100	0	8	1060
find ff248a20	1178	0	100	0	8	1070
genson 120	63	5	16	1	3	38
grep 3220317a	1083	2	84	0	8	989
grep 3c3bdace	1085	2	83	1	8	991
grep 5fa8c7c9	1083	2	84	0	8	989
grep 7aa698d3	1085	2	84	0	8	991
grep c96b0f2c	1087	2	84	0	8	993
jerryscript 3267	1914	218	612	6	25	1053
jerryscript 3276	1881	218	613	5	26	1019
jerryscript 3286	1915	218	613	5	25	1054
jerryscript 3297	1897	218	614	4	25	1036
jerryscript 3389	1971	218	614	4	25	1110
jerryscript 426	1895	218	612	6	25	1034
rhino 385	1197	9	370	8	23	787
rhino 386	1192	9	370	8	23	782

Table 4.1: Number of features in the grammars used

$\langle A \rangle \rightarrow \langle B \rangle +$ is the same as $\langle A \rangle \rightarrow \langle B \rangle \mid \langle B \rangle \langle A \rangle$

$\langle A \rangle \rightarrow \langle B \rangle *$ is the same as $\langle A \rangle \rightarrow$ "" $\mid \langle B \rangle \langle A \rangle$

$\langle A \rangle \rightarrow \langle B \rangle ?$ is the same as $\langle A \rangle \rightarrow$ "" $\mid \langle B \rangle$

Figure 4.1: Alternate ways to write quantifications.

Grammars of the same size can differ in the kind of features. While exists() are prevalent for all grammars, there is a difference nevertheless. For the JavaScript grammars on Closure, about 11% of the features are max-char(). In contrast, the `find` grammar has no max-char() at all. This indicates that there are no productions which derive an infinite number of words (see how I remove structural correlations in Section 2.6.2). `find` is a tool which can be used to search for files within the file system. Intuitively, one would expect that file names, supplied to `find` as search terms, are unrestricted within the `find` grammar. However, the grammar for `find` generates not only `find` invocations, but also a script which generates some directory structure to search in. In order to make sure that the file names `find` searches actually exist within this structure, the grammar restricts the allowed file names – and therefore, search terms – to a predefined set. But then, the question which file name was choosen is an exists(), and there are no max-char().

Another observation is that max-qu-length() are seldom among all grammars. It seems that grammar authors avoid quantifications. A quantification can always be written with a recursive rule, as Figure 4.1 illustrates. Therefore, there is no language which requires quantification within its grammar. For Closure, JerryScript, genson and Rhino, I got my grammars from Havrikov and Zeller [31], who translated them from the ANTLR grammar repository[54], or the JavaScript> specification[36] respectively. The formalism used in the JavaScript specification does not allow for quantifications. The specification therefore uses an alternative way to write such structures everywhere. This may have prompted the translator not to use quantifications either. For `find` and `grep`, there are simply not that many elements which can be repeated.

4.1.2 Evaluation

In the last section, the search space was characterized in terms of features derived from the grammar. But then, the number of features used in the decision tree is the portion of the search space which a user of Alhazen can concentrate on. In this section, I will look at how large that portion of the search space.

Table 4.2 gives data on shape and features used in the decision trees. All data is presented as an average over four runs in the twoInputs configurations. I got this data from the same runs I used in Section 3.3.3.

First, let's have a look at the size of the trees. The first column in Table 4.2 gives the average number of nodes, the second column gives the average number of leaves. The relationship of the size of the grammar to the size of the tree is not clear. Trees for the `JavaScript` grammar, which has the highest number of features with about 2000, have between 10 and 20 nodes. However, the `grep` and `find` bugs, which have a grammar with about 1000 features, have between 20 and 30 nodes. Within this experiment, smaller grammars generate larger trees.

subject	average # of tree nodes	average # of tree leaves	average # of features	average # of max-char()	average # of max-char-length()	average # of max-qu-length()	average # of max-numeric()	average # of exists()
NetHack	4.0	2.5	1.25	0.0	1.25	0.0	0.0	0.0
calculator	5.0	3.0	2.0	0.0	0.0	0.0	1.0	1.0
closure 1978	19.5	10.25	9.0	1.0	2.0	0.0	0.0	6.0
closure 2808	17.5	9.25	7.75	0.0	3.0	0.25	0.0	4.5
closure 2842	13.5	7.25	6.25	0.5	3.5	0.0	0.0	2.25
closure 2937	17.0	9.0	7.75	1.25	3.0	0.0	0.0	3.5
closure 3178	11.5	6.25	5.25	0.5	0.75	0.0	0.25	3.75
closure 3379	16.0	8.5	7.5	0.25	1.5	0.0	0.0	5.75
find 07b941b1	21.0	11.0	7.5	0.0	2.25	0.0	0.0	5.25
find 091557f6	5.5	3.25	2.25	0.0	1.5	0.0	0.0	0.75
find dbcb10e9	16.5	8.75	7.25	0.0	1.5	0.0	0.25	5.5
find ff248a20	20.0	10.5	7.5	0.0	2.25	0.0	0.0	5.25
genson 120	3.0	2.0	1.0	0.0	0.0	0.0	0.0	1.0
grep 3220317a	20.0	10.5	9.5	0.0	0.75	0.0	0.25	8.5
grep 3c3bdace	4.0	2.5	1.5	0.0	0.25	0.0	0.0	1.25
grep 5fa8c7c9	18.5	9.75	7.0	1.75	1.25	0.0	0.0	4.0
grep 7aa698d3	17.5	9.25	7.75	0.25	1.25	0.0	0.25	6.0
grep c96b0f2c	30.0	15.5	11.25	1.5	2.0	0.0	0.5	7.25
jerryscript 3267	12.5	6.75	5.75	0.0	1.0	0.0	0.0	4.75
jerryscript 3276	11.0	6.0	5.0	0.0	2.25	0.0	0.0	2.75
jerryscript 3286	9.5	5.25	4.25	0.25	0.25	0.0	0.0	3.75
jerryscript 3297	14.5	7.75	6.75	0.5	3.0	0.0	0.0	3.25
jerryscript 3389	10.5	5.75	4.75	0.0	0.75	0.0	0.0	4.0
jerryscript 426	13.0	7.0	6.0	0.75	1.25	0.0	0.0	4.0
rhino 385	9.5	5.25	4.25	0.0	2.0	0.5	0.0	1.75
rhino 386	13.0	7.0	6.0	0.0	2.5	0.0	0.0	3.5

Table 4.2: Average size of the decision trees generated by ALHAZEN in the twoInputs configuration, together with the average number of features used.

There are, however, some outliers. The trees for `grep` 3c3bdace have 4 nodes on average, so the one shown in Figure 3.15 is slightly smaller than average. However, ALHAZEN does not create unreasonably large trees for a bug like this.

This raises the question why the tree for `grep` c96b0f2c is so extremely large. Further investigation shows that the reason is in the grammar. `Grep` can be configured to use a different locale than the system's default. This is done with an environment variable, which, in the grammar, is modeled as an alternative over all available locales. Now, `grep` c96b0f2c depends on the locale, and therefore the decision tree contains a cascade of nodes which check for the exists() of many of those locales. For this bug, even the original bug report gives four different inputs which trigger the bug.

Table 4.3 gives the relation between the number of features used in the grammar, and in the decision tree. Ignoring the calculator, which is really just an example, the search space for all subjects is reduced to 2%, or even less. The least significant reduction is for NETHACK, however, NETHACK has just 62 features. Therefore, even a reduction to 1 feature would be 1.6%. On this subject, 2% are therefore close to the maximal possible reduction. Looking at all subjects again, software developers can reduce the input variety they need to reason about in debugging a lot.

> ALHAZEN reduces the search space by at least 98%.

4.2 Preparing the User Study

The result from the previous section is encouraging, however, a real user study is required to see whether those advantages materialize in reality. This study is supposed to answer the following research questions:

1. Do the test cases, generated by ALHAZEN, help developers to fix bugs faster?
2. Does the diagnosis, generated by ALHAZEN, help developers to fix bugs faster?
3. Do the test cases, generated by ALHAZEN, help developers to provide better fixes?
4. Does the diagnosis, generated by ALHAZEN, help developers to provide better fixes?

The design for the study was developed and refined in four phases:

Initial Design In this phase, we developed the objective of the user study, and proposed a design for a study that can answer the research questions. Results from this phase will be presented in Section 4.2.2.

Pre-pilot Study Within the pre-pilot, we tested the proposed design with 25 participants. The results, including the problems with the design we discovered, will be described in Section 4.2.3.

Design Revision Building on the collected experience, we revised the study design. The changes will be described and discussed in Section 4.2.4.

subject	average # of features in the decision tree	# of features in the grammar	Remaining Search Space
NetHack	1.25	62	2.0%
calculator	2.0	17	11.8%
closure 1978	9.0	1919	0.5%
closure 2808	7.75	1881	0.4%
closure 2842	6.25	1899	0.3%
closure 2937	7.75	1890	0.4%
closure 3178	5.25	1939	0.3%
closure 3379	7.5	1886	0.4%
find 07b941b1	7.5	1172	0.6%
find 091557f6	2.25	1179	0.2%
find dbcb10e9	7.25	1168	0.6%
find ff248a20	7.5	1178	0.6%
genson 120	1.0	63	1.6%
grep 3220317a	9.5	1083	0.9%
grep 3c3bdace	1.5	1085	0.1%
grep 5fa8c7c9	7.0	1083	0.6%
grep 7aa698d3	7.75	1085	0.7%
grep c96b0f2c	11.25	1087	1.0%
jerryscript 3267	5.75	1914	0.3%
jerryscript 3276	5.0	1881	0.3%
jerryscript 3286	4.25	1915	0.2%
jerryscript 3297	6.75	1897	0.4%
jerryscript 3389	4.75	1971	0.2%
jerryscript 426	6.0	1895	0.3%
rhino 385	4.25	1197	0.4%
rhino 386	6.0	1192	0.5%

Table 4.3: Search space reduction by ALHAZEN.

Pilot Study In the pilot study, I tested the revised design with 38 participants. The results can be found in Section 4.2.6.

The first two phases, initial design and pre-pilot were carried out by Sven Fackert, as part of his master thesis[18]. I advised him throughout the process.

4.2.1 Measuring Repair Quality

Research questions 3 and 4 investigate whether the quality of the provided fixes improves when the developers have test cases or diagnosis available respectively. This brings up the question how repair quality can be measured.

In general, a repair is successful if it fixes the bug. However, it is a common problem in quality assurance that the program behavior cannot be checked entirely, as verification is impractical in many cases. Therefore, the success of repair is usually estimated based on success of a set of test cases, exercising the behavior for specific points of the input space.

Definition 21: Successful repair
A repair is *successful* if the resulting fix passes all tests.

For our study, it means that we get a boolean variable which indicates whether a repair was successful. For a boolean variable, a chi-squared[64] test indicates whether the frequency of successful repairs in different participant groups are significantly different.

This definition, however, does not allow to reason about different levels of repair quality. There is no notion of one repair being better (or worse) than another repair. A different approach to measurements of repair quality was presented by Yi et al. [74]. They establish a correlation between repair quality in automated repair approaches, and classical test suite adequacy metrics like code-coverage. To this end, they utilized multiple test suites, each with a different value for some coverage metric, and derived an automated fix for each of those test suites. Then, they used the union of all test suites to check whether the repair fixes the bug. Within their work, regression ratio is defined as follows:

Definition 22: Regression ratio as defined by Yi et al. [74]
The *regression ratio* for set of repair attempts is defined as

$$\text{regression ratio} = \frac{\text{\# of failed repairs}}{\text{\#total number of repairs}}$$

A repair is considered successful if it passes all tests, just as defined before. Then, the fraction of successful repairs for a specific value of a coverage metric is calculated.

Mind that they have more than one repair for each value of a coverage metric. This means they do not need to measure the quality of a single repair, but the quality of a group of repairs. Within the user study, each participant suggests a repair, and we are interested into a quality measure for every single repair. Therefore, this definition is not directly applicable.

	t_1	t_2	t_3	quality
r_1	pass	pass	pass	1
r_2	fail	pass	pass	0.66
r_3	fail	pass	fail	0.33
r_4	fail	fail	pass	0.66
r_5	fail	fail	pass	0.33

Table 4.4: Three test cases, three repairs and the repair quality for each.

We can, however, apply the same idea to measure the quality of a single repair. That is, we define quality as the fraction of tests passed.

Definition 23: Repair Quality
The *quality of a repair* is defined as

$$\text{quality} = \frac{\text{\# of passed tests}}{\text{\# of tests}}$$

At first glance, this looks like a continuous scale. A repair which passed 5 out of 10 tests is 0.1 better than a repair which passes 4 out of 10 tests. However, this is intuition is misleading: The quality value gives no indication of which subset of the test cases passes. Let's say we have three tests t_1, t_2 and t_3. There are five repairs, r_1 to r_5. Table 4.4 shows which repair passes which tests.

Let's look at the tests first: t_1 is passed by just one repair, while t_2 is passed by three repairs, and t_3 is passed by four repairs. One could say that t_1 is more difficult than t_2, which is more difficult than t_3. But then, can we really say that the quality difference between r_1 and r_2 is the same as the difference between r_2 and r_3? Does passing the most difficult test mean the same increase in quality as passing the least difficult test? This example shows that quality allows to order the repairs, but it cannot be compared directly. Even repairs with equal quality may not in fact be equally good. r_3 passes just one test, namely t_2. The same is true for r_5, which passes no test but t_3. Therefore, r_5 and r_3 have the same repair quality, however, t_3 is easier than t_2, so, this notion may be misleading.

Quality data is not continuous, but ordinal at best. The value hides some of the underlying complexity.

The question which metric is more appropriate, frequency of successful repairs within a group or quality, is somewhat philosophical. One can argue that a repair which does not fix the bug entirely, and still fails some test cases, is useless. Following this argument, the quality value is meaningless: Any quality lower than 1 is insufficient. On the other hand, one can also argue that the program behaves correctly for a larger part of the input space, and therefore there is some improvement.

Within the user study, I will examine quality in both definitions: As a binary successful repair variable, as well as the ordinal quality value.

Within the pre-pilot study As mentioned before, the pre-pilot study was conducted by Sven Fackert as part of his master thesis. Within the thesis, Fackert worked with regression ratio, defined like this:

Definition 24: Regression ratio as defined by Fackert [18]
The *regression ratio* of a repair is defined as

$$\text{regression ratio} = \frac{\text{\# of failed tests}}{\text{\# of tests}}$$

A test is either passed or failed, and therefore it is easy to see that

$$\text{regression ratio} = 1 - \text{quality}$$

All observations about quality hold for regression ratio just as well. I decided to use quality in the thesis, and used the mentioned formula to a convert data from the pre-pilot study, because it is more intuitive to have a higher-is-better metric, such as quality.

4.2.2 Initial Design

We first evaluated our study design in a pre-pilot with 25 participants. This pre-pilot will be described in this section.

Overview of the Study Design

In order to answer our research question, we want to observe the time taken to fix a bug. We want the participants to be knowledgeable in Python programming, therefore, we perform a pre-screening. As part of this pre-screening, participants have to answer 10 questions about Python programming, and solve three small programming task. Participants who got at least 9 out of the 13 problems correct where invited for the main study. We raffled off a 15 € amazon voucher among the participants of the pre-screening.

The participants which passed the pre-screening where assigned to 1 out of 4 groups in a round-robin fashion.

The d0t0 group serves as control group: How do software developers perform on our tasks without any debugging aid? With the othe rgroups, we test three different treatments: ALHAZEN's diagnosis, the generated tests, or both.

Participants first received a training task, to make sure they know how to use the infrastructure of the user study. After that, we presented them with four programming tasks in the Python programming language. The details of task selection, and the tasks themselves will be given in Section 4.2.2. After each task, participants answered a questionnaire about the task, and the perceived usefulness of ALHAZEN. After the last task, participants where asked additional questions about their impression of ALHAZEN. The questionnaires will be described in Section 4.2.2.

Participant Groups within the User Study
Within the user study, the participants are assigned to one of the following groups:

d0t0 The control group receives neither ALHAZEN's diagnosis, nor generated test cases. Therefore, they debug based on just a single test case.

d1t0 The diagnosis group received ALHAZEN's diagnosis, but not the generated test cases.

d0t1 The test group received the generated test cases, but not ALHAZEN's diagnosis.

d1t1 The test and diagnosis group received both, generated test cases and ALHAZEN's diagnosis.

Figure 4.2: Overview over the participant groups within the user study.

Technical Setup

During the study, participants need to fix bugs, and we need to be able to measure time. Due to the COVID-19 pandemic, and also due to the high number of participants, it wasn't possible for us to conduct the study on site. Therefore, we decided to use a customized version of Jupyter Notebook[38]. This tool allows developers to write Python code in the browser, run it and examine outcomes. It does not contain a traditional debugger, which means that participants did not have such a tool at their disposal. We customized jupyter to allow for time measurements and to include the questionnaires before and after each task.

Tasks

The tasks within the user study are supposed to model real-world bugs, that developers might encounter in their daily work. At the same time, they need to be solvable by ALHAZEN, and they need to be in the Python programming language. In Section 3.3.3, I explained how the RHINO 385 bug cannot be diagnosed by ALHAZEN. It would be pointless to use such a bug in the user study. The result would just be further proof for a known limitation of ALHAZEN.

Setting up ALHAZEN includes to write a grammar for the program under test, write a predicate of interest and run the tool. These tasks require some understanding of grammars and ALHAZEN's operation. They cannot be solved within 15 minutes. Also, I assume that, if ALHAZEN is used professionally, at least the grammar writing would be caried out once, and reused. But then, the time spend on grammar writing is of limited concern. Therefore, we decided not to include those steps in the user study.

Furthermore, I decided not to use real bugs. Those would be in a large software system, and participants would need to know a lot of context about those systems. Since learning the general structure of the code and other project-specific information would take most of the participant's time, effects caused by

Alhazen may be clouded. Also, it would be very difficult to recruit participants who are willing to spend more than one hour on such a study, which means that each of the four tasks may take at most 15 minutes, preferably less. Therefore, I decided to use four toy tasks, modeled after real-world bugs.

I examined three sources to find suitable tasks:

1. The debugging book[76] is an online resource which teaches current debugging techniques. The material includes examples, that are used to demonstrate these techniques.

2. I looked at the bug tracker for Ansible[4]. Ansible is an IT automation tool, that can be used to automate repetitive administration tasks. Ansible is written in Python, and among the top repositories with Python code on github. Within the Ansible repository, I looked at the 50 most recent merge request [1] which were labeled as bugfix, and were neither labeled as documentation nor contained the words "Update" and "version" in the title.

3. I looked at the bug tracker for Django[20]. Django is a web framework which is written in Python, and used for many projects. Just as Ansible, it is one of the top Python projects on github. Within the Django repository, I looked at merge requests where the title started with "Fixed", followed by the reference number of a bug report.

For the Django and Ansible repositories, I wrote a quick, one line summary of the changes. Afterwards, I sorted the bugs into categories, based on the one line summary.

As an example, merge request #73947[3] in Ansible fixes a problem with automated configuration of machines running Amazon Linux. Ansible tries to detect the used kernel version and fails, because old versions of Amazon Linux used a naming scheme that was not supported. My one-line summary for this bug was "missing special case in string parsing". Later, I sorted this bug into the category "Missing Special Case". Bug #14179[10] in Django was a crash when the Origin Header in a http request contained an invalid host. My one-line summary was "missing special case, missing host", as the fix indicates that Django should just ignore this case. I rated this as "Missing Special Case" as well. All bug numbers, one-line summaries and categories can be found in Appendix A.

After this analysis, I came up with the following artificial bugs:

RemoveHTML The code for this task is used as an example within the debugging book[76]. The code in question is supposed to remove HTML tags from a string, however, it contains a bug which makes it fail if there are quotes in the input.

uname The uname task parses the output of the `uname` facility to obtain version number, kernel specifier and other properties of the system it is running on. The code contains a bug where kernel version numbers that do not include a patch number cannot be parsed. This artificial bug is inspired by merge request

[1] Merge Requests are how developers contribute code to the ansible project.

#73947[3] in ansible. As in the merge request, a regex which decomposes a kernel identifier does not handle a special case.

RPN The RPN task consists of a function which accepts a string with a mathematical expression in reversed polish notation, and returns the result of evaluating this expression. The bug is within the handling of the `"+"` operator. For this operator, the code does not increase an index which is used to access the string. This causes an endless loop, which is terminated as soon as the internal stack of the implementation is empty.

This task was inspired by the "Incorrect Communication with Other Program Parts" category. While the program in the task is too small to have other program parts, merge requests in this category often added or removed method calls which informed other program parts about program state (e.g. as part of the Observer pattern[22]). The missing update of an index is – in essence – a similar type of bug.

Lists The Lists task is a parser for Lists of numbers in Lisp notation. It would fail to parse the digit 9, due to incorrect API usage of the Python standard API.

Questionnaires

In addition to raw performance numbers, we want to collect the subjective feedback of the participants towards Alhazen. In order to do so, we use two different questionnaires:

1. The Task-Questionnaire is used after each task, and contains questions specific to the tasks.
2. The Final Questionnaire is used at the end of the study, and contains questions about Alhazen in general.

In the following, I will describe each questionnaire.

Task Questionnaire The task questionnaire aims to collect feedback for this specific task. We asked participants who were presented with the Alhazen diagnosis (groups d1t1 and d1t0) these questions:

1. The diagnosis was useful.
2. The diagnosis was easy to understand.

Those questions aim at identifying potential problems with how the diagnosis is framed.

Participants in groups d0t1 and d1t1 were also asked about the tests:

3. The additional tests were useful.

With this question, we attempt to find out whether Alhazen's test cases contribute to fixing the bug. We did not ask about the understandability of the test cases, as we did with the diagnosis, because test cases are standard instruments in debugging, and written in Python.

For all questions about the diagnosis as well as about the tests, participants gave their answers in a 7-element Likert scale[44], choosing between 'Strongly disagree' and 'Strongly agree' in 7 steps.

Within the Task Questionnaire, participants also had the option to give additional remarks about the task in a free-text form. Also, they were asked whether they had any interruptions while solving the task. Interruptions would mean that our time measurement is not reliable.

Final Questionnaire In the final questionnaire, we attempt to collect the participant's final thoughts about Alhazen. In order to do so, we adapted the Technology Acceptance Model (TAM)[14]. Participants in the d1t0 and d1t1 are supposed to answer 10 questions:

1. Using the outputs from Alhazen in my job would enable me to accomplish tasks more quickly.
2. Using the outputs from Alhazen would improve my job performance.
3. Using the outputs from Alhazen in my job would increase my productivity.
4. Using the outputs from Alhazen would enhance my effectiveness on the job.
5. Using the outputs from Alhazen would make it easier to do my job.
6. I would find the outputs from Alhazen useful in my job.
7. Learning to use the outputs from Alhazen would be easy for me.
8. Using the outputs from Alhazen would be clear and understandable.
9. It would be easy for me to become skillful at using the outputs from Alhazen.
10. I would find the outputs from Alhazen easy to use.

The TAM, as found in literature, contains two more questions:

11. I would find it easy to get the Alhazen Diagnosis to do what I want it to do.
12. I would find the Alhazen Diagnosis to be flexible to interact with.

We decided not to ask those, as we did not give participants the chance to actually use Alhazen, and therefore they would not be able to answer those questions.

4.2.3 Conducting the Pre-Pilot

We posted an invitation to the study in several groups that are frequented by Python developers on Facebook, Reddit and LinkedIn. During the pilot study, we used only small to medium-sized groups, as we wanted to be able to announce the full study, which requires more participants, in larger groups later. We also

Recruitment Step	Participants	Ratio to previus step
Potential Audience	85000	
Visited Study Page	165	0.19%
Completed Screening Test	42	25.45%
Passed Screening Test	30	71.43%
Invited to second part	30	100.00%
Completed all tasks	25	83.33%
Completed without interruptions	21	84.00%

Table 4.5: Conversion rate for the invitation to the pre-pilot study. This data is taken from Sven Fackert's master thesis[18].

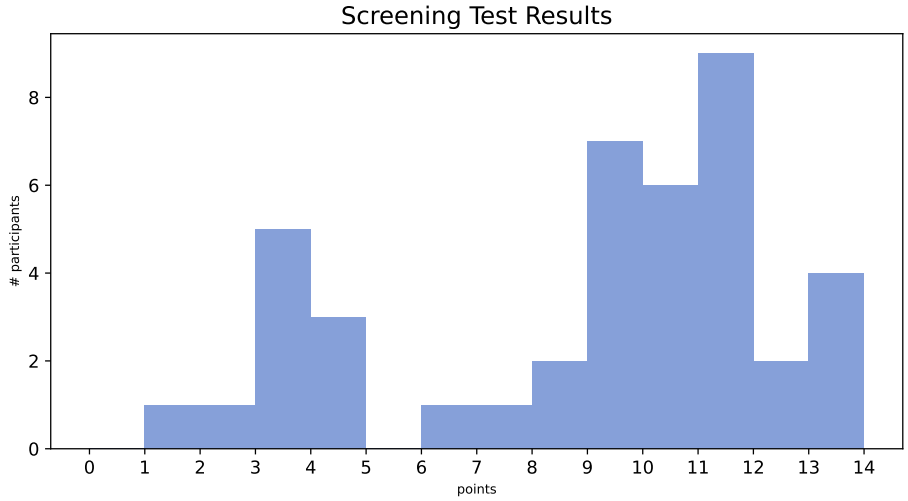


Figure 4.3: Histogram of screening test results within the pre-pilot study (data collected by Sven Fackert in his master thesis).

contacted personal acquaintances – mostly fellow students who transitioned to industry positions – and asked them to participate.

Table 4.5 shows the number of participants in each step. The groups we posted in had roughly 85000 members in total. Out of those, 0.19% visited the start page of the study, and 42 (25.45%) completed the screening test. We counted participants who did not answer the demographical questions on the last page of the screening test as participants who did not complete the screening test. This means that 123 potential participants either did not even start the screening test, or did not finish it. As those participants did not interact with the web page further, we have no data that may help to diagnose why the conversion rate is this low at that point.

30 of the remaining 42 participants passed the screening test. Participants were expected to get at least 9 out of 13 points in the screening test.

Figure 4.3 shows the distribution of achieved results. There is a large group of people, 10 in total, who scored between 0 and 5 points. Those are clearly under-performers. For this group, the screening test serves its purpose: It keeps people with insufficient Python knowledge from participating. Within

Group	mean	std
d0t0	1254.43	435.62
d0t1	568.24	321.61
d1t0	609.30	376.86
d1t1	651.56	632.22

(a) RemoveHTML

Group	mean	std
d0t0	1007.62	713.30
d0t1	1174.94	597.00
d1t0	1289.87	478.36
d1t1	711.54	1080.21

(b) uname

Group	mean	std
d0t0	954.08	296.14
d0t1	647.83	225.85
d1t0	1660.01	1949.24
d1t1	608.76	606.45

(c) Lists

Group	mean	std
d0t0	858.79	882.33
d0t1	658.46	362.57
d1t0	556.30	314.64
d1t1	405.08	258.14

(d) RPN

Table 4.6: The mean and standard deviation of the time spend per task (in seconds) within the pre-pilot study.

the second group, 4 participants are excluded because they had between 6 and 9 points. However, it is clear that there is a strong difference between the number of people who scored 9, and 8 points respectively. Therefore, the threshold of 9 is choosen correctly.

We invited the 30 participants who scored more than 9 points to participate in the study, and 25 of them actually did. Out of those, 4 participants reported interruptions. We excluded the data from those participants from further analysis, as the time measurements are imprecise.

> The pre-pilot study shows that our recruitment strategy is valid, but the conversion rates are low.

Additionally, several of our announcement posts were deleted on Facebook. Administrators of Facebook groups do not seem to like this kind of posts.

Analysis of Pre-Pilot Study Results

As mentioned in Section 4.1, the main focus of the pre-pilot study was on testing the study design, rather than collecting data on Alhazen. Therefore, the analysis of the data concentrates on possible shortcomings of the infrastructure, rather than findings about Alhazen.

Table 4.6 shows the time spend per task for the individual groups. Looking at this data, it is striking how large the standard deviations are. In some cases, the standard deviation is higher than the mean (d1t1 with uname, d1t0 with Lists), in other cases extremely close to the mean (d1t1 with RemoveHTML, d0t0 with RPN, d1t1 with Lists).

However, the task durations in Table 4.6 indicate that we reached one of our design goals: Participants spend on average 10 minutes per task, which is the time that we were aiming for when we designed the tasks. However, completing the study still took about an hour, which means that the 15 € amazon vouchers we handed out are below the pay professional software developers expect. The low conversion rate may be linked to this.

Group	Repair Quality		
	0.56	0.89	1.00
d0t0	0	2	2
d0t1	1	1	3
d1t0	0	1	1
d1t1	0	8	6

(a) RemoveHTML

Group	Repair Quality				
	0.00	0.18	0.55	0.73	0.91
d0t0	0	2	0	1	1
d0t1	1	1	0	3	0
d1t0	1	0	0	0	1
d1t1	2	2	2	5	3

(b) uname

Group	Repair Quality	
	0.50	1.00
d0t0	1	3
d0t1	1	4
d1t0	0	2
d1t1	1	13

(c) Lists

Group	Repair Quality	
	0.30	1.00
d0t0	0	4
d0t1	1	4
d1t0	0	2
d1t1	1	13

(d) RPN

Table 4.7: Results for Repair Quality as collected in the pre-pilot study.

> The compensation per participant should be increased.

Table 4.7 shows the repair quality for all tasks. As observed in Section 4.2.1, repair quality is an ordinal variable. Therefore, I treat the data as categorial values, and give the frequency for each observed value, in each group. The number of categories differs for the different tasks: For uname, there are five different values in total, and no participant provided a fully functional repair. For the other tasks, there are not as many different values, just 2 for Lists and RPN, and 3 for RemoveHTML. This indicates that the tasks do not give enough room for errors. We can confirm this by looking at the individual submissions. For both the Lists and the RPN task, all participants either provided a correct fix, or did something ridiculous.

> If the regression ratio is to be evaluated, the tasks Lists and RPN need to be reworked.

In addition to the regression ratios and task durations, we also collected free-text remarks from participants. Upon review of those remarks, we made the following observations:

Technical Problems Two participants reported technical difficulties: In one case, the participant wrote "My kernel broke and I was not able run tests anymore :-(". The kernel is the component in a jupyter notebook which executed the Python code, so the participant was obviously aware of the technical basis for our infrastructure. However, looking at the logs, we cannot confirm whether the kernel actually caused problems. Another participant wrote: "On the second task, after fixing the code, the test cases gave no output. Test cases should show pass to indicate succesful fixes." This may again point to a crashed kernel, but also to a lack of feedback on passed tests in the user interface. I observed the second problem as well, and I assume that missing feedback in the UI lead the first participant to believe their kernel

had crashed.

> It should be made sure that each test case always gives feedback.

Unclear task description For the uname task, several participants wrote statements such as "It was unclear if the specifier and the 'patch' version should be stored together or as a separate field as provided in my solution. ", "expected output not clear" or "The problem was not the debugging itself, but that I had too few information about what the program should do." Obviously, participants cannot be expected to fix a bug if they don't know the expected behavior.

> The uname task lacked information about the expected output during the pre-pilot study.

4.2.4 Refining the Study Design

The pre-pilot study in the previous section highlighted some problems with the study design. In this section, I will detail the changes done to the infrastructure and tasks, before the pilot study was undertaken. The first finding was about the regression ratio in the Lists and RPN tasks.

> For reference
> If the regression ratio is to be evaluated, the tasks Lists and RPN need to be reworked.
> See Section 4.2.3

In order to address this problem, both tasks were replaced.

RPN 2 The RPN task was replaced with a new task, RPN 2. While this task is a calculator that accepts inputs in reversed polish notation as well, the bug is different. Within the old RPN task, the bug was a missing update of an index, in an attempt to mirror the "Incorrect Communication with Other Program Parts" category of bugs. In the new task, the code will instead increment the index by 1 after a number was parsed. The index needs to be incremented while parsing a number, as it has to point to the next character after the number when number parsing finishes. Therefore, participants may believe that it is correct to add 1 at this point. However, the code updated the index as part of a loop earlier.

Lists The Lists task was replaced with a new task which sums up a list of numbers. However, it cannot handle negative numbers when reading the input from a string. This is an example for the "Special Case Missing" category.

The next observation was:

> For reference
> The compensation per participant should be increased.
> See Section 4.2.3

Participants each receive a 30 € amazon voucher. The voucher which is raffled to participants of the screening test was replaced with a 30 € voucher as

well.

In Section 4.2.3, it was not entirely clear whether the problem encountered by the participants was a crash within jupyter's kernel, or a lack of feedback in the user interface. To address this, I made sure that all tests always give visible feedback, and we changed the infrastructure to give participants an option to restart the jupyter kernel. Obviously, only participants who know jupyter well enough will be able to recognize a kernel failure as such. Participants who do not know jupyter likely still cannot recover from a crashed kernel. As explained in the previous section, I consider it much more likely that the participant mistook a lack of visible feedback for a crashed kernel.

Within the uname task, all tests were rewritten to contain more information about the expected output. This also means that tests created by Alhazen were recreated. While the version used in the pre-pilot study just generated inputs, and checked for a program crash, the new version used a construction-based oracle (see Section 1.4) to be able to provide meaningful assertions to the participants.

Estimating the required number of participants

Another important factor is the number of participants per group that will be invited to the study. This number is subject to several constraints: On the one hand, a larger number of participants means more statistical power, and more reliable results. On the other hand, the low conversion rate reported in Section 4.2.3 indicates that it will be difficult to collect a high number of data points. Also, with a compensation of 30 € per participant, the available funding becomes a limiting factor. Therefore, the number of participants invited should be large enough to gain statistical power, but at the same time small enough to be practical.

The number of participants required for a given level of statistical significance can be calculated, based on estimates of the observed statistics. I am going to estimate those numbers based on the data observed during the pre-pilot study.

Sample Size Estimation for Task Duration First, I am going to estimate the required number of participants for research questions 1 and 2, which are concerned with the task duration. Table 4.6 shows the observed mean and standard deviations for the task durations. I estimate that the effect size will be strong[2]. This means that the value of Cohen's d (see Cohen [11]) will be at least 0.8.

Each of the groups d0t0, d1t0, d0t1 and d1t1 can be compared with each other group. Those comparisons are symmetrical, meaning that a comparison of d0t0 to d1t0 gives the same result as a comparison of d1t0 to d0t0. The effect size is always calculated as a comparison of one group to another, so, with four groups, there are 6 comparisons. The effect size table, as given by Cohen [11], can be used to see the number of participants required to get an effect size of 0.8, assuming the real study gives the same means and standard deviations as observed within the pre-pilot study. Table 4.8 gives Cohen's d and the estimated number of participants required for each of the six comparisons.

[2]this is commonly done for studies of this kind.

group 1	group 2	Cohen's d	#participants
d0t0	d1t0	1.48	9.00
d0t0	d0t1	1.58	9.00
d0t0	d1t1	0.95	17.00
d1t0	d0t1	0.11	393.00
d1t0	d1t1	-0.07	1571.00
d0t1	d1t1	-0.13	393.00
median			205.00

(a) RemoveHTML

group 1	group 2	Cohen's d	#participants
d0t0	d1t0	-0.40	99.00
d0t0	d0t1	-0.23	175.00
d0t0	d1t1	0.27	175.00
d1t0	d0t1	0.19	393.00
d1t0	d1t1	0.54	45.00
d0t1	d1t1	0.43	64.00
median			137.00

(b) uname

group 1	group 2	Cohen's d	#participants
d0t0	d1t0	0.34	99.00
d0t0	d0t1	0.23	175.00
d0t0	d1t1	0.51	45.00
d1t0	d0t1	-0.28	175.00
d1t0	d1t1	0.48	64.00
d0t1	d1t1	0.70	33.00
median			81.50

(c) RPN

group 1	group 2	Cohen's d	#participants
d0t0	d1t0	-0.36	99.00
d0t0	d0t1	1.03	12.00
d0t0	d1t1	0.57	45.00
d1t0	d0t1	0.52	45.00
d1t0	d1t1	0.54	45.00
d0t1	d1t1	0.06	1571.00
median			45.00

(d) Lists

Table 4.8: Estimated number of participants necessary, based on observed task durations.

Task	p-Value	Effect Size	# of participants	required # of participants (strong effect)	required # of participants (observed effect)
RemoveHTML	0.93	0.08	26	15.05	1683.37
uname	1.00		26	15.05	
RPN	0.68	0.14	26	15.05	496.38
Lists	0.68	0.14	26	15.05	491.60

Table 4.9: Estimated number of participants based on frequency of successful repairs, together with the statistics observed in the pre-pilot study.

Based on task duration, the estimates range from 9 participants per group, to detect a difference between d0t0 and d1t0 for the RemoveHTML task, to 1571 to detect a difference between d1t0 and d1t1 for the RemoveHTML task. The median of the estimated number of participants required for the RemoveHTML task is 205, signifying that I would require 205 participants to be able to get an effect size of 0.8 for 3 out of 6 comparisons. Those numbers assume that standard deviation and mean within the full study are the same as within the pre-pilot study.

Sample Size Estimation for Frequency of Successful Repairs For research questions 3 and 4, there are two options on how to define quality (see Section 4.2.1). I will deal with quality as frequency of successful repairs first. This definition assumes that a repair is always either successful, or failed. Therefore, repair success is a binary variable, and the only meaningful statistic is how many participants per group provided a successful repair. Statistical significance for this kind of variable is checked with a chi-squared test which compares the frequency of successful repairs within each group.

For this kind of tests, the number of participants required can be estimated as described by Cohen [11], Table 4.9 shows those estimates. The first three columns give p-Value, effect size and number of participants within the pre-pilot study. Those results hold no surprises: The p-Values are high, and therefore none of the results is statistically significant. The effect sizes are small. For the uname task, no participant provided a correct solution. This was most likely caused by the incomplete task description. With a frequency of 0, calculations for effect size cannot be performed, as they require a division by this frequency. The fifth column gives the estimate on the required number of participants, and it holds a surprise: According to those results, we require 16 participants, but the pre-pilot study had 26 participants already. Still, the data is not statistically significant, as one can see from the huge p-values. The reason is my assumption about the effect size: I assume an effect size of 0.8. However, the observed effect

For reference

Definition 23: Repair Quality
The *quality of a repair* is defined as

$$\text{quality} = \frac{\text{\# of passed tests}}{\text{\# of tests}}$$

See Section 4.2.1

sizes in the pre-pilot study are all below 0.2. The last column of Table 4.9 shows the required number of participants, calculated based on the observed effect size. They are orders of magnitude higher, and not within the range where we could expect to have this number of participants in the study.

Sample Size Estimation for Repair Quality Estimating the required number of participants based on repair quality requires careful attention: As remarked in Section 4.2.1, repair quality is not continuous. Therefore, I have to treat it as a categorial variable, and apply a chi^2 test, just as in the previous section.

Table 4.10 shows the obtained estimates. The first observation is that the effect sizes are higher than for frequency-based evaluation. However, the effects are still weak. Then, the p-Values are somewhat lower, but still not statistically significant. As observed before, this is the best we can hope for in the pre-pilot study. The estimates on number of participants required are in the same order as for frequency-based evaluations, if at all, there is a tendency for them to be somewhat lower.

Conclusion Based on task duration, quality and frequency of successful repairs, estimates indicate that I require a number of participants which is higher than practical. However, the data I based my estimates on was collected in the pre-pilot study, and we already observed some problems with this data. Most notably, the uname task did not define the expected outcomes, and therefore participants were confused. It is likely that the data collected from this task is invalid, and should not be used for estimating the number of required participants. For the RPN and Lists tasks, I observed that the tasks were either solved successfully, or not at all. Therefore, the quality for those tasks are likely unusable as well. Still, 377 participants for a difference in quality in the RemoveHTML task seems impractical.

Looking at the estimates based on task duration, Table 4.8 also gives the median for each task. This indicates that about 200 participants per group would be enough to get a meaningful value for half the comparisons. This is still not practical.

Another option is to concentrate on one comparison. If I want to detect the difference between d0t0 and d1t1, which is certainly the most interesting, I require 17 (RemoveHTML), 175 (uname), 45 (RPN) and 45 (Lists) participants per group. This leads to a total of 350 participants (two groups, 175 participants each). However, due to the problems with the uname task, it is questionable whether this data is reliable. Assuming that the estimates for RPN and Lists are more realistic, I can detect a significant difference for this comparison with 50

Task	p-Value	Effect Size	# of participants	required # of participants (observed effect)
RemoveHTML	0.51	0.18	26.00	377.74
Machine-Info	0.64	0.18	26.00	536.02
RPN	0.68	0.14	26.00	496.38
List	0.68	0.14	26.00	491.60

Table 4.10: Estimated number of participants based on repair quality, together with the statistics observed in the pre-pilot study.

participants per group. This number works for the d0t0 against d1t1 comparison on RemoveHTML, RPN and Lists, and gives us some room, 5 additional participants per group, in case our estimates where slightly off. Using 50 participants per group means that I will not be able to see statistically significant differences in repair quality or frequency of successful repairs, but the estimated number of required participants for quality comparisons is more than 100 in almost all cases, which I consider unpractical.

I will therefore attempt to recruit 50 participants per group in the pilot study.

4.2.5 Conducting the Pilot Study

As for the pre-pilot, I posted invitations for participants on Facebook, Reddit and LinkedIn. This time, I also used twitter, posting on @DomSteinhoefel, @bjrnmath, @debugging_book, @FuzzingBook and @AndreasZeller [3]. As for the pre-pilot, posts were deleted immediately on Facebook and Reddit. I also used the mailing lists of our previous courses on debugging and cybersecurity to contact our former students. This is a controversial decision, as I was aiming to recruite professional software developers, and those of our students who still use their university mail addresses are most likely not professional software developers yet.

I did not record which of our postings attracted how many participants. However, the postings were done about a week apart from each other, so I can estimate which posting attracted how many participants based on the date they took the test. Considering this data, LinkedIn, the @debugging_book, @FuzzingBook and @AndreasZeller account on twitter as well as the student mailing lists had the highest conversion rates. This is not necessarily precise. The postings on the student mailing lists and the @AndreasZeller twitter account were done on the same day, so my timing-based method is insensitive for a

[3] I'd like to use the opportunity to thank Dominic Steinhöfel, Björn Mathis and Andreas Zeller for promoting the user study.

Recruitment Step	Participants	Ratio to previous step
Visited Study Page	269	
Completed Screening Test	62	23%
Passed Screening Test	58	93%
Invited to Second Part	58	100%
Completed all Tasks	38	65%

Table 4.11: Conversion rate for the invitation to the pilot study.

difference between those. Also, participants may have clicked a link on another platform or account days after the initial posting. Especially on the twitter accounts, those tweets stayed the most recent – and therefore topmost – entry for several days.

Table 4.11 shows the total number of participants, and how far into the process they went. Comparing to the pre-pilot, there is a noteable change in passing rates for the screening test: 93% of the participants did pass. 3 out of the 4 participants who failed provided no answers at all, or at least no answer to most of the questions. It is reasonable to assume that those did not intent to participate in the first place.

As for the difference to the pre-pilot study, we can only assume that our first call for participation reached and motivated some people who did not meet the requirements. This probably happened when recruiting among personal acquaintances [4].

> The call for participation reaches and motivates professional developers only, therefore the screening test is mostly pointless.

In contrast to the pre-pilot, the time used for the screening test was measured. We did not instruct participants to solve the screening test without interruptions, and we did not ask whether they had interruptions afterwards, so the times may not be precise. Participants took 17 minutes on average to solve all three programming problems. The standard deviation was about 14 minutes, with a minimum at about 3 minutes, and a maximum of about 67 minutes.

Table 4.11 gives a number of 38, or 65%, who finished the study. This number means participants who finished all tasks. All 58 participants who were invited for the second part started the first task, but some dropped out later.

All in all, this shows that it is problematic to find a sufficient number of participants: Despite thousands of people who saw the call for participation on twitter and LinkedIn, only 269 people visited the start page of the study. Even if it is quite impressive to see that 13% finish the study, it will not be possible to get to a sizeable number of participants this way. We stopped accepting additional participants one week after the last twitter post, with no expectation of additional participants.

> The currently employed recruitment strategy does not generate the required number of participants.

[4] All participants remained anonymous, so we cannot be sure.

Task	# participants	outliers	interruptions	remaining # participants	total removed
RemoveHTML	42	1	3	38	4
Machine Info	40	1	6	33	7
RPN 2	36	1	8	28	9
Lists	38	2	4	32	6

Table 4.12: Participants that were removed in cleaning per task.

We either need more recruitment channels, say more LinkedIn groups or additional high-reach twitter accounts, or a new strategy entirely.

4.2.6 Analysis of Pilot Study Results

As in the pre-pilot, the main objective of data analysis is to find potential problems with the study design. Therefore, I will look at each task individually, and report on the observations. Due to the low number of participants, the data collected in the pilot study is not statistically significant. The only type of data that may provide meaningful insights is the free text additional remarks collected from the participants. Those may provide some insights into how participants use Alhazen, as well as problems with the study design.

Data Cleaning

As far as data is shown within the following sections, the data was cleaned beforehand. Table 4.12 gives information on the number of cleaned data points. There were three main reasons for removing data points.

For some participants, I observed that the code they handed in was identical to the original task code. Those participants hit the submit button without modifying the code at all. I discarded them from further analysis.

Participants were asked whether they completed the tasks without interruptions. If not, they were asked to report on the interruptions. Each participant who reported that there had been an interruption was removed from the dataset for this task. The fourth column of Table 4.12 gives the number of participants who reported interruptions.

Furthermore, I removed outliers within the time measurements. Participants who were more than 2.5 standard deviations away from the mean task duration were remove from the dataset for this task. The threshold, 2.5 standard deviations, was decided upon before data collection started. The number of participants removed for this reason is shown in column three of Table 4.12.

Task 1: RemoveHTML

The RemoveHTML task is taken from the debugging book[76]. The code attempts to remove HTML markup from a string. It has a bug concerning the handling of quotation marks.

For the RemoveHTML task, 3 participants were removed from the dataset because they reported an interruption while working on the task. Then, 1 participant had to be removed because they were considered an outlier, according to the threshold given in Section 4.2.6.

8 out of 38 participants provided additional remarks. Out of those, three complained about a lack of documentation:

1. I have never used these Markdown, therefore it was a bit difficult to unterstand, what I should do

2. did not get precisely

3. Might need more information in the question as one way to pass the test is to comment out line checking for quotes

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

All in all, I don't think this is a reason to change the task. As the internet, and programming tasks in the context of the internet, are ubiquitous nowadays, participants who do not understand this requirement will be rare. The third participant got the point: The single test case that the groups d0t0 and d1t0 got can in fact be passed by commenting out those lines. Indeed, 4 participants provided a solution which just commented those two lines. Missing test cases or incomplete specification are a standard problem in software engineering, and if Alhazen helps developers to realize a missing test or incomplete specification, this would be good. Therefore, presenting some incomplete specifications is in line with the scientific objectives of our study.

> Test cases and task descriptions within the user study may provide an incomplete specification.

The data on achieved quality is shown in Figure 4.4.

In Section 4.2.1, I discussed that quality data needs to be treated as a ordinal value, which is why I count participants per achieved quality, rather than averaging. Due to the low number of participants, this data is not statistically significant. There are three different quality values: One participant reached a quality of 0.81. This participant, for some reason, replaced a quotation mark with a space, therefore handling spaces instead of quotes. All four participants who reached a score of 0.88 just commented the lines that contain the handling of quotes. Finally, 34 participants provided a correct solution. This is a problem in itself: More variety in the quality of the solution, leads to a more powerful statistical test. Three different quality values are not sufficient here.

> Tasks should be designed to allow for incomplete fixes.

One of the participants who provided a correct solution, wrote:

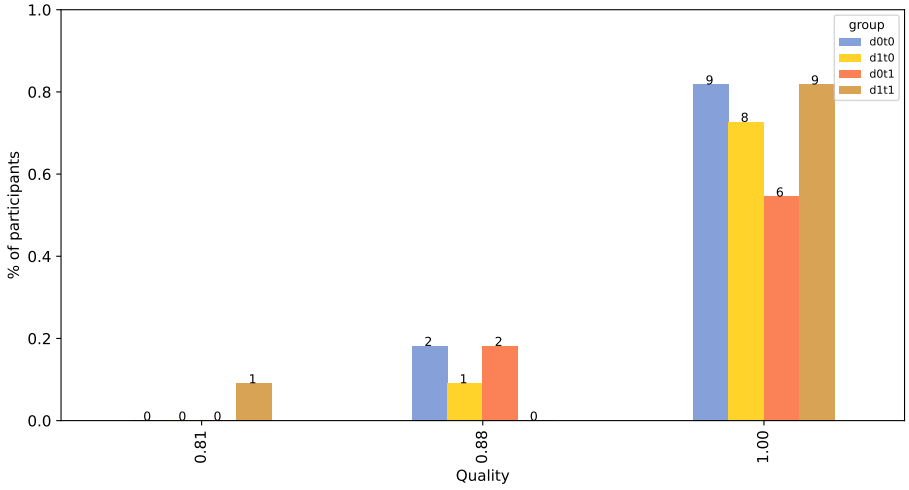


Figure 4.4: A histogram of quality for the RemoveHTML task within the pilot study. While the numbers on top of the bars give absolute values, the bars are scaled relative to the size of the largest group.

4. reading the original source was already sufficient for finding the error so here Alhazen was only used to check that my modification was indeed the needed fix

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Therefore ALHAZEN gave them additional trust in the correctness of their reasoning. Also, they provided a correct fix, so they were not lead to believe that it would be correct to just remove the handling of quotes entirely. However, the participant was in the d1t1 group, and therefore also had additional tests at their disposal. Still, this participant found ALHAZEN somewhat useful.

Another participant realized that they made a mistake after handing in:

5. oh no I realized my mistake now... '>' chars within attributes of strings...

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This is a correct observation, but came too late. Apparently the participant clicked "Submit", while they were still thinking about the task.

One participant reported the usage of an external tool:

6. It was great yeah solved it on my editor and pasted it

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Copy and paste of code may take some more time than solving in the browser directly, but in a remote study, we cannot prevent participants from doing this. I don't think it is a problem.

Finally, one participant points at a flaw in our design. They wrote:

7. I remembered the function (although not the problem/solution) from the Debugging Book

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

In hindsight, it was a terrible decision to use a task from the book, and recruit among our students, who were aware of the book because we use it in our courses. 26 of the 34 participants who provided a correct solution gave the solution that can be found in the debugging book. However, I consider it very unlikely that 26 participants came from our students. Looking at the timeline, some participants provided this solution before we send the call for participation to our students. Also, it is not surprising to see this solution a couple of times: It is the correct solution, after all. We are unable to decide whether participants arrived at the correct solution on their own, or copied it from somewhere else.

> Tasks should not be publicly available before the study is started.

Let's have a look at what other participants who achieved a quality of 1.0 did. One participants provided a fix that is too complicated. They added a condition which is always true to an if statement. The test cases cannot detect this, because there is no program run where this condition would be false.

One participant split an if with a conjunction within its condition into two if statements. This leads to some code duplication, because the else case needs to be duplicated, but is a correct solution. The last participant added an additional `else if` case to an `if` statement, which, again, is unreachable, as the condition is always false.

While many of the solutions are identical, some are semantically identical, but bear marks of working participants: One participant added a `print` statement, indicating the use of print line debugging. Several participant added comments to the source code, likely to aid their understanding. Other solutions are, again, just identical. We cannot be sure how much "cheating" took place.

> In the current design, we cannot reliably detect whether participants submitted their own work, or copied from somewhere else.

For completenes, I provide the data on task duration in Figure 4.5. Due to the low number of participants, this data is inconclusive, and not statistically significant.

In addition to time and quality measurements, we also asked participants about the perceived usefulness of Alhazen's diagnosis, and test cases. Data for the RemoveHTML task can be found in Figure 4.6. The questions about usefulness of tests and diagnosis respectively were only asked to those participants who saw tests or diagnosis, or both. Only those participants were considered for the calculation of mean, median and standard deviation. As for task duration, the number of participants is too small to draw any conclusions from this data. The task was in general perceived as easy, and there was no problem in understanding the diagnosis. This data gives no reason to question the usefulness of our questionnaire.

> The questionnaires provide meaningful information.

Task 2: uname

Within the uname task, participants were asked to fix a piece of code which extracts operating system information, like version number, kernel name and architecture, from a string. For this task, I was able to use data from 10 participants in the d0t0 group, 4 for d0t1, 9 for d1t0 and 8 within d1t1.

As for the previous task, I am going to discuss the additional remarks made by the participants. Doing so quickly shows a problem with the task. 11 participants reported problems related to regex. One participant even made a rather puzzeling statement:

1. The bug, if I found correctly, was in the regex expression and not in the program!

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This indicates that they do not think that the regular expression is a part of the program. I consider this a misconception, similar to a statement like "The bug was not in the program, but in the source code (of the program)".

Then, some participants indicated the use of external tools:

2. Didnt have much experience on Regex strings. But after reading a bit, I check online tools to test the given regex string and I could see that it doesnt matches with the input. I tried to fix the regex string, but the program still doesnt work.

3. I haven't used regex in ages and used https://regexr.com/ to better understand the one presented

4. using classic external regex101 to test regex is faster than try/error

5. At first I didn't notice that the minus was wrongly inside the capture group. I used print to find out why the RELEASE assertion failed and noticed it immediately afterwards.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This is understandable, as those tools provide better insights into the functioning of a regex than traditional debuggers. However, it may mean that the data we collect is not about Alhazen, but about those external tools.

The largest group, however, are those participants who expressed an unfamiliarity with regex:

6. I am familiar with regex but not much with Python regex. I think that the difficult in identifying this bug comes more from knowing or not knowing Python regex rather than other issues

7. I am not used to this regex match

8. I had to look up everything about regexes b/c i *m not familiar with regexes.

9. Regex refresher was required, especially how to use + and ?

10. Had a hard time with regex! Quite confusing at times.

11. It was easy to figure out that something was wrong with the regular expressions, but resolving it takes some familiarity of it.

12. Just testing the limits of my regex ability really. Also the correct behavour is not completely unambiguous which makes things a bit tricky.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This shows an unfamiliarity with the technology: + and ?, which one participant reported as unfamiliar, are basic operators, and, most likely, the very first concepts any training material about regexes mentions.

> Make sure participants know all concepts used in the task's code.

However, the last participant points to a problem that became more evident when I examined the submitted solutions: The task did not specify how the code should handle some situations. While, as I explained in my discussion of the RemoveHTML task, some amount of vagueness is acceptable, this task did not describe how the code should handle invalid inputs. Several participants provided code which would accept malformed version strings, however, there is no agreement on what a version string looks like between different kernels [5], so the question what is malformed is debatable already. Then, with no information about how the code should react in this situation, the participants could do nothing but ignore this. Participants who accepted malformed strings still achieved a quality score of 1.0. It is not easy to add tests which fail with those repairs. They would test for expectations that the participants were not aware of. I would need to describe the expected behavior in more detail. But then, it becomes much easier to spot the bug.

> The tasks within the pilot study where hard to solve, as the expected behavior was not always obvious, or fully specified.

All quality data can be seen in Figure 4.7. There are 7 different values for this task, which is a lot more variety than for the previous task. However, it is unclear whether this actually means that the task gave enough opportunities for incomplete fixes, or is just an artifact caused by the participants' unfamilarity with regex.

Two participants indicated that they found the task more difficult than the previous one:

13. It took time to find which expression was causing the errors.

14. It took me a bit longer to solve it for the other test cases too, the first test case was easy to solve. but the rest were kind of tricky

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This also reflects in the data from the questionnaire, which can be found in Figure 4.8. The statement "This task was very easy" met more disagreement than for the previous task.

Finally, some participants expressed their opinions on Alhazen:

[5] e.g. the Darwin kernel of Mac OS X has other conventions than the Linux kernel

One participant concludes that more test cases, or examples, are more helpful than Alhazen.

15. examples would have been better

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This is reflected by the replies to the questionnaire: The participants agree or strongly agree with the statement that the additional tests were useful.

Another participant feels that the diagnosis was wrong:

16. The diagnosis detected a too short version string instead of a missing second dot version as the problem. Also, it did not mention that the minus (as in "-RELEASE") was also a problem

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Alhazen's diagnosis indeed indicates that the length of the version string is the problem. The point is that within the training data, and within all examples generated by the feedback loop, only long version strings have a second dot in them. This is mandated by the grammar: A dot can only appear after three characters.

The last two point to another problem with the task:

17. I suppose there were two bugs in the code and Alhazen's hint was only useful to find one of the two

18. Alhazen lead me to the first of the two bugs.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Indeed, there were two problems in the task: The code failed on a patch version after the minor version number (that is, a second dot in the version number), and it failed, because it added a minus to its output in some situations.

The diagnosis and the test cases should be about one malfunction only.

This also reflects in the questionnaires: The usefulness and understandability of the diagnosis was rated as mostly neutral.

For completeness, I give data on task duration in Figure 4.9. This data is not statistically significant.

Task 3: RPN 2

The RPN 2 task is a calculator which accepts inputs in reversed polish notation. The bug is that it misses operators after numbers if there is no space in between. The Alhazen diagnosis is:

1. The program fails if: The second operand in an expression is not followed by a whitespace, and the input is longer than 3 characters.

The first part is correct. The second part, the restriction on the input length, is true because I need an expression with an operator, and therefore at least two operands (1 character each, separated by a space), and an operator. The smallest possible input which triggers this bug has 4 characters. The diagnosis would be easier to understand, if ALHAZEN were to point out that an operator is required, however, the structure of the grammar means that there is no exists(Operator), but only four exists() for +, *, \and - respectively. ALHAZEN delivers a shorter explanation by (ab-)using the coincidental correlation of max-char-length(input) with the existence of an operator.

For this task, I have usable data from 10 participants in the d0t0 group, 5 participants in d0t1, 7 participants in d1t0 and 6 participants in d1t1.

As for all other tasks, this means that the data is not statistically significant. I am therefore looking at additional remarks only.

As with the previous task, some users express that they are unfamiliar with the topic.

1. A refresher on RPN would be helpful
2. didnt get it

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Two others also point out that this task is difficult for someone who does not know reversed polish notation.

3. lucky I know how stacks & RPN calculator work :)
4. I was familiar with the topic -> easier

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

For this task, one participant reported a technical problem:

5. Somehow the "run test"-button disappeared. Hopefully the passing result still came through (the pos+1 in parse_num was the culprit)

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

I can confirm that this participant submitted a working – and correct – solution. Still, the infrastructure may need some more testing. However, the problem of setting up this tool is not an easy one: All participants use different browsers or operating systems, and even if the code is flawless, network problems may still be a source of problems. Therefore, I have to accept that some datapoints are not useable due to technical problems in the end. In addition to this, some participants need to be dropped because they had interruptions, or trigger the outlier threshold (see Section 4.2.6). This has consequences for the required number of participants.

> The number of participants invited should be sufficient to be able to discard some participants due to technical or other problems.

For the RPN 2 task, one participants commented on the correct solution.

6. The solution was a double incrementation of pos in parse_num.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This statement is correct. There was a double increment within the code to parse numbers, skipping the character immediately after the number.

As with the RemoveHTML task, this was spotted by most participants. Which, again, leads to a sparse quality landscape. Figure 4.10 shows just two quality levels. Just as the RemoveHTML task, this task does not give enough possibilities to make mistakes.

Still, two participants expressed problems with this task:

7. It's quite difficult for me to comprehend the debugging for this specific question. No matter how much I try to understand it, Whoever wrote this code isn't to be allowed near corporate environments as a developer, it's quite poorly written. I'm sorry no offence but i couldn't understand it as an intermediate developer.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

I am in no position to judge the code quality, as I wrote it myself. However, just one other participant expressed similar concerns, and I myself consider the code probably not my best work ever (I had to cause a bug on purpose, after all), but reasonable.

The second participant wrote:

8. I took me a while to parse the code but after I used print to see what is pushed to and popped from the stack, I found the issue quite quickly when I noticed that the result of the addition was not pushed to the stack.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

This sounds less like a complaint about the code, and more like a story about debugging. The participant reports the use of `print` statements, as a debugging helper. As this participant was in the d0t0 group, this cannot be seen as a statement on the quality of the diagnosis or tests.

However, participants did express their opinions about either of those:

9. I could not understand the diagnosis at first, but when i completed the task it made more sense

10. That would have taken forever without the test cases

11. I am not sure what was meant by the input length restriction

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

The second participant reports that the test cases were helpful. As the third participant points out, the input length restriction is an artifact of Alhazen's algorithm: The shortest possible behavior triggering input has 4 characters, and for Alhazen, using this coincidental correlation is the easiest way to express this. Maybe the comment by the first participant, that they only understood the diagnosis after fixing the bug, points in the same direction: When the developers do not know the grammar and are unaware of the concept of coincidental correlations, diagnosis which contain one are confusing for them.

Finally, the last two participants report on their debugging strategies:

12. Using breakpoints here would be helpful, although i understand that jupyter notebook is useful for coding in the browser

13. The buggy subroutine was the last one I checked

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

One of them points out that they were looking for traditional debugging tools, the second one just has a classical debugging story. The bug is always in the last subroutine one looks at[6].

For completeness, Figure 4.11 and Figure 4.12 give the task duration and questionnaire results for the RPN 2 task. This data is not statistically significant. The questionnaire data indicates – as for the previous task – that participants were more happy with the tests than with the diagnosis.

Task 4: Lists

Within the Lists task, participants were asked to debug a piece of code which sums up a list of numbers, given as a string.

For this task I had 10 participants within the d0t0 group, 5 participants within the d0t1 group, 7 participants within d1t0 and 6 participants within d1t1.

As for the other tasks, the number of participants per group is slightly unbalanced. The reason for this are random effects: The conversion rates were slightly different between the groups (but the difference is not statistically significant), and outliers and interruptions happened at different rates within the groups. For such a small sample size, this has to be expected.

Within our infrastructure, participants were assigned to a group when the invitation to the second part was sent. Unfortunately, the system cannot know the conversion rate per group, or how many data points will be discarded, at this point in time. Therefore, I suggest to assign later: If the assignment only happens when a participant accepts the invitation, the effect of different conversion rates can be balanced. If the system can even check whether data is usable, it could just assign another participant to a group whenever a data point is invalid.

> The infrastructure for a user study should be able to assign participants to groups as late as possible.

However, all those effects are purely random, therefore a higher number of participants may balance this automatically.

For the Lists task, 6 participants provided an additional remark.

One participant wrote:

1. so strange ways of doing simple stuff :) but fun!

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Thinking about this, I, as someone who deals with parsing every now and then[7], found the way how the code works straight forward. After reading this comment, I realized that it would have been easier to split the string at spaces

[6] That is, mostly, because one stops looking after finding it

[7] e.g. in two chapters of this dissertation

with a library function and then convert each part afterwards. I guess everyone is influenced by his training and experience. However, this may provide an insight into task design.

> Tasks should be reviewed by an experienced developer other than the person who designed them, to check for code quality and obvious issues.

One participant reports a technical problem:

2. A warning that the first run takes forever would have been nice. For a few minutes I wasn't sure whether the server died on me

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

There was no such warning because it never occured in my testing. I suspect that the server may have been overloaded occasionally, or that the connection may have been unstable. As time-taking is implemented client-side, the results are still valid. However, long execution times may lead to higher-than-necessary task durations.

> During the study, the server performance should be monitored.

However, as mentioned before it will be impossible to rule out all technical difficulties.

One participant, again, reported on using a different debugging tool:

3. I printed the number before yielding it and then found the issue quickly.

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Also, several participants reported satisfaction with the Alhazen diagnosis.

4. This time, the diagnosis was quick and easy to understand
5. This was easier thanks a lot for this. I'm just an intermediate cse junior year student and this seemed doable because of the diagnosis
6. Alhazen's diagnosis directly pointed me to right function to check and from there the bug was easy to notice

All participant remarks are printed as given by the participants, including all spelling and grammar errors.

Alhazen is perceived as useful on this task. This result also reflects in the questionnaire results, shown in Figure 4.13. Perceived usefulness of both the diagnosis and tests, as well as readability of the diagnosis, are excellent. Still, those results are not statistically significant.

Figure 4.14 indicates another good property of this task: It has six different quality levels.

One participant achieved a quality of 0. They changed the correct function, but their fix does not even make sense syntactically. There is one execution path where the function does not have a return value, despite the fact that it is supposed to have one. Many programming languages would consider this a syntactical mistake, and refuse to compile.

All participants who achieved a quality of 0.5 made a rather curious mistake: The original code would extract numbers from the input string, and use Python's builtin mechanism to convert those strings to integers. The code, however, did not account for negative numbers. A leading minus sign lead to a parsing error. Participants who achieved a quality of 0.5 double-compensated. They checked for a leading minus themselves, and then multiplied the number by -1, in an attempt to make it a negative number. However, after extracting the entire number from the string, Python's builtin mechanism handles the minus quite readily. Therefore, the multiplication by -1 re-introduced the same problem.

The participant who achieved a quality of 0.75 re-wrote the entire number-parsing, and missed the fact that numbers can have more than one digit. This is interesting: For the uname task, I observed that some solutions were wrong, but not considered as such, as the expected format of the inputs was unclear, and therefore it was not clear whether this was indeed a mistake by the participants. For this task, the input format is decimal numbers. A format every developer is familiar with. Therefore, I can consider this solution wrong, despite the fact that all test cases the participants saw, even within the t1 groups, had only single-digit numbers.

The test cases contained another hidden gem: Numbers cannot only start with minus sign, but also with a plus sign. Given that there was no test cases which indicated this, it might be over-the-top to expect that developers realize this when fixing such a bug. Still, developers who achieved a quality of 0.92 fixed the problem correctly, but did not extend the code to support numbers prefixed with a plus as well. Many of them also parsed things like `"1-9"` as a single number, which would fail when Python's builtin mechanism was used to convert those numbers. However, I did not consider this an error, as the task description did not specify whether numbers were required to be separated by spaces, and how invalid numbers should be treated.

Finally, several participants achieved a quality of 0.83. This value could be achieved in different ways: Two participants supported single-digit numbers only, just as the one with a quality of 0.75, but realized that numbers cannot only start with a - sign, but also with a + sign. One participant fails with a weird error message on numbers prefixed with two minus signs. I did allow participants to consider this an error and throw an exception, however, this participant generated an error which my oracle code did not handle.

All in all, this is the task with the most diverse quality of fixes.

> Use tasks with input formats that are even more well-established than HTML.

For completeness, Figure 4.15 gives the task duration for this task. The data is not statistically significant.

4.2.7 P-Hacking

As mentioned before, none of the results of the pilot study are statistically significant. However, this may not only depend on the data, but also on the analysis deployed. All analyses I deployed so far were decided on up-front, that is, before the first participant started the study. In this section, I will do some analyses that were not defined up-front. This is scientifically unsound.

The problem is that almost any dataset contains some seemingly interesting correlations or observations. However, many of those are just random effects, and not generalizable to other data sets. Therefore, reporting observations in data that were not expected beforehand runs the risk of false positives. With this caution in mind, I will engage in this type of analysis here.

Are some participants just slower?

One noteworthy point about the data collected in the study is the huge standard deviation for task duration. Within several groups and task, the standard deviation is so large that some participants seem to solve the problems twice as fast as others. Within the pilot study, I also collected the time taken for the three programming questions within the screening test. Participants required on average 893.63 seconds to solve those programming questions, with a standard deviation of 619.03. Therefore, the distribution of screening test duration looks quite similar to the durations observed for the tasks. It seems that some developers are just faster than others all the time. This is in line with earlier results on individual differences in performance on debugging problems[62]. However, if this holds, participants who were fast in the screening test should be fast for the tasks just as well. Statistically speaking, there should be a correlation between screening test duration and task duration.

I checked for two different correlations: Pearson's correlation coefficient[56] is a value between -1 and 1 which assumes its extreme values if there is a linear relationship between two variables. Spearman's rank correlation coefficient[66] is between -1 and 1 as well, however, it assumes its extreme values if there is a monotonic function that describes the relationship between two variables. Therefore, I check for both, linear and non-linear relationships.

It is likely that participants in the d1 and t1 groups are somewhat slower, as they need to process additional information, the diagnosis or test cases respectively. Therefore, I checked correlation for each task and each group. The full data can be found in Table 4.13. The table gives the number of participants per group and task in its first column. Those numbers are small, and correlation coefficients are unlikely to be statistically significant. However, the mean of the Pearson's correlation coefficients is 0.40, with a standard deviation of 0.46. This is too small to assume a linear relationship between screening test duration and task duration. Spearman's correlation coefficients are 0.37 on average with a standard deviation of 0.35. Therefore, I cannot assume a non-linear relationship either. Those results do not support the hypothesis that individual developers just are working at different speeds.

> Time taken for the screening test has no relationship to time taken on the tasks.

Does more time spend lead to better quality?

Reading and understading test cases or a diagnosis takes time. But then, if the participants spend additional time thinking about the problem, do they deliever higher quality? If this is the case, there should be a correlation between quality and task duration.

Task	Group	# participants	pearson	spearman
RemoveHTML	d0t0	12	0.17	0.43
RemoveHTML	d1t0	9	0.91	0.40
RemoveHTML	d0t1	9	0.87	0.63
RemoveHTML	d1t1	10	-0.10	0.21
uname	d0t0	11	0.05	0.29
uname	d1t0	9	0.62	0.28
uname	d0t1	9	0.91	0.83
uname	d1t1	8	-0.06	0.17
RPN 2	d0t0	11	0.04	0.24
RPN 2	d1t0	8	0.98	0.95
RPN 2	d0t1	8	0.86	0.90
RPN 2	d1t1	7	-0.14	-0.21
Lists	d0t0	11	0.41	0.05
Lists	d1t0	9	0.89	0.25
Lists	d0t1	9	0.49	0.75
Lists	d1t1	8	-0.36	-0.12

Table 4.13: Pearon and Spearman Rank correlation coefficient for screening test and task duration.

Task	Group	# participants	spearman
RemoveHTML	d0t0	11	-0.37267799624996495
RemoveHTML	d0t1	8	-0.2519763153394848
RemoveHTML	d1t0	9	-0.2738612787525831
RemoveHTML	d1t1	10	0.4107919181288746
uname	d0t0	10	-0.4061811972299616
uname	d0t1	5	0.2581988897471611
uname	d1t0	9	
uname	d1t1	9	0.6468665190255972
RPN 2	d0t0	10	
RPN 2	d0t1	5	
RPN 2	d1t0	7	-0.6123724356957945
RPN 2	d1t1	6	0.3927922024247863
Lists	d0t0	10	0.24219967151691824
Lists	d0t1	7	0.6424452109375584
Lists	d1t0	8	-0.6546536707079771
Lists	d1t1	7	0.40768712416360564

Table 4.14: Pearon and Spearman Rank correlation coefficient for quality and task duration.

Table 4.14 shows this data. Some table cells stay empty: In those cases, the correlation coefficients could not be calcuated, because the quality data did not have enough levels. Here, I used spearman's rank correlation coefficient, because quality is ordial, rather than rational. Most of the correlation coefficients are small. There are just three outliers: The d1t0 shows a moderate to strong *downhill* effect for the Lists and RPN 2 tasks. For those tasks, participants who took more time delievered lower quality. However, this result is based on just 8 and 7 participants respectively, and cannot be seen as a larger trend. The third outlier is d1t1 on unname, which shows the expected effect. However, with 15 experiments who do not, it is safe to say that this is not a reliable result. Moreover, just 7 out of 16 experiments show the expected trend. All in all, this data is inconclusive, but discouraging. The low number of participants means that all of this could be random effects.

Enlarging the data set

In the pilot and pre-pilot, the RemoveHTML task was identical. This means that the data for this task could be combined into one, larger data set. Scientifically, this is a venturous endeavor. It is only valid if we recruited from the same populations for both studies. However, the screening test results between pre-pilot and pilot were qute different, indicating that we did not. Therefore the data may be incompatible.

Combining the data set nevertheless, I have data from 66 participants. After cleaning the data as described in Section 4.2.6, 52 participants remain. 19 out of those participated in the prepilot, the remaining 33 data points were collected in the pilot study.

Figure 4.16 shows the combined data for task duration. Obviously, the most relevant question is whether any of this is statistically significant. Figure 4.17 gives, among other data, the p-values for each comparison. Those values were calculated with a standard t-test. Still, no comparison is statistically significant.

Let's have a look at repair quality. The data can be found in Figure 4.18. I used a chi^2 test to calculate p-values and effect sizes. Again, no comparison is statistically significant.

Estimating the required number of participants

In Section 4.2.4, I used data from the pre-pilot to provide an estimate on the required number of participants. In this section, I'll repeat the same calculations, using the combined data from pilot and pre-pilot study for the RemoveHTML task.

The sample size estimates for task duration can be found in Figure 4.17. Numbers are in between 72 and 7748. This, again, means that the recruitment strategy used in the pilot study would not have been capable of generating enough data.

Figure 4.19 shows the p-values and sample size estimates based on quality. In contrast to Section 4.2.4, I calculated this based on quality only, and not based on the frequency of successful repairs. Please observe, from Figure 4.18, that all but the d0t0 and d1t1 groups have just two levels of quality nevertheless, and therefore the result is identical to the one I would get with a frequency

comparison. As mentioned before, none of the results is statistically significant as is. Comparisions require in between 422 and 898 participants.

> The original sample size estimation, 50 participants per group, was too low.

4.3 Proposed Design for a User Study

At first glance, this chapter of my dissertation reports on a failure: I provided no insights on the usefulness of Alhazen as a debugging tool. The data is not statistically significant, and even if it were, the effects are negligible. However, this chapter does provide insights into why the user study was insufficient to show the usefulness of Alhazen. Therefore, I will use the lessons learned to draft a study that would be capable of answering my research questions about Alhazen. This study can be carried out in future work.

4.3.1 Recruitment Strategy

In Section 4.2.5, I observed

> For reference
> The currently employed recruitment strategy does not generate the required number of participants.
> See Section 4.2.5

Therefore, the recruitment strategy needs to be changed. Platforms like Fiverr[8] or upwork[9] allow for the hiring of freelance software developers. As I already gave amazon vouchers to the participants, the study would not be more expensive if such a service is used to recruit participants. Also, it stands to reason that platforms which are used to hire developers in the first place, and tasks which are similar to the usual offers on those platforms, attract developers more easily. Therefore, I suggest to use such a platform for participant recruitment.

When doing so, another observation has to be considered:

> For reference
> The original sample size estimation, 50 participants per group, was too low.
> See Section 4.2.7

This is alarming, as it would be close to impossible, even on Fiverr or upwork, to recruit large groups of participants. However, I am going to suggest significant changes to the task design in the next section. For those new tasks, the sample size estimates may not be applicable. Therefore, I suggest to conduct another pilot study, and re-calculate the sample size estimates with values from the new pilot.

When deciding on the number of participants for the study, there is another factor to consider:

[8] https://www.fiverr.com
[9] https://www.upwork.com/

For reference

The number of participants invited should be sufficient to be able to discard some participants due to technical or other problems.

See Section 4.2.6

Table 4.12 shows how many participants were removed as outliers or because they reported interruptions for each task. I removed between 6 and 10 participants, that is between 15% and 26% of all participants. Therefore, about a 25% more participants than estimated should be invited to participate. This does not consider the number of participants who were invited, but did not take part.

4.3.2 Within Subject versus Between Subject

In the pilot study, participants were assigned to a group, and this assignment meant whether they saw test cases or a diagnosis for all tasks. This is a "between subject" design, as subjects are compared to each other. However, huge standard deviations may result from individual differences between developers. Therefore, I suggest to switch to a "within subject" design. In this design, a participant receives diagnosis and tests for some tasks, but not for all. Then, their performance on different tasks can be compared. This means that individual differences can be ignored – they affected performance on both tasks – but complicates task design: If one task is substancially simpler than another task, this may hide effects from Alhazen. Therefore, it should be randomized which treatment is provided to which participant on which task.

4.3.3 Task Design and Screening Test

Even after reworking the tasks for the full study (see Section 4.2.4), there were some problems with the tasks. Those were:

For reference

Test cases and task descriptions within the user study may provide an incomplete specification.

See Section 4.2.6

For reference

The tasks within the pilot study where hard to solve, as the expected behavior was not always obvious, or fully specified.

See Section 4.2.6

In other words: The level of detail given in task descriptions needs to be balanced carefully. Too much details may render Alhazen's diagnosis superfluous. Also, real-world debugging tasks seldomly come with multi-page documents about expected program behavior. Too little details makes it hard to judge participant submissions for quality, as it is unclear whether a problem is a bug, a misunderstanding or this behavior was considered unspecified and therefore irrelevant by the participant.

Then, there is another requirement for the tasks:

For reference

Tasks should be designed to allow for incomplete fixes.

See Section 4.2.6

I therefore suggest a completely different setup: Participants should receive a rather large program, in the 1000 to 3000 lines range. Then, they are asked to debug five bugs within this program, one after another.

1. A rather simple bug, which forces them to review large parts of the code base (warm-up task)
2. One bug where they receive just one test case (similar to the d0t0 group in the pilot study)
3. One bug where they receive Alhazen's test cases (similar to the d0t1 group in the pilot study)
4. One bug where they receive Alhazen's diagnosis (similar to the d1t0 group in the pilot study)
5. One bug where they receive Alhazen's diagnosis and test cases (similar to the d1t1 group in the pilot study)

It should be randomized which bugs is presented with which treatment to which participant.

With this design, the buggy code is presented in a larger context. Parts of the specification can be inferred from this context, which means that we can test for precise behavior, while at the same time not provide too much details in the task description. Also, a larger context makes it simpler to provide bugs which allow for incomplete fixes.

While designing this larger program, and three to four tasks in the context of this program, the other insights from pilot and pre-pilot should be considered. Those were:

For reference

The diagnosis and the test cases should be about one malfunction only.

See Section 4.2.6

For reference

Make sure participants know all concepts used in the task's code.

See Section 4.2.6

For reference

Tasks should be reviewed by an experienced developer other than the person who designed them, to check for code quality and obvious issues.

See Section 4.2.6

For reference

Tasks should not be publicly available before the study is started.

See Section 4.2.6

However, using a larger program likely creates a new problem: When participants work on the first task, they do not know the code base well. Therefore, this task likely takes longer than usual: The time includes time that is used to learn the code base. Subsequent tasks on the same code base tak less time, because participants know the code base already. This learning effect may cloud the influence of ALHAZEN on the time that participants need to fix the bugs. My suggested study design counters this effect in two ways. Firstly, the first task helps participants to learn the code base, but is not measured. Therefore, the second task already profits from learning, and the learning effect has less influence on differences between tasks 2, 3, 4 and 5. Second, the order of tasks 2 to 5 is fixed, but it is randomized which task receives which treatment: Some participants get test cases and diagnosis for the first bug, while other have just one test on the first bug, but have the diagnosis available for one of the other problems. This way, tasks which profit stronger from learning effects have the treatment for some, but not all participants, which allows for compensation of the learning effect statistically.

I also observed:

For reference

The call for participation reaches and motivates professional developers only, therefore the screening test is mostly pointless.

See Section 4.2.5

When recruiting on freelancer platforms, it is very likely that we again recruit professional developers only, therefore I assume that no screening test is necessary. However, the warm-up task can at the same time be used to identify low-performing participants.

Lastly, there was an observation about cheating:

For reference

In the current design, we cannot reliably detect whether participants submitted their own work, or copied from somewhere else.

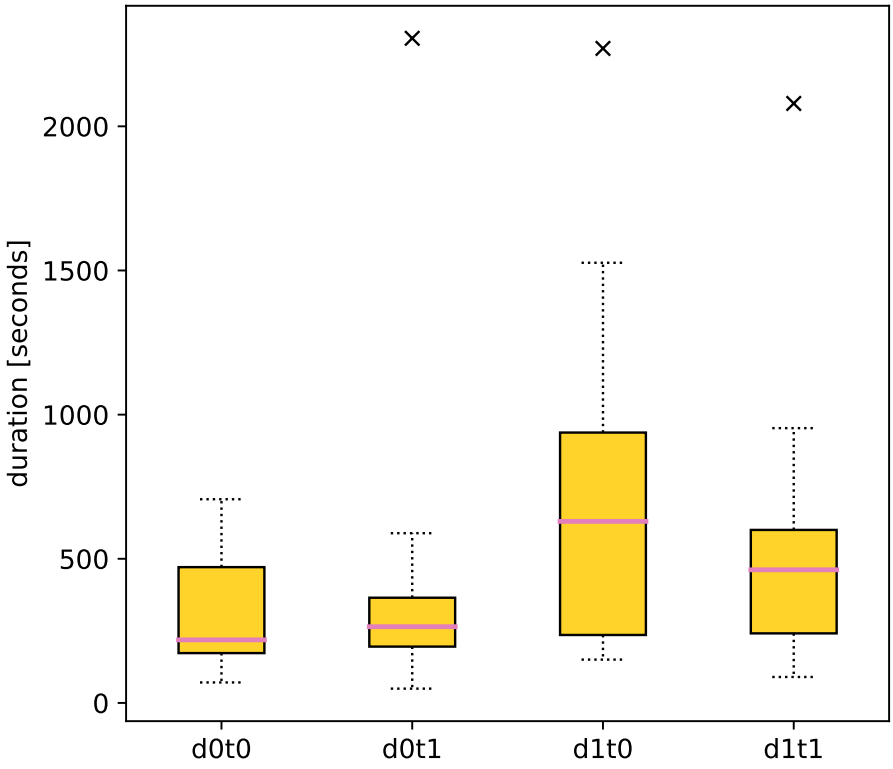
See Section 4.2.6

Small syntactical changes, like randomizing function order, or renaming variables and functions can be applied to the code. This way, participants who outright copied from someone else can be identified. Also, a larger program makes it less likely that fixes made by different developers are absolutely identical. This simplifies the detection of copied solutions.

4.4 Conclusion

In Section 4.1 I showed how developers can focus on smaller parts of the input space, thanks to ALHAZEN. Unfortunately, those advantages did not materialize

in the user study. However, shortcomings in the design of the study, most notably the insufficient number of participants, are likely to be the cause. Therefore, I presented an improved study design in Section 4.3. I am hopeful that this design, if implemented, provides more usable data on ALHAZEN's merit.

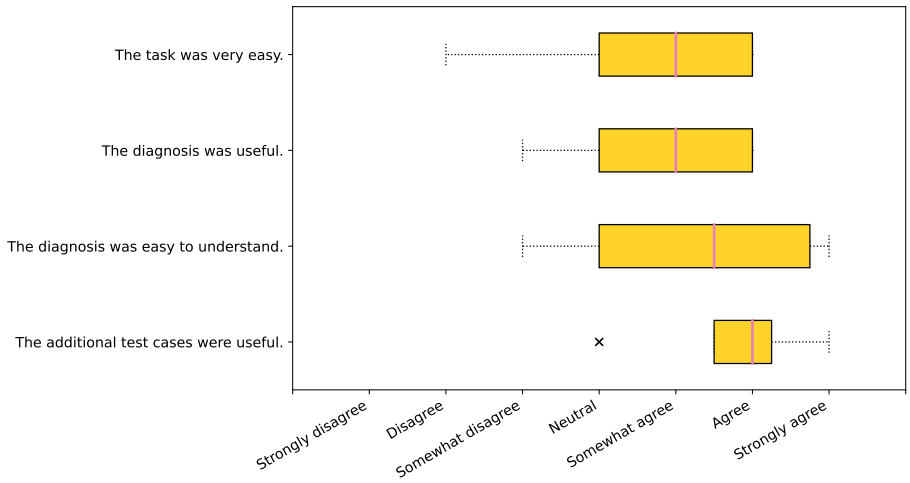


(a) A box plot of task duration for the RemoveHTML task within the pilot study.

group	# participants	median [s]	mean [s]	std
d0t0	11	218.78	318.41	210.05
d0t1	8	264.47	507.84	745.10
d1t0	9	629.73	816.37	695.74
d1t1	9	461.75	596.73	616.63

(b) A statistics on task duration for the RemoveHTML task within the pilot study.

Figure 4.5: Task Duration within the RemoveHTML Task.



(a) A box plot of perceived usefulness of the RemoveHTML task within the pilot study.

question	# participants	median	mean	std
The additional test cases were useful.	8	5.00	4.75	1.16
The diagnosis was easy to understand.	18	4.50	4.28	1.45
The diagnosis was useful.	18	4.00	3.94	0.94
The task was very easy.	37	4.00	3.78	1.03

(b) A statistics on perceived usefulness for the RemoveHTML task within the pilot study.

Figure 4.6: Perceived usefulness within the RemoveHTML Task.

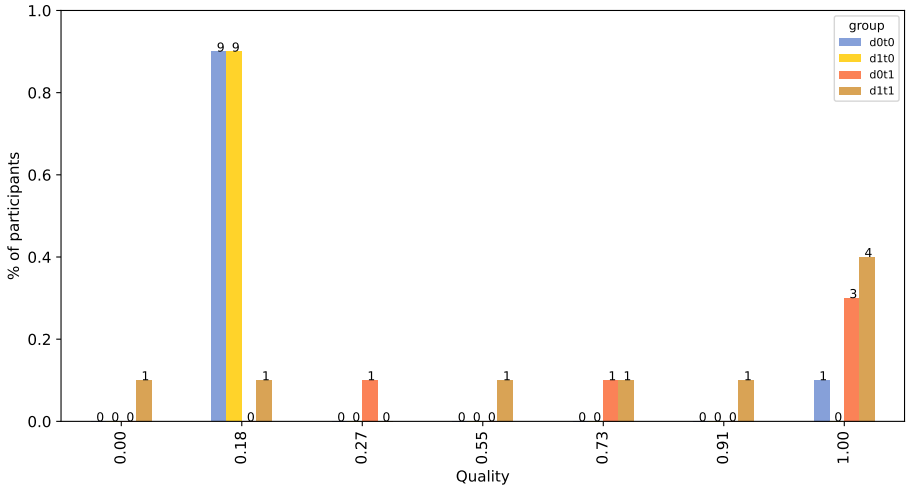
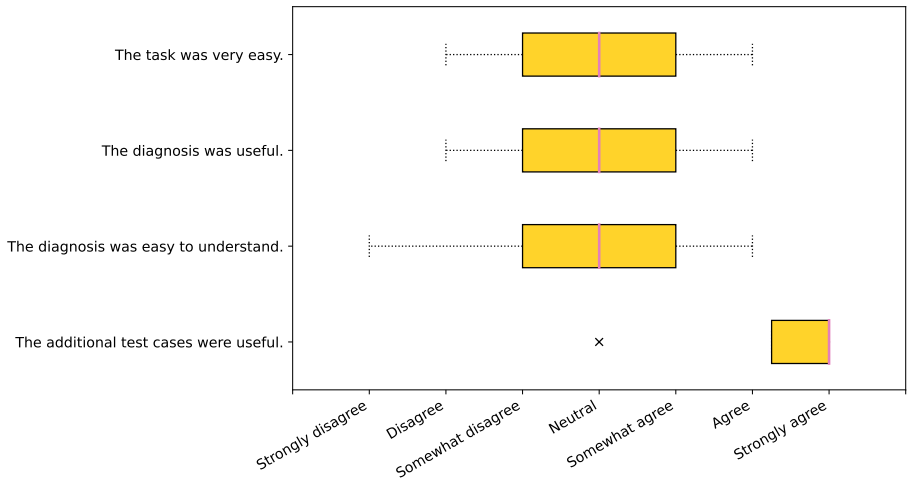


Figure 4.7: A histogram of quality for the uname task within the pilot study. While the numbers on top of the bars give absolute values, the bars are scaled relative to the size of the largest group.

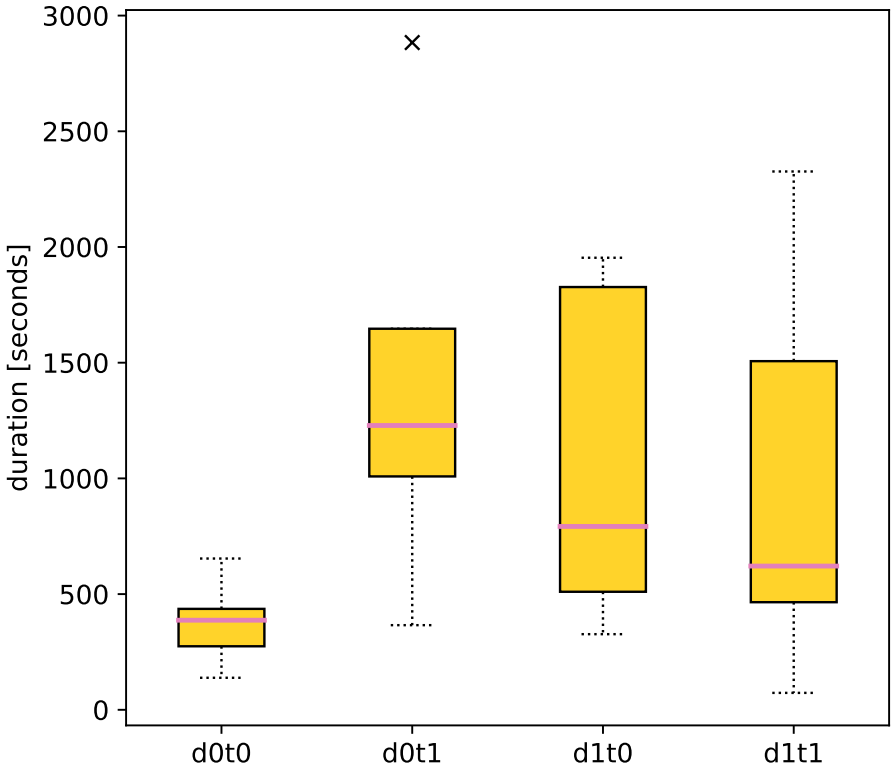


(a) A box plot of perceived usefulness of the uname task within the pilot study.

question	# participants	median	mean	std
The additional test cases were useful.	4	6.00	5.25	1.50
The diagnosis was easy to understand.	17	3.00	3.06	1.56
The diagnosis was useful.	17	3.00	3.06	1.14
The task was very easy.	31	3.00	3.03	1.47

(b) A statistics on perceived usefulness for the uname task within the pilot study.

Figure 4.8: Perceived usefulness within the uname Task.



(a) A box plot of task duration for the uname task within the pilot study.

group	# participants	median [s]	mean [s]	std
d0t0	10	387.18	375.64	146.49
d0t1	4	1228.65	1426.64	1052.99
d1t0	9	792.39	1082.69	676.78
d1t1	8	621.15	983.78	874.47

(b) Statistics on task duration for the uname task within the pilot study.

Figure 4.9: Task Duration within the uname Task.

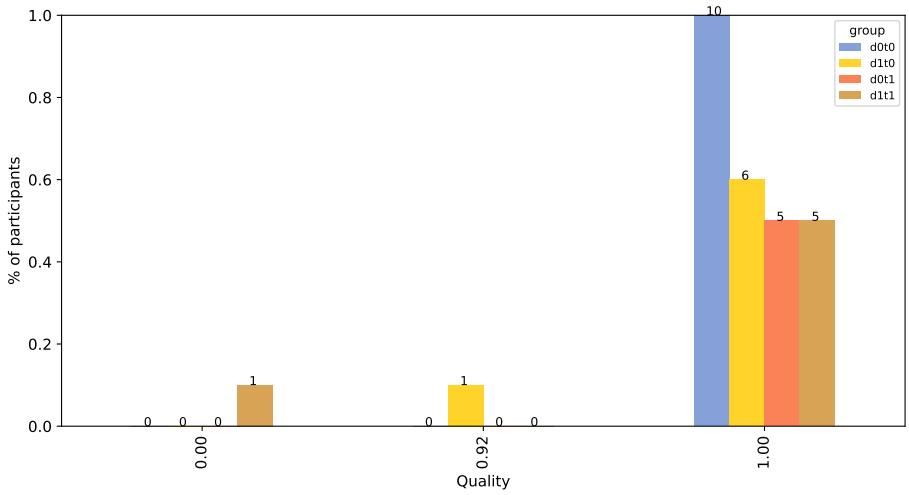
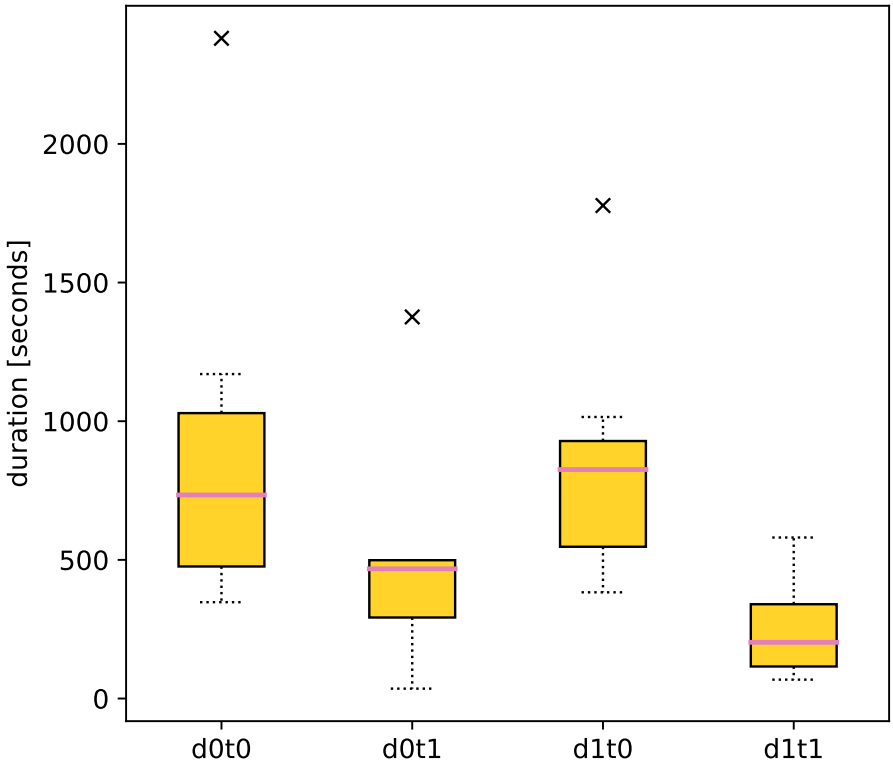


Figure 4.10: A histogram of quality for the RPN 2 task within the pilot study. While the numbers on top of the bars give absolute values, the bars are scaled relative to the size of the largest group.

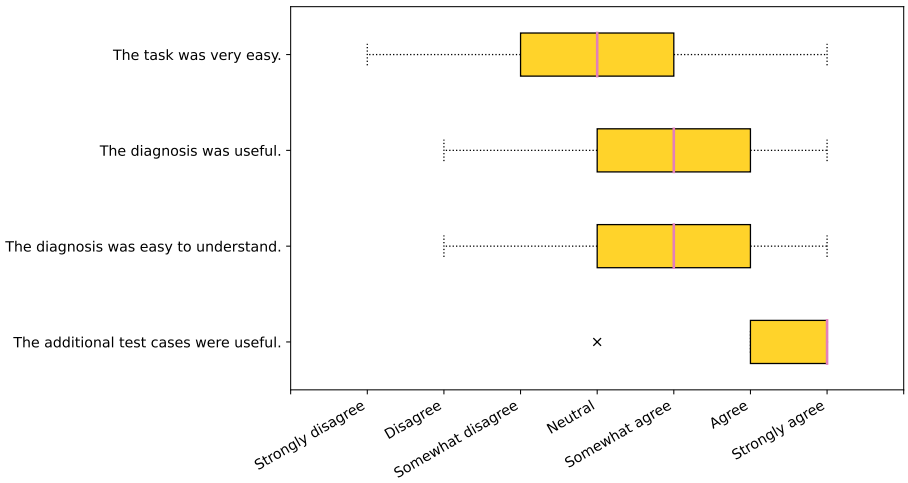


(a) A box plot of task duration for the RPN 2 task within the pilot study.

group	# participants	median [s]	mean [s]	std
d0t0	10	734.04	883.12	597.74
d0t1	5	467.75	534.07	505.32
d1t0	7	825.73	848.33	463.88
d1t1	6	202.75	254.43	193.95

(b) Statistics on task duration for the RPN 2 task within the pilot study.

Figure 4.11: Task Duration within the RPN 2 Task.

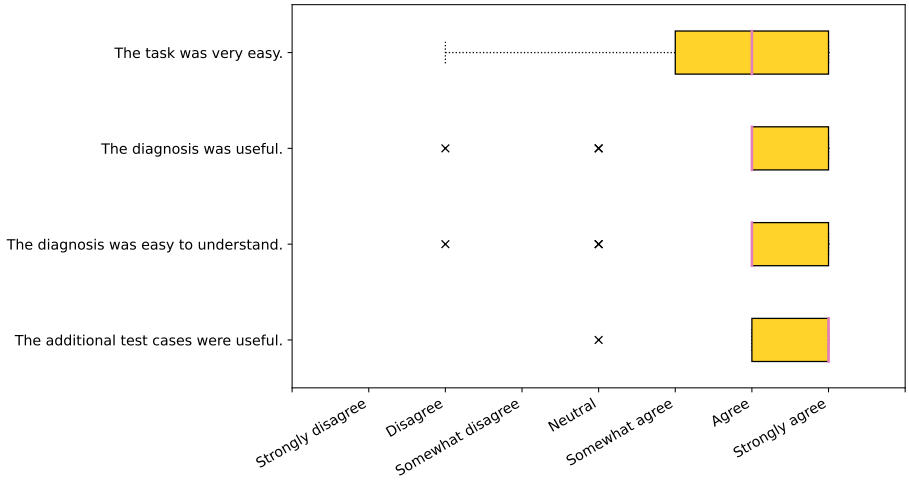


(a) A box plot of perceived usefulness of the RPN 2 task within the pilot study.

question	# participants	median	mean	std
The additional test cases were useful.	5	6.00	5.20	1.30
The diagnosis was easy to understand.	13	4.00	4.15	1.41
The diagnosis was useful.	13	4.00	3.92	1.61
The task was very easy.	28	3.00	3.18	1.39

(b) A statistics on perceived usefulness for the RPN 2 task within the pilot study.

Figure 4.12: Perceived usefulness within the RPN 2 Task.



(a) A box plot of perceived usefulness of the Lists task within the pilot study.

question	# participants	median	mean	std
The additional test cases were useful.	7	6.00	5.29	1.11
The diagnosis was easy to understand.	15	5.00	4.93	1.49
The diagnosis was useful.	15	5.00	4.87	1.46
The task was very easy.	32	5.00	4.78	1.31

(b) A statistics on perceived usefulness for the Lists task within the pilot study.

Figure 4.13: Perceived usefulness within the Lists Task.

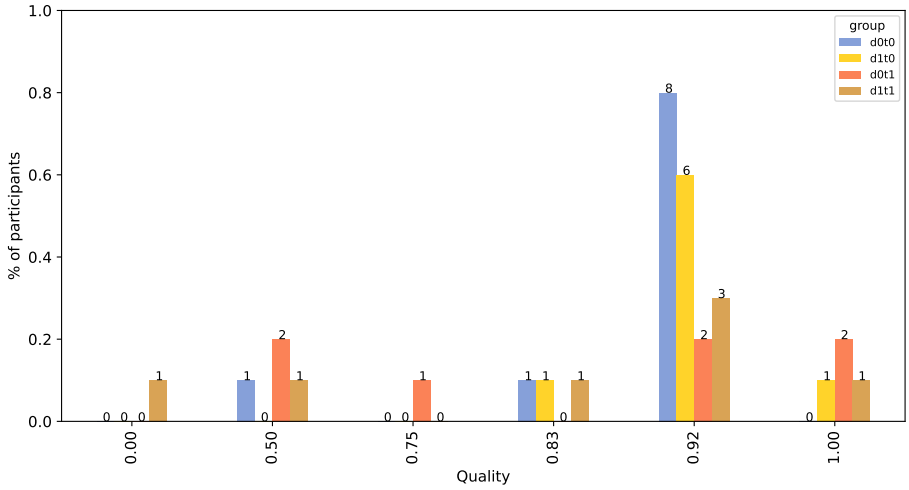
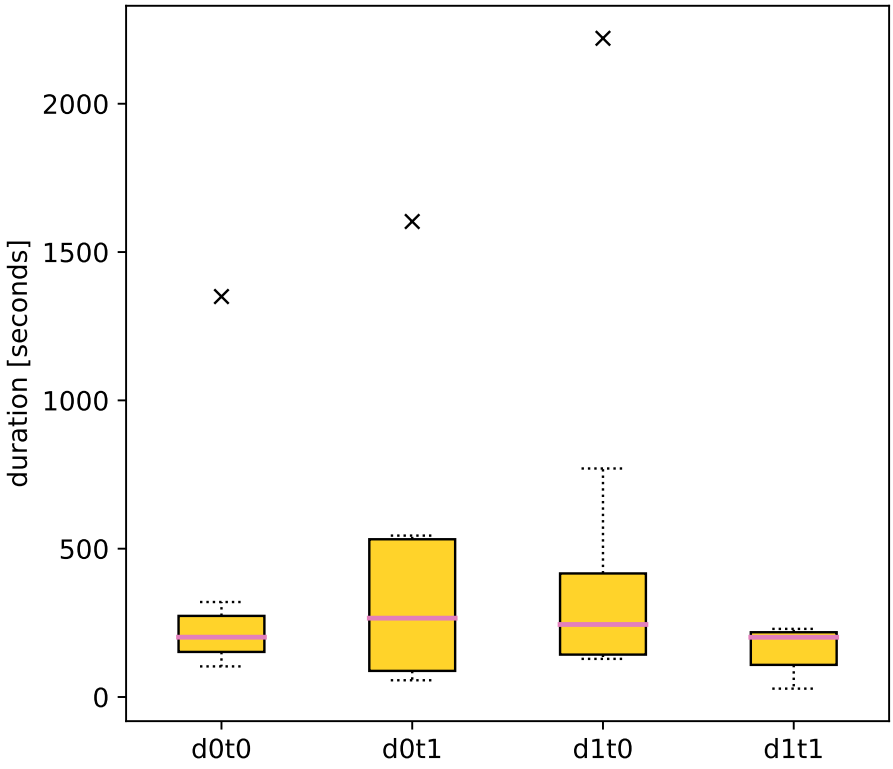


Figure 4.14: A histogram of quality for the Lists task within the pilot study. While the numbers on top of the bars give absolute values, the bars are scaled relative to size of the largest group.

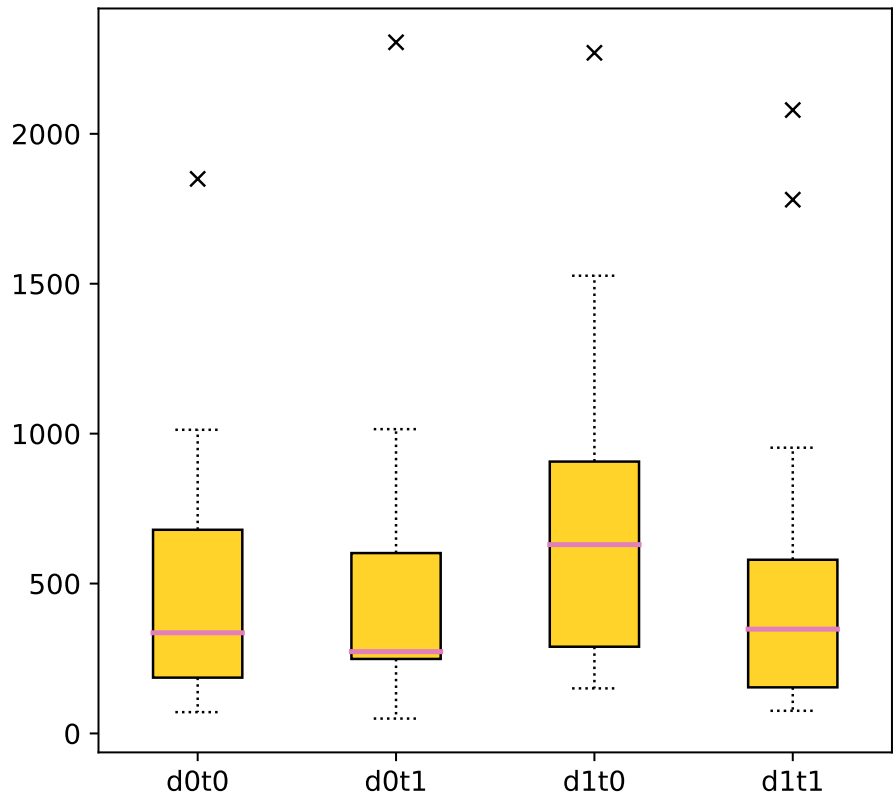


(a) A box plot of task duration for the Lists task within the pilot study.

group	# participants	median [s]	mean [s]	std
d0t0	10	201.66	314.56	370.92
d0t1	7	265.83	452.12	546.84
d1t0	8	244.47	523.98	716.97
d1t1	7	201.39	158.93	77.93

(b) Statistics on task duration for the Lists task within the pilot study.

Figure 4.15: Task Duration within the Lists Task.



(a) A box plot of task duration for the RemoveHTML task, combining data from pilot and pre-pilot study.

group	# participants	median	mean	std
d0t0	14	335.75	516.13	481.60
d0t1	11	272.75	541.35	648.27
d1t0	11	629.73	778.72	639.11
d1t1	14	347.91	571.89	625.79

(b) Statistics on task duration for the RemoveHTML task, combining data from pilot and pre-pilot study.

Figure 4.16: Task Duration within the RemoveHTML Task, combining data from pilot and pre-pilot study.

Group 1	Group 2	p-Value	Effect Size	Description	Power	required # of participants
d0t0	d1t0	0.25	0.47	small to medium	0.20	72
d0t0	d0t1	0.91	0.05	small	0.05	7748
d0t0	d1t1	0.79	0.10	small	0.06	1576
d1t0	d0t1	0.40	0.37	small to medium	0.13	117
d1t0	d1t1	0.42	0.33	small to medium	0.12	148
d0t1	d1t1	0.91	0.05	small	0.05	6802

Figure 4.17: Sample size estimation and p-values on task duration for the RemoveHTML task, combining data from pilot and pre-pilot study.

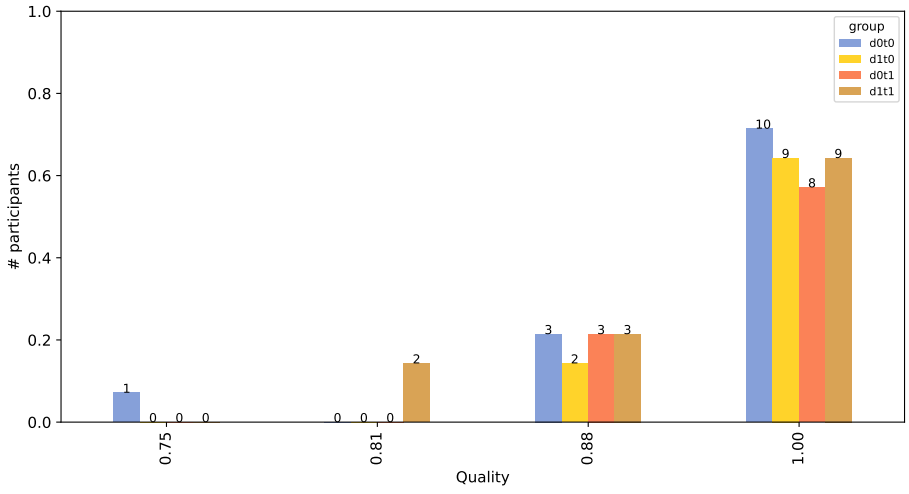


Figure 4.18: The observed quality values for repairs on the RemoveHTML task, combining data from the pre-pilot and pilot studies.

Group 1	Group 2	p-value	Effect Size	Description	# of participants	required # of participants
d0t0	d1t0	0.64	0.13	small	25	868
d0t0	d0t1	0.65	0.13	small	25	898
d0t0	d1t1	0.38	0.19	small	28	532
d1t0	d0t1	1.00	0.00	very small	22	
d1t0	d1t1	0.39	0.19	small	25	422
d0t1	d1t1	0.42	0.19	small	25	456

Figure 4.19: Sample size estimation and p-values on quality for the Remove-HTML task, combining data from pilot and pre-pilot study.

Chapter 5

Targeted Carving

The previous chapters showed how input features can be used for debugging. In this chapter, I will turn towards testing. When I argued why ALHAZEN can be helpful for developers, I made the assumption that developers can concentrate their debugging efforts on a smaller part of the program under investigation when they know which input parts are relevant for a bug. In this chapter, I will investigate whether automated testing tools can focus their effort on specific program parts, by focusing on specific parts of the inputs.

The main idea is to identify a program part which deals with a specific input part, and generate a unit test for this part of the program, based on an analysis of a system test that exercised the entire program. This unit test can be analyzed independent of the surrounding system, and then the result of this analysis can be used to improve the system test.

Of course the system test could be analyzed and improved directly, however, unit test analysis is much faster than system test analysis. Therefore, analysing on the unit-level holds the promise of a performance boost over a system-level analysis. Therefore, I refer to the tool I develop in this chapter as a *system test booster*

The tool, which is called BASILISK, is based on three techniques:

Carving [16] generates unit tests from observations of system test executions. Within BASILISK, I use carving to generate unit tests for the functions that process the relevant input parts.

Unit Analysis Next, I use existing tools to analyse the unit tests. This analysis yields additional unit tests, which describe border behavior of the code under test. Those are the cases that ALHAZEN needs to learn from.

Lifting transforms the unit tests, generated by carving and refined by symbolic execution, into system-level inputs. Lifting was developed by me.

For unit test analysis, I will examine two different techniques: Symbolic execution[8] executes a program symbolically, and therefore builds a set of constraints which describe inputs that reach a specific program part. When my new technique is used with KLEE at the unit level, I call it BASILISK. Fuzzing[49] uses randomly generated inputs to find those which trigger crashes, or achieve new coverage. Combinung my technique with LIBFUZZER at the unit-level, I call it FUZZILISK.

⟨*start*⟩ → ⟨*function*⟩ `"("` ⟨*number*⟩ `")"`

⟨*function*⟩ → `"tan"` | `"cos"` | `"sin"` | `"sqrt"`

⟨*number*⟩ → `/-?[0-9+](.[0-9+])?/`

Figure 5.1: The input grammar for a simple calculator.

I will also investigate whether Basilisk can be combined with Alhazen. Consider the running example. Alhazen is supposed to find out that `"sqrt(x)"` fails for all $x <= 0$. Alhazen does so by repeated trials: The initial inputs are `"sqrt(-9)"` and `"sqrt(900)"`, which leads to the hypothesis that `"sqrt(x)"` fails for all $x <= 445$. Alhazen cut the search space in half, similar to how a binary search would find the correct value[1]. This process is repeated, until the partition of the input space that the decision tree describes cleanly separates behavior-triggering and non-behavior-triggering runs. This process is rather costly: Alhazen needs several tries until the decision boundary is finally found. If Basilisk can indeed focus on the program part which processes `"-9"`, it should be able to find this boundary value much faster.

The implementation presented in this section is just a prototype. There clearly are some design decisions that I would take differently if I were to build this tool again. Those will be discussed in Section 5.7.1. However, it still yields results that show that this approach should be considered.

5.1 Motivating example

In this section, I will describe the basic operation of Basilisk with an example. I will do so in the context of Alhazen, such that the chapter already motivates the integration of both tools, however, keep in mind that Basilisk may be useful on its own already. I will again use the running example I used earlier: The calculator.

The calculator accepts inputs as described by the grammar in Figure 5.1. Those are function invocations like `"sqrt(9)"` or `"tan(22)"`. It crashes if in `"sqrt(x)"`, x is smaller or equals 0.

Alhazen initially receives two inputs `"sqrt(900)"`, and `"sqrt(-9)"`. The second one fails, the first one does not. As in the previous chapters Alhazen starts by forming an initial hypothesis. It claims that all inputs with a number below `"-445"` fail. While this is not the correct boundary value, this hypothesis is still correct, at least in a sense: It explains all observations made so far, it is just not precise.

Alhazen further extracts a predicate set, just as before. In this case the predicate sets are

- $\{\text{max-numeric}(\langle \textit{number} \rangle) < 445\}$; and
- $\{\text{max-numeric}(\langle \textit{number} \rangle) \geq 445\}$.

[1] In fact, the test point – `"445"` – is selected by the decision tree learner. It happens to be half way between `"-9"` and `"900"`, as it would be in a binary search, but the algorithm is not binary search.

For each predicate set, Basilisk now analyses the input and sees that the hypothesis considers a part of it only: The ⟨*number*⟩ is addressed, but not the ⟨*function*⟩. Therefore, Basilisk assumes that ⟨*number*⟩ is the relevant part of the input, and decides to further modify this part of the input. For each predicate set, it identifies a sample input which fulfills this predicate set. In the example, there are only two sample inputs, and therefore `"sqrt(-9)"` is used for the first predicate set, and`"sqrt(900)"` for the second.

Basilisk now *carves* unit tests from the execution of those samples. Carving was pioneered by Elbaum et al. [16]. It works in the following way: Basilisk executes the samples obtained from Alhazen. During the execution, it observes function calls within the program under test. The calculator program starts by reading the input sample from a file. Then, it passes the expression, as a string, to a function named `eval`. `eval` identifies the function and calls `parse_number` to extract the argument and convert it to a numeric value. Then, `eval` invokes `msqrt`. This function finally computes the square root, and contains the bug. Basilisk generates a unit test for each of those functions. Each unit test consists of two parts. First, it (re-)generate the memory contents at the time of function invocation, and then it invokes the function with the same arguments as observed during the execution on the input sample. For the example, Basilisk generates:

1. a unit test for `eval` with the argument `"sqrt(-445)"`,
2. a unit test for `parse_number` with the argument `"-445)"`,
3. a unit test for `msqrt` with the integer argument -445,

Afterwards, Basilisk uses a symbolic execution engine, Klee[8], to *analyze* each of those unit tests. Symbolic execution identifes an input for each execution path within the function under test.

1. For `eval`, Klee gives `"sqrt(-40)"`, `"sqrt(-445)"` and `"sqrt(-448)"`.
2. For `parse_number`, Klee gave `"-40)"`, `"-445)"` and `"-448)"`.
3. For `msqrt`, I obtain 2147483647, and 0.

Basilisk uses those values to generate new system tests, a process I call *lifting*. Within the example, it generates the input samples `"sqrt(-40)"`, `"sqrt(-445)"`, `"sqrt(-448)"`, `"sqrt(2147483647)"`, and `"sqrt(0)"`.

Those new sample inputs are quite informative for Alhazen. `"sqrt(0)"` shows exactly the decision boundary, and `"sqrt(2147483647)"` indicates an upper bound for the value of the argument. Using those new samples, Alhazen can immediately generate a better hypothesis.

5.2 Background

Besides Carving, there are several enabling technologies for Basilisk. The first is unit-level analysis: Basilisk generates unit tests, which would be useless without techniques to analyse them, and suggest new parameter values. The first two subsections of this section present fuzzing and symbolic execution, the unit-level analysis techniques I am going to use in this thesis. The last section presents LLVM, the framework I used in the implementation of carving for Basilisk.

Listing 5.1: Buggy code to find the square root of an integer.

```
1   // This function checks whether expr starts with a given string,
2   // and, if so, increases the expr pointer.
3   bool match(char **expr, const char *expected) {
4         size_t i = 0;
5         for(;expected[i] != '\0';i++) {
6               if((*expr)[i] != expected[i]) return false;
7         }
8         *expr += i;
9         return true;
10  }
```

*expr	symbolic
expected	"sqrt"

(a) Initial program state

*expr	symbolic
expected	"sqrt"
i	0

(b) Program state after executing `size_t i = 0`.

*expr[0]	!= "s"
expected	"sqrt"
i	0

(c) First program state after evaluating `*expr[0] != expected[0]`.

*expr[0]	== "s"
expected	"sqrt"
i	0

(d) Second program state after evaluating `*expr[0] != expected[0]`.

Table 5.1: Execution states in a symbolic execution of Listing 5.1.

5.2.1 Symbolic Execution

A key element of Basilisk is symbolic execution. Symbolic execution is used to generate new test inputs, exercising previously undiscovered execution path within the program. Symbolic execution divides program values in two groups:

Concrete values are handled as in any other type of execution.

Symbolic values are treated as modifiable; Instead of executing, the execution engine collects *constraints* on how they are used. Later on, a constraint solver can be used to get concrete values for symbolic values, which exercise a specific program path.

Let's illustrate this with an example. Consider the function `match`, displayed in Listing 5.1. This function checks whether a string starts with a given pattern. For the example, assume that `expr` is a pointer to symbolic memory, while `expected` in the concrete value `"sqrt"`. This program state is shown in Table 5.1a. The program now executes the first statement of the function, `size_t i = 0`. This introduces a new variable, named `i` and with the value 0, as can be seen in the next program state in Table 5.1b. The execution now proceeds to evaluate the condition for the `for` loop. It finds that `expected[0]` is `"s"`, and therefore `expected[0] != '\0'`. The program enters the loop body, and evaluates the expression `(*expr)[i] != expected[i]`. Here, `*expr[0]` is a symbolic value.

Therefore, the execution engine generates two states. The first state, shown in Table 5.1c, records the formula `expr[0] != 's'`, and, as there is a `return` statement within the body of the `if` statement, terminates the function. The second state, shown in Table 5.1d, records the formula `expr[0] == 's'` and goes on to execute `i++`. If symbolic execution goes on for a couple more steps, the resulting formula is `*expr[0] == "s"` $\wedge$ `*expr[1] == "q"` $\wedge$ `*expr[2] == "r"` $\wedge$ `*expr[3] == "t"`. As a result, a constraint solver would generate the value `"sqrt"`, and therefore an input which executes the `for` loop to its entirety. However, at the same time, symbolic execution also generates the formulas `*expr[0] != "s"` and `*expr[0] == "s"` $\wedge$ `*expr[1] != "q"` and `*expr[0] == "s"` $\wedge$ `*expr[1] == "q"` $\wedge$ `*expr[2] != "r"` and `*expr[0] == "s"` $\wedge$ `*expr[0] == "q"` $\wedge$ `*expr[0] == "r"` $\wedge$ `*expr[0] != "t"`. If for each of those formulas, one input is generated, and `match` is executed with all those inputs, all possible execution path are exercised.

This example hints at one weakness of symbolic execution. Symbolic execution reasons about all execution path of a program. However, especially in the context of loops, programs can have many execution path. Just think about an `if` statement in the body of a loop. In the example, I showed how an `if` statement generates two new program states that symbolic execution needs to consider. Within the example, one branch of the `if` leads to a `return` statement. If both branches allow another iteration of the loop to follow, it means that there are 2^i states for a loop with i iterations. This leads to exponential growth in the number of program states to consider, and therefore in the runtime of a symbolic execution engine. This problem is knwon as *path explosion*.

Klee

In this thesis, I am using Klee[8]. Klee is a symbolic execution engine for the C language. At its time, Klee provided a number of optimizations over state-of-the-art symbolic execution engines.

Compact State Reresentation As explained when I introduced the path explosion problem, symbolic execution engines need to keep track of thousands, if not hundreds of thousands, of program states. Klee meets this challenge with an in-memory representation which favors re-use. A memory-object-level copy-on-write mechanism allows for a compact representation of memory states.

Query Optimisation The authors of Klee observed that the time spend on constraint solving dominates the runtime of the symbolic execution itself. Therefore they implemented a number of query simplifications, as well as caching techniques to find solutions faster.

State Scheduling A key question within symbolic execution is which state to execute next. With an `if` statement, should the engine investigate the `true` branch or the `false` branch? What if there is still a branch from a previous `if` statement to be considered? The Klee authors treat the problem like a tree search, and implemented two new heuristics to tackle the problem.

Those optimizations allow Klee to scale to programs that were previously impossible to execute symbolically.

5.2.2 Fuzzing

Fuzzing was introduced by Miller, Fredriksen, and So [49]. The main idea is very simple: Provide the program under test with random data, and see whether it crashes. At first glance, this is silly. Noone provides programs with random data, so all crashes found during fuzzing would be crashes that never happen in production. However, programs need to stand up to user errors, and, if they expose an interface to untrusted parties, for example by exposing it to the internet, they may be targets for attacks. Programs need to be hardened against malformed inputs, no matter whether those were generated by user error or ill intent. Every crash may be exploitable, or, in multi-user systems, cause a service disruption for other users. Therefore, software developers usually fix every crash that can be triggered with an input given to a public interface of their program.

Fuzzing has seen quite some development in recent years. Fuzzers can be characterized by several characteristics. The first question is how much information they have about the program under test.

Black-box fuzzers have no information about the program under test at all. This class includes Radamsa, and Miller's original work.

White-box fuzzers perform analysis of the program under test. Examples include tools like RedQueen[5], which uses program analysis to solve some program constraints symbolically.

Gray-box fuzzers are somewhere in between white- and black-box fuzzers. While no formal definition exists, the term is usually used to say that a fuzzer does use information from program analysis, but that the analysis is very basic. An example for this is AFL, which collects coverage information.

The second question is how the fuzzer derives new inputs:

Random input generation derives new inputs at random, without any information. The aforementioned paper by Miller, Fredriksen, and So [49] is an example for this appraoch.

Mutational input generation derives new inputs by mutations of existing inputs. Tools like Radamsa[32] and AFL[75] are examples for this.

Search-based input generation uses a search algorithm, for example an evolutionary algorithm([75]) or gradient descent([43]), to decide which mutations to apply to existing inputs.

In this thesis, I am going to use a fuzzer called Radamsa[32]. This fuzzer is a mutational, black-box fuzzer. Basilisk itself can be interpreted as a white-box fuzzer, using unit-level analysis as a means to generate new inputs.

Unit-level Fuzzing

Fuzzing also appears in my thesis as a unit-level analysis technique. While fuzzing at the unit-level has been done before, for example by Pacheco and Ernst [52], it suffers from a conceptual problem. Unit-level interfaces are typically not

externally accessible. Consequently, malformed inputs are not a concern: Other layers of the system can filter them before they cause harm. In contrast to system level interfaces, unit-level interfaces are not a target for attack, or user error. But then, unit-level fuzzing suffers a high false positive rate. Every crashing input may be unrealistic, in the sense that it could never go into the unit within the full system. Basilisk solves this problem, because it can validate unit-level inputs by lifting them to the system level. Only if the generated system test fails, it will be reported to the developer. Therefore, Basilisk, just as all other variants of system-level fuzzing, has a false positive rate of zero.

5.2.3 Low-Level Virtual Machine

Basilisk is built on top of the Low-Level virtual machine (LLVM)[41]. Despite its name, LLVM is not a virtual machine, but a compiler construction framework. It is used in compilers such as the C compiler clang, the compiler for the rust programming language, or Nvidia's Cuda compiler [12]. Compilers built on top of LLVM first translate the program into an intermediate representation, called LLVM IR. Then, optimizations can be applied to this representation. That means that LLVM optimizations can be reused for different programming languages.

LLVM IR is a representation of the program in static-single assignment form (SSA) [13]. This means that each variable, called a register, can be assigned only onces. Therefore it is easy to reason about which values are in which register when, a property that makes it especially easy to implement optimisations and, as I will do in this thesis, instrumentations.

5.3 Carving C Programs

Elbaum et al. [16] presented carving as a way to generate unit tests out of system tests. Carving observes function invocations within the system test, and uses those invocations as unit tests. Elbaum et al. [16] present two variants of carving:

State-Based Carving extracts exactly one function invocation, and writes all its inputs into the test as concrete values.

Action-Based Carving chooses one function invocation as well, but uses more of the observed function calls to construct all required input values.

In the following, I will present both approaches in more detail.

5.3.1 Carving Approach

When executing a system test, the program performs a sequence of function invocations. Those invocations form a tree: Each function is a child of the function which invoked it. Also, each function accepts inputs, and returns outputs: Concrete values, which may be results of computations in other functions, and pointers to memory areas. All this forms the execution graph of a program.

Definition 25: Execution Graph
An *execution graph* is a directed, acyclic graph where each function invocation within a program execution and each memory area used by the

program is represented as a node. If, during the execution, function `A` invoked function `B`, there is an edge from the node labeled with `A` to the node labeled with `B`. If a memory area `M` is returned by `A`, there is an edge from `A` to `M`, if it is used as an input to `A`, there is an edge from `M` to `A`. If a memory area `M` contains a pointer to a memory area `M2`, there is an edge from `M` to `M2`.

Listing 5.2 shows an excerpt from the calculator which serves as an example The program accepts a file name as input, and evaluates the expression within the file. The code listing does not include any error handling or validity checks for the input.

Listing 5.2: Code for the calculator example. Some error handling and function bodies were removed for readability.

```
1       // helper function to read all text from a file
2       // left out for readability
3       char *read_all_text(char *path) ...

5       // This function checks whether expr starts with a given string,
6       // and, if so, increases the expr pointer.
7       bool match(char **expr, const char *expected) {
8             size_t i = 0;
9             for(;expected[i] != '\0';i++) {
10                  if((*expr)[i] != expected[i]) return false;
11            }
12            *expr += i;
13            return true;
14      }

16      // this function parses a number from expr,
17      // and increases the expr pointer afterwards
18      int32_t parse_number(char **expr) {
19            int32_t number = 0;
20            int32_t factor = 1;
21            if(*expr[0] == '-') {
22                  (*expr)++;
23                  factor = -1;
24            }
25            char *start = *expr;
26            for(;'0' <= *expr[0] && *expr[0] <= '9';(*expr)++) {
27                  int32_t digit = *expr[0] - '0';
28                  number *= 10; number += digit;
29            }
30            return number*factor;
31      }

33      int32_t msqrt(int32_t x) {
34         x = max(x, 0);
35         int32_t approx = -1;
36         int32_t guess = x / 2;
37         while(approx != guess) {
38            approx = guess;
39            guess = (approx + x / approx) / 2;
```

```
40          }
41          return approx;
42      }

44      int32_t eval(char *input) {
45            int32_t (*f)(int32_t) = NULL;
46            char** expr = &input;
47            // check which function it is
48            if(match(expr, "sqrt")) {
49                  f = msqrt;
50            } else if(match(expr, "sin")) {
51              f = msin;
52            } else if(match(expr, "cos")) {
53                  f = mcos;
54            } else if(match(expr, "tan")) {
55                  f = mtan;
56            }
57            if((*expr)[0] != '(') return -1;
58            (*expr)++;
59            // read the number
60            char *s = *expr;
61            int32_t number = parse_number(expr);
62            if((*expr)[0] != ')') return -1;
63            (*expr)++;
64            // calculate and return the result
65            return f(number);
66      }

68      int main(int argc, char **argv) {
69            char *text = read_all_text(argv[1]);
70            int32_t result = eval(text);
71            free(text);
72            printf("result: %d\n", result);
73            return 0;
74      }
```

When the program is invoked with the argument `in.expr`, and the text `"sqrt(900)"` in the file `in.expr`, the execution graph in Figure 5.2 is obtained. The `main` function, in Line 68 of the code, is represented as node F1 in the graph. It can be seen in the code that it takes two inputs: The number of arguments `argc` and a pointer to the arguments, `argv`. Within the execution graph, `argc` is not visible, because it is a value, not a pointer. `argv` is a pointer to an array of pointers. The array itself is represented by the node M0. It contains pointers to the program name `argv[0]` and the string `"in.expr"`, which is `argv[1]`. In the memory graph, those are represented by edges to nodes M1 and M2.

`main` calls `read_all_text` (Line 69), therefore there is an edge to node F2 in the graph. `read_all_text` receives `"in.expr"` as an argument. In the program, this is a pointer to M2. Therefore, an edge from M2 to F2 exists. `read_all_text` reads the entire text from the file, and returns a pointer to a new memory area, represented by node M3 in Figure 5.2. This memory area is returned by F2, and hence there is an edge from F2 to M3. `eval` uses `match` to check the function name (Line 48, node F3). `match`, however, does not receive a pointer to `"sqrt(900)"`, but a pointer to a pointer. This is done because match also advances the position

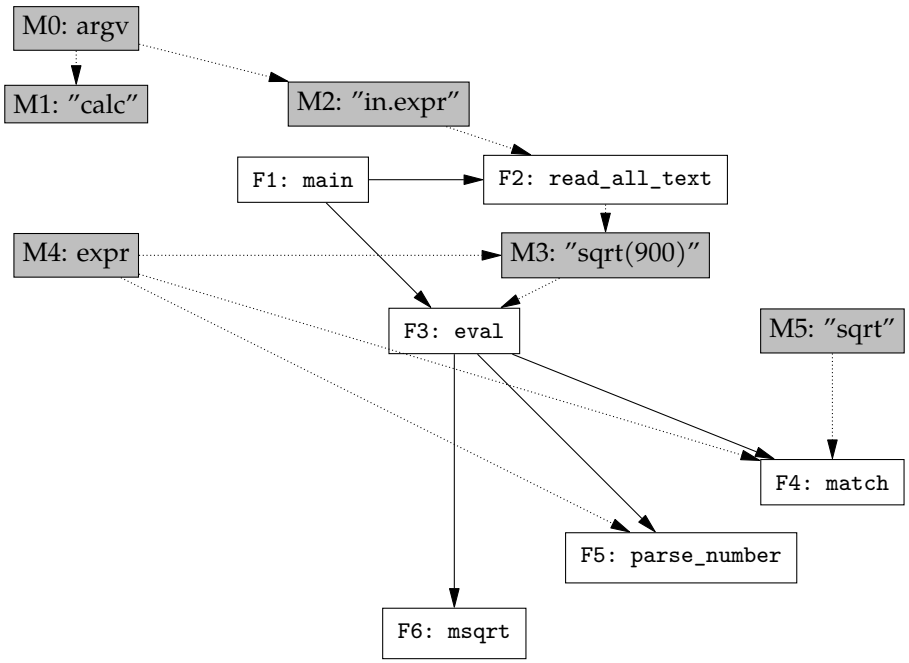


Figure 5.2: Execution graph for the program in Listing 5.2.

of this pointer to the next character that needs to be processed by `eval`. Therefore, memory area M4 contains a pointer to M3, represented by an edge from M4 to M3. Then, there is an edge from M4 to F3, indicating that `match` receives this pointer. As second argument, `match` receives the string to compare with, `"sqrt"`. This is a constant in the source code, and therefore a pointer to the program's data section, represented by an edge from memory area M5. During execution, `match` updates the pointer it received as first argument, this, however, just means the value in the memory area changes. It does not generate new memory area. Therefore, F5 and F6, the subsequent calls to `parse_number` (Line 61) and `msqrt` (Line 65) still receive the same memory area, and their nodes have incoming edges from M4 just as well. Both `parse_number` and `msqrt` finally return an integer, but this is not visible within the execution graph, as it is call by value, not call by reference.

Carving a unit test means to slice the execution graph: A function invocation can be replayed by executing the same function, with the same inputs. Each input is represented as an edge, and there are two ways to handle edges in carving:

Recording the input value can be recorded in the original execution, and used in the test execution. Within Basilisk this is done for all values.

Recreation If the edge represents that a memory area returned by a function call `A` is used within `B`, the edge can be traversed backwards: `A` can be called in the test, and the memory area can be passed through. All input values for the call to `A` need to be created within the test, which means that incoming edges for `A` need to be recorded or recreated as well.

Listing 5.3: A test obtained from the execution graph in Figure 5.2 via state-based carving.

```
1 void state_based_test() {
2       char *n1 = (char *) malloc(5);
3       strncpy(n1, "900)", 5);
4       parse_number(&n1);
5 }
```

Listing 5.4: A test obtained from the execution graph in Figure 5.2 via action-based carving.

```
1 void action_based_test() {
2       char *n1 = read_all_text("in.expr");
3       char *n2 = &n1;
4       match(n2, "sqrt");
5       parse_number(n2);
6 }
```

Elbaum et al. [16] present an approach which is based on recording only, they call it *state-based carving*. They also present *action-based carving*, which is based on recreating only.

Within the example, a state-based test for `parse_number` can be found in Listing 5.3. The function under test, `parse_number` is invoked in Line 4. Before that, all input values are constructed by allocating memory, and directly setting the values recorded in the execution graph. This means that the generated test calls just one function: the function under test. Input values are constructed based on the observations.

In contrast, action-based carving generates the test in listing 5.4. In this test, `parse_number` is invoked in Line 4. The `parse_number` node in the execution graph has one input: the memory area M4. M4 contains a pointer to M3, and it is also used in F4, `match`. M3 is returned by F2 `read_all_text`. F2 receives M2 as an input, and M2 is a system-level input. Therefore, the test contains the call to `read_all_text`, as well as the call to `match`. `n2` is set up to be a pointer to `n1`, corresponding to M4 and M3 respectively. The second argument to `match` is a constant within the program, and therefore it is used as is.

5.3.2 Implementation of Carving

In this section, I describe how to carve C programs. In order to do so, I need to construct the execution graph of a program execution. Building an execution graph requires three different kinds of information:

The Call Graph is a sub-graph of the execution graph.

The Program Memory contains all memory areas used by the program. Information about the memory is required to build the memory nodes within the execution graph.

Method Arguments are not strictly a part of the execution graph, however, they are required in order to re-generate a function invocation, and therefore need to be recorded.

The main mechanism that I use to collect this information are *probes*. Probes are additional instructions, inserted into the program under test. Every time a probe is executed, it reports data to a supervisor process. This way, the program execution generates a *trace*, a list of the probes triggered, and information reported.

My implementation of carving is based on LLVM[41]. It works in two phases. At *instrumentation-time*, I insert probes into the program, and create a file which I call the structure file. The structure file contains information on types used in the program, the functions that exist, and their signature, as well as global variables. This file is read by the supervisor process, which uses the information on types to calculate their in-memory sizes, the information on functions and their signatures is used to interpret the data on method invocations correctly, and the information on global variables is used to identify those variables correctly. Basilisk assigns each function and each global variable a unique id. Those ids are used to identify such objects within the traces.

The instrumented program is compiled to machine code with the standard LLVM compiler. At *runtime*, Basilisk sends information about which probes where triggered to the supervisor process.

The trace for an execution of the program in Listing 5.2 can be found in Table 5.2. Within the trace, I replaced the ids of functions and global variables with the function and variable names for better readability. Instead of going through the trace top to bottom, I will describe the individual probes (or events) ordered by their functionality.

Obtaining the Call Graph

The first challenge is to report the call graph. This is done with two events: `Called` and `Return`. Within Table 5.2, event 31 is a `Called` event. It reports that the `main` method has been invoked. After that, there is another `Called` event at position 37. This one reports that `read_all_text` was invoked. The trace did not yet contain a `Return` for `main`, and therefore I know that `read_all_text` was invoked by `main`. This means there is an edge from the call graph node for `main` to `read_all_text`. This corresponds to Figure 5.2, where the nodes F1 and F2 are connected. The `return` event for `read_all_text` is event 50. Then, in event 51, there is a `Called` event for `eval`. This means that `read_all_text` returned before `eval` was called, but `main` did not. Therefore, the node for `eval` (F3 in Figure 5.2) is connected to `main`, but not to `read_all_text`. In general, nodes are connected if the corresponding methods are active at the same time. The program behaves as if the call graph were visited in a depth-first traversal, outputing `Called` events when a node is entered, and `Return` events when all children were visited. It is easily possible to re-create the call graph from this representation. The only difficulty lies in multi-threading. If more than one thread sends events, each thread behaves in the way described before, and care has to be taken to make sure that events are not mixed up. For this reason, each event contains the thread id of the generating thread.

Parameter Values

For carving, Basilisk also needs concrete values for all parameters. Those are reported with `P*` events. Within Table 5.2, I can observe `PInt` events, for integer parameters, and `PPtr` events for pointer parameters.

The first `PInt` event within the example is event 32. The data associated with this event indicates the function and thread that the value belongs to, in order to enable associations of the events with the call graph, as well as the concrete value (2) and the parameter id. Parameters are numbered starting at 0, the parameter id of 0 therefore indicates that this is the `argc` parameter of the `main` function. Event 36, a `PPtr` event, reports that the second argument to `main` is the pointer 140730708315240. Other events reported about the memory contents at this location. They will be explained in Section 5.3.2.

The implementation also supports `PDouble` events, for floating-point numbers, and `PArea` events, for cases where an entire memory region is passed by value, rather than by pointer. Those do not appear within the example, as they are not required for the example program.

Return Values

Return values are reported with the `P*` events, just as parameters. However, they have negative parameter ids. A simple case of this can be seen in event 48 within Table 5.2. This event reports the return value for the `read_all_text` function, a pointer to memory addres 140730708322148. The id is -1, indicating a direct return value.

If a function argument is a pointer to a pointer, the function may modify this pointer, and use this as another return value. Within the example, the function `match` updated the pointer within its first argument to indicate where the matched string ends. This is reflected in the trace: Event 63 is a `PPtr` event with a parameter id of -2, indicating a value returned via the first parameter. As for parameters, the concrete contents of memory areas pointed to are reported with other methods.

Live Coverage Information

Basilisk supports two variants of obtaining coverage data. Traces can contain an event for each basic block of the program, or LLVM's LCOV can be used, and updated with live reporting of coverage data. This feature is not strictly required for Basilisk itself, but the tool was used in other projects which required this

LLVM IR splits functions into basic blocks. Each basic block contains a sequence of instructions with no branch or return instructions, such that the control-flow graph of a function is a graph of basic blocks. When Basilisk reports a function entry (as in event 66), it also reports the first basic block which is being reported. After that, it supports a `NEXT_BB` event, which reports every time a new basic block is entered. Due to performace considerations, this was turned off for Basilisk runs within this thesis [2]. It can be used to calculate instruction, block, branch coverage and linear code sequence and jump coverage (LCSAJ). To my knowledge, mine is the only tool which can report LCSAJ coverage of a C program. The hefty performance overhead may be the reason for this.

[2] For this reason, there are no such events in Table 5.2

As a more performant option, Basilisk can also report live data from LCOV. LCOV creates a bit set where each bit corresponds to one edge of the control-flow graph of the program. It sets those bits to 1 when the corresponding edge is executed, effectively reporting branch coverage. Instruction coverage or basic block coverage can be reported from this data. Basilisk reports the total length of this bit set with an `AllCounters` event (event 41 within the example), and then uses a `BitCounter` event to report the counters for the edges within a specific method, every time a method returns. This allows the calculation of coverage while the program is running, rather than reporting coverage only after the program terminated.

Reporting Memory Objects

Now comes the hard part: I need to report each memory object accessed by the program under test. Memory objects can be in three different memory areas:

- The *data section* of the program binary contains memory that is allocated to a global variable.
- The *call stack* contains variables allocated as local variables within a function.
- The *program heap* contains memory objects that were allocated with the `malloc` and `free` functions of the C standard library.

All three can be found in the example:

Data Section Source code constants are compiled to global variables within LLVM. Therefore, the second argument to `match`, `"sqrt"`, is a pointer to a value in the data section.

Call Stack The first argument to `match`, on the other hand, is a pointer to a pointer. The pointer can be found within the call frame of the `eval` function, and therefore the pointer points into the call frame.

Program Heap The output of `read_all_text` is a pointer to a memory area on the heap. It was allocated with the `malloc` library function, and it is `free`'d with the `free` function within `eval`.

In general, Basilisk returns memory objects with the `Area` event. This can be seen, for example, in event 47. This event reports that the value `"sqrt(9)\n"` can be found at memory address 29225952. The same memory address is used when event 48 reports the return value of `read_all_text`. Another example is event 58, which reports that the value of the global variable `.str.1` is `"sqrt\0"`. Within Table 5.2, I use the name of a global variable instead of a numeric memory address whenever possible.

However, I can only create such events when the length of a memory object is known. Otherwise, I would not be able to access the value, as accesses outside the bounds of a memory object lead to access violation errors.

This is easiest[3] for memory objects on the heap. Those are created via the `malloc` library function, and later on, destroyed via the `free` library function.

[3] easiest, not easy

Within the process of the program under test, Basilisk's instrumentation keeps a map of addresses and the length of the memory objects at those addresses. Memory objects are entered into the map on calls of `malloc`, and removed when `free` is called.

When a pointer is encountered, this map can be queried to get the length of the memory event pointed to. Then, it is known how much memory can savely be accessed, and reported in the `Area` event. Pointers can also point into such a memory object. As an example, the pointer returned by `match`, and reported in `Area` event 62, points four bytes after the start of the memory area which was reported by event 52. This means that the map built by Basilisk cannot be an associative map. It needs to be able to answer queries for the largest key smaller than a search value. Within my implementation, I used an AVL tree[2].

The supervisor process also needs to know whether an `Area` event refers to a new memory area, or contains updated values for an existing one. While invocations of `malloc` do not send an event, it can be assumed that there was a `malloc` when the first `Area` event for a memory area arrives, invocations of `free` need to be reported. Otherwise, a newly (m)allocated memory object may be confused with one that is `free`'d already. For that reason, the instrumentation also delivers events for `free`. One such `Free` event can be seen as event 94 in Table 5.2. It corresponds to the invocation of `free` within `eval` in Listing 5.2.

Memory objects on the call stack are only slightly more complicated. Within LLVM, such memory objects are allocated via the `alloca` instruction of LLVM IR. When `alloca` is invoked, I add the memory area to the map, just as I did for `malloc` calls. After that, there is no need for special treatment when an `Area` event is triggered. However, there are no `free` invocations for such memory areas. Instead, they are implicitly `free`'d when the function they were allocated in returns. For that reason, I clear them from the map when the function returns, and the `Return` event contains the bottom and top addresses of the function's call frame. The supervisor process uses this information to avoid confusion of old, and newly allocated stack objects, just as it uses the `free` event for heap memory.

Last are global variables. Those can, in theory, be found at instrumentation time. They are contained in the program source code. However, their memory addresses are only known at runtime. Basilisk's instrumentation informs the supervisor about them with the `Global` event, several of which can be found at the beginning of Table 5.2. Their length can be found at instrumentation time, and therefore they can be added to the map right when the program starts. There is no equivalento to the `Free` event, as global variables exist until the program terminates.

Basilisk also issues `global` events for all functions in the program. Those also contain the memory address of a function, and can be used to recognize function pointers when those appear within function arguments.

Recursion In some cases, it is not enough to report the content of a memory area. In the example, this can be seen for the second argument of the `main` function. `argv` is an array of pointers. Basilisk needs to create the structure that is formed by M0, M1 and M1 in Figure 5.2. This is done with three `Area` events. Event 33 corresponds to M0. It is at memory address 140730708315240, and contains the values 140730708322111 and 140730708322148. Both are memory

addresses themselves. During instrumentation, Basilisk saw that the type of the variable is `i8**`, a pointer to a pointer. Therefore, it inserts code which recursively dumps the content of the memory area. At runtime, it queries the map for the length of the memory area at address 140730708315240. It receives the answer that the length is 16. The type, known at instrumentation time, is `i8**`. Therefore Basilisk knows that each member of this memory area is 8 bytes long, and can calculate that there are 2 members within an area of length 16. At instrumentation time, Basilisk determined that the type of those members is `i8*`, and therefore a pointer type. It generates the `Area` events for the memory areas that both pointers point to. The members of those areas have the type `i8`, so no further recursive events is required. If the type were `i8***`, or a pointer to a struct which has a pointer member, Basilisk would recursively dump all reachable memory. In the example, events 34 and 35 contain the strings that are arguments to the `main` function.

A limitation of this process is that Basilisk needs to be able to determine that memory contains a pointer based on the LLVM type of memory object at instrumentation time. If this is not possible, Basilisk will not invoke recursive dumping.

Table 5.2: The event sequence generated when execution Listing 5.2.

1	Global	name: address:	.str 5071819
2	Area	address: value:	.str `"r\0"`
3	Global	name: address:	.str.1 5126502
4	Area	address: value:	.str.1 `"sqrt\0"`
5	Global	name: address:	.str.2 5126507
6	Area	address: value:	.str.2 `"sin\0"`
7	Global	name: address:	.str.3 5073229
8	Area	address: value:	.str.3 `"cos\0"`
9	Global	name: address:	.str.4 5126511
10	Area	address: value:	.str.4 `"tan\0"`
11	Global	name: address:	.str.5 5126515

12	Area	address: value:	.str.5 `"No number!\n\0"`
13	Global	name: address:	.str.6 5126527
14	Area	address: value:	.str.6 `"Could not identify function!\n\0"`
15	Global	name: address:	.str.7 5126557
16	Area	address: value:	.str.7 `"Wrong number of arguments\n\0"`
17	Global	name: address:	.str.8 5126584
18	Area	address: value:	.str.8 `"result: %d\n\0"`
19	Global	name: address:	fsize 4997264
20	Global	name: address:	read_all_text 4997616
21	Global	name: address:	match 4998144
22	Global	name: address:	parse_number 4998784
23	Global	name: address:	abs 4999600
24	Global	name: address:	msqrt 4999952
25	Global	name: address:	msin 5000512
26	Global	name: address:	mtan 5000768
27	Global	name: address:	mcos 5001024
28	Global	name: address:	eval 5001280
29	Global	name: address:	main 5002512
30	All Counters	length:	41

31	Called	threadId: 140040092858304 function: main basicBlock: 0
32	PInt	threadId: 140040092858304 function: main parameterId: 0 value: 2
33	Area	address: 140730708315240 value: { 140730708322111, 140730708322148}
34	Area	address: 140730708322111 value: `"calc_instr\0"`
35	Area	address: 140730708322148 value: `"in.expr\0"`
36	PPtr	threadId: 140040092858304 function: main parameterId: 1 pointer: 140730708315240
37	Called	threadId: 140040092858304 function: read_all_text basicBlock: 0
38	Area	address: 140730708322148 value: `"in.expr\0"`
39	PPtr	threadId: 140040092858304 function: read_all_text parameterId: 0 pointer: 140730708322148
	...	
47	Area	address: 29225952 value: `"sqrt(9)\n"`
48	PPtr	threadId: 140040092858304 function: read_all_text parameterId: -1 pointer: 29225952
49	BitCounters	threadId: 140040092858304 function: read_all_text
50	Return	threadId: 140040092858304 function: read_all_text frameBottom: 140730708314816 frameTop: 140730708314656

51	Called	threadId: 140040092858304 function: eval basicBlock: 0
52	Area	address: 29225952 value: `"sqrt(9)\n"`
53	PPtr	threadId: 140040092858304 function: eval parameterId: 0 pointer: 29225952
54	Called	threadId: 140040092858304 function: match basicBlock: 0
55	Area	address: 140730708314688 value: { 29225952 }
56	Area	address: 29225952 value: `"sqrt(9)\n"`
57	PPtr	threadId: 140040092858304 function: match parameterId: 0 pointer: 140730708314688
58	Area	address: .str.1 value: `"sqrt\0"`
59	PPtr	threadId: 140040092858304 function: match parameterId: 1 pointer: 5126502
60	PInt	threadId: 140040092858304 function: match parameterId: -1 value: -1
61	Area	address: 140730708314688 value: { 29225956 }
62	Area	address: 29225956 value: `"(9)\n"`
63	PPtr	threadId: 140040092858304 function: match parameterId: -2 pointer: 140730708314688
64	BitCounters	threadId: 140040092858304 function: match

65	Return	threadId: 140040092858304 function: match frameBottom: 140730708314608 frameTop: 140730708314448
66	Called	threadId: 140040092858304 function: parse_number basicBlock: 0
67	Area	address: 140730708314688 value: `"\FFFD\FFFD\0001\0\0\0\0"`
68	Area	address: 29225957 value: `"9)\n"`
69	PPtr	threadId: 140040092858304 function: parse_number parameterId: 0 pointer: 140730708314688
70	PInt	threadId: 140040092858304 function: parse_number parameterId: -1 value: 9
71	Area	address: 140730708314688 value: { 29225958 }
72	Area	address: 29225958 value: `")\n"`
73	PPtr	threadId: 140040092858304 function: parse_number parameterId: -2 pointer: 140730708314688
74	BitCounters	threadId: 140040092858304 function: parse_number
75	Return	threadId: 140040092858304 function: parse_number frameBottom: 140730708314608 frameTop: 140730708314416
76	Called	threadId: 140040092858304 function: msqrt basicBlock: 0
77	PInt	threadId: 140040092858304 function: msqrt parameterId: 0 value: 9

78	Called	threadId: 140040092858304 function: abs basicBlock: 0
79	PInt	threadId: 140040092858304 function: abs parameterId: 0 value: -5
80	PInt	threadId: 140040092858304 function: abs parameterId: -1 value: 5
81	BitCounters	threadId: 140040092858304 function: abs
82	Return	threadId: 140040092858304 function: abs frameBottom: 140730708314448 frameTop: 140730708314352
83	Called	threadId: 140040092858304 function: abs basicBlock: 0
84	PInt	threadId: 140040092858304 function: abs parameterId: 0 value: 1
85	PInt	threadId: 140040092858304 function: abs parameterId: -1 value: 1
86	BitCounters	threadId: 140040092858304 function: abs
87	Return	threadId: 140040092858304 function: abs frameBottom: 140730708314448 frameTop: 140730708314352
88	PInt	threadId: 140040092858304 function: msqrt parameterId: -1 value: 4
89	BitCounters	threadId: 140040092858304 function: msqrt

90	Return	threadId: 140040092858304 function: msqrt frameBottom: 140730708314608 frameTop: 140730708314448
91	PInt	threadId: 140040092858304 function: eval parameterId: -1 value: 4
92	BitCounters	threadId: 140040092858304 function: eval
93	Return	threadId: 140040092858304 function: eval frameBottom: 140730708314816 frameTop: 140730708314608
94	free	address: 29225952
95	PInt	threadId: 140040092858304 function: main parameterId: -1 value: 0
96	Area	address: 140730708315240 value: { 140730708322111, 140730708322148}
97	Area	address: 140730708322111 value: `"calc_instr\0"`
98	Area	address: 140730708322148 value: `"in.expr\0"`
99	PPtr	threadId: 140040092858304 function: main parameterId: -3 pointer: 140730708315240
100	BitCounters	threadId: 140040092858304 function: main
102	Return	threadId: 140040092858304 function: main frameBottom: 140730708315008 frameTop: 140730708314816

5.3.3 Parameterizing Unit Tests

If unit-level analysis is supposed to find better input values for test cases, the tests need to be *parameterized*, where unit-level analysers are allow to choose new values for the parameters. In this section, I will explain how to introduce parameters into a carved unit test.

Carved tests are, basically, subgraphs of the execution graph. The test in

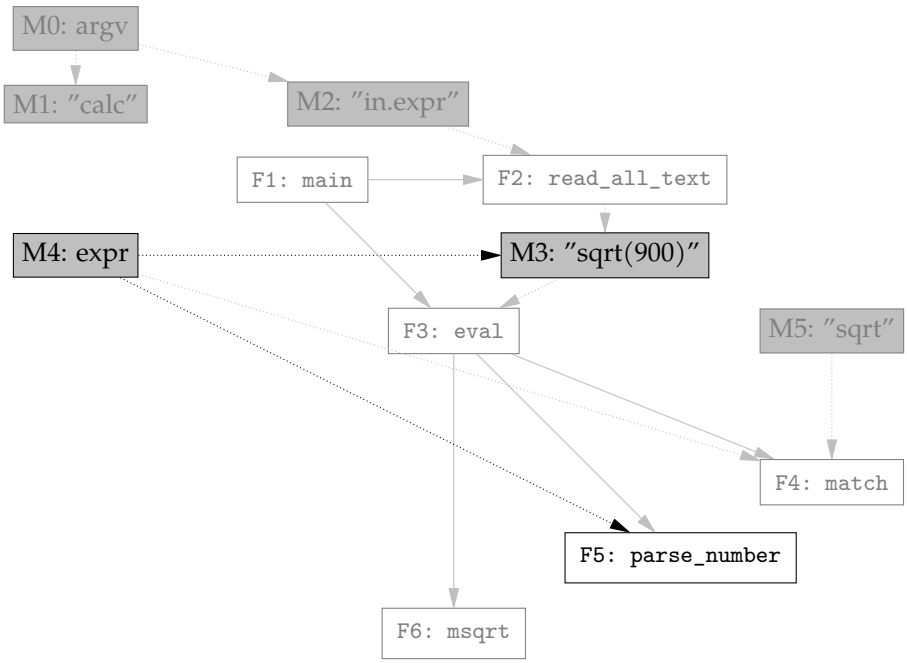


Figure 5.3: Execution graph for the program in Listing 5.2, with the test case in Listing 5.3 highlighted.

Listing 5.3 is a subgraph of the execution graph in Figure 5.2. Figure 5.3 highlights the nodes which form the test. For parameterization, I start by introducing additional nodes, one for each input. This is superfluous for command line parameters: Those are present as arguments to the main function anyways. For other inputs, for example the contents of a file read by the program under test, there may not be such a node. I introduce new nodes for all cases, because the new nodes are different from existing nodes: They are parameters. All such nodes represent byte sequences. Even if the input is, e.g. an integer, it will be represented as a byte sequence at this point. I routinely allow the byte sequences to be three bytes longer than what I observed in the system test. The reason is that especially byte sequences which are interpreted as strings usually allow for additional characters. I choose the value 3, because it allows Basilisk to discover whether longer sequences are permitted, but does not lead to a combinatorical explosion within the search space for symbolic execution.

Then, I check for similarity between the newly introduced nodes and existing memory segments and values. At this point, I check for byte similarity with memory areas and values, but also for encodings. As an example, `"900"` is given to the calculator as part of the string `"sqrt(900)"`, but the integer parameter 900 would still be recognized as similar. In the same way, the byte sequence `0x84030000` would be recognized as the 32-bit integer representation of 900. Basilisk now replaces the recognized values and segments of memory areas with data it copied from the newly introduced nodes. If the similarity was based on an encoding of the data, methods which convert the encodings are added in between.

Consider a test for `msqrt` as an example. The graph for this example is trivial:

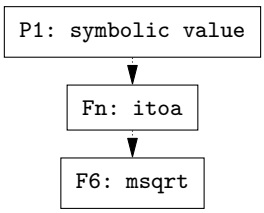


Figure 5.4: The parameterized test case for `msqrt`.

`msqrt` has just one parameter, the integer 900, which is not represented in an execution graph, because it is a value, and not a pointer to a memory area. Figure 5.4 shows the graph for the resulting test case. The node labeled with `P1` is the parameter for the input. It has no concrete value, but represents a parameter which can be filled by unit-level analysis later. A subsequence of this, starting at the fifth character, is fed into `itoa`. This function converts a string into an integer. `itoa` is not part of the original program, it was introduced by parameterization, to convert the byte sequence into an integer, which can then be fed into the function under test, `msqrt`. The test chooses a substring starting at the fifth character of the symbolic input, because within the original input, `"sqrt(900)"`, the string which corresponds to the parameter to `msqrt`, `"900"`, starts at the fifth character.

5.3.4 Lifting Unit-Level Values to the System-Level

Unit-level analysis finds new values for the parameters in a test. Those can then be turned into a new system test: Each system-level input corresponds to one of the newly introduced memory areas within the unit test. The new value that unit-level analysis proposes for this input can be used to replace the corresponding input in the system test. In some cases, ony parts of the parameter were used in the unit test. For example, within the example test for `msqrt`, only `"900"` is used, whereas the whole input is `"sqrt(900)"`. In this case, only the segment which was used is replaced with a new value.

5.4 Evaluation

In my evaluation, I attempt to answer the following research questions:

1. How much overhead does Basilisk's instrumentation generate?
2. How do the *system tests* generated by Basilisk and Fuzzilisk compare against system tests from Radamsa (Section 5.4.3)?
3. Does unit-level fuzzing save time, in comparison to system-level fuzzing (Section 5.4.5)?

5.4.1 Performance of Basilisk

Basilisk performs compile-time instrumentation, and runtime monitoring. While compile-time instrumentation itself does not lead to increased runtimes, it may

mean that some compiler optimizations are not possible any longer. Runtime monitoring happens at runtime[4], and therefore has some runtime overhead. Both effects mean that the program under test is a lot slower when Basilisk is used. In this section, I will quantify this effect.

In order to do so, I ran 5 of the subjects I used with Alhazen earlier. I used the two test inputs from the twoInputs configuration, ran each 100 times, and measured the runtime.

I used three different configurations:

Uninstrumented The subject is run with no instrumentation, and no monitoring.

Coverage The subject only emits the `Called` and `BitCounter` and `AllCounters` events.

Carving The subject emits all events used for carving.

Afterwards, I calculated the mean of the runtime for the Uninstrumented configuration, and a slowdown for the other configurations

> **Definition 26: Slowdown**
> Let $\bar{r_u}$ be the arithmetic mean of the runtime for the Uninstrumented configuration, and r_t be the runtime for one run with configuration t, the *Slowdown* s_t for this run is defined as $\frac{r_t}{\bar{r_u}}$.

Using this definition, the mean of the slowdown for the Uninstrumented configuration is always 1 [5]. Therefore, results for other configurations can be interpreted as "Executing configuration c takes $\bar{s_c}$ times as long as executing the uninstrumented subject."

The results for all subjects can be found in Figure 5.5, as well as Table 5.3 to Table 5.7. The 'test' column lists the id of the test being executed. Those are 0 and 1 for most subjects, but, for technical reasons, different for grep.

Slowdowns range from barely noticeable, for the calculator, to more than 20x, for `jq`. The standard deviation for `jq` is smaller than for the other subjects. The reason is that I ran all subjects with a timeout of 1 second per test run, which was triggered by `jq` on almost all runs. Therefore, the 20x slowdown on `jq` is at best a lower bound of the real slowdown.

[4] hence the name

[5]

$$\bar{r}_u = \frac{1}{n}\sum_{i=0}^{n} r_u^i \text{ and } \bar{s}_u = \frac{1}{n}\sum_{j=0}^{n} \frac{r_u^j}{\bar{r_u}}$$

and therefore

$$\bar{s_u} = \frac{1}{n}\sum_{j=0}^{n} \frac{r_u^j}{\frac{1}{n}\sum_{i=0}^{n} r_u^i} = \frac{1}{n}\sum_{j=0}^{n} \frac{n}{1}\frac{r_u^j}{\sum_{i=0}^{n} r_u^i}$$

$$= \sum_{j=0}^{n} \frac{r_u^j}{\sum_{i=0}^{n} r_u^i} = \frac{\sum_{j=0}^{n} r_u^j}{\sum_{i=0}^{n} r_u^i}$$

$$= 1$$

subject	config	test	Slowdown mean	Slowdown std	runtime mean	runtime std
calculator	Carving	0	1.00	0.05	0.11	0.01
calculator	Carving	1	1.83	2.02	0.00	0.00
calculator	Carving	total	1.41	1.48	0.06	0.06
calculator	Coverage	0	0.99	0.02	0.11	0.00
calculator	Coverage	1	1.72	2.72	0.00	0.00
calculator	Coverage	total	1.36	1.95	0.06	0.06
calculator	Uninstrumented	0	1.00	0.22	0.11	0.02
calculator	Uninstrumented	1	1.00	2.85	0.00	0.00
calculator	Uninstrumented	total	1.00	2.02	0.06	0.06

Table 5.3: Mean of the slowdown for all configurations on the calculator.

BASILISK's instrumentation leads to a slowdown of more than 20x.

The reason for the extreme slowdown for `jq` is in the implementation. `jq` has 1169 functions in 108165 lines of code. While `grep` has more code, 191093 lines, it has just 478 functions. This means that for `jq`, the instrumentation reports a lot more function calls, which means more `Called` events. Also, parameters are dumped for each function call, so more function calls means more parameter reporting events as well. As a result, `jq` generates 349129.5 events on average [6], while `grep` 6d952be emits just 36975 events.

In addition to this, `jq` moves rich structures, rather than simple strings or values. This means that more `Area` events are required. `Area` events contain more data than other events, and therefore require more transmission time.

All in all, it has to be observed that BASILISK leads to a dramatic slowdown. This may be acceptable for runs that serve to generate examples to learn from, but would be inacceptable even for test runs.

5.4.2 Evaluation Subjects

BASILISK is designed to work with programs under test written in C. Also, BASILISK requires instrumentation within the subjects, and, due to the fact that BASILISK is implemented based on LLVM, it requires the subjects to be compiled to LLVM IR rather than machine code. No program is distributed in this format, and therefore I had to compile all subjects from source, using the CLANG compiler. This proofed to be difficult. Most programs are distributed with a build script that assumes the GCC compiler, and while some build systems make it easy to use a different compiler, not all do, and not all projects use an off-the-shelf build system in the first place. In addition to the battle against custom build scripts that were not prepared for usage with a different compiler or custom

[6] Over 100 runs. As all runs perform the same operations, all runs should generate the same number of events, however, the timeout may trigger earlier for some runs.

subject	config	test	Slowdown		runtime	
			mean	std	mean	std
jq	Carving	0	19.27	0.01	1.00	0.00
jq	Carving	1	19.21	0.04	1.00	0.00
jq	Carving	total	19.24	0.04	1.00	0.00
jq	Coverage	0	19.26	0.00	1.00	0.00
jq	Coverage	1	19.20	0.00	1.00	0.00
jq	Coverage	total	19.23	0.03	1.00	0.00
jq	Uninstrumented	0	1.00	0.06	0.05	0.00
jq	Uninstrumented	1	1.00	0.05	0.05	0.00
jq	Uninstrumented	total	1.00	0.06	0.05	0.00

Table 5.4: Mean of the slowdown for all configurations on jq.

compiler options, some programs used gcc-specific language extensions, and would not compile with LLVM/CLANG. Even after winning the battle against the build system, programs which use a custom memory allocation mechanism are often incompatible with BASILISK's instrumentation. In the end, I found 6 programs which I can use with BASILISK:

- Four of the subjects are part of *GNU coreutils*, a collection of standard command line tools which is used, e.g. on Linux. The `cut` program reads text from a file and outputs substrings, as specified by the user. The `paste` program can be used to merge lines from different text files. The `tac` command reads a file and outputs it in reversed order. `b2sum` computes a message digest, some kind of checksum, from an input file.
- `sed` is a stream editor that applies a list of user-specified commands on its input and outputs the resulting text.
- The last subject is the `dc` program. `dc` is a programming language with arbitrary-precision floating-point arithmetic. `dc` uses reverse polish notation, where the operator follows its operands: `"1 2 +"` yields the output `3`. `dc` has registers, which can be used as variables.

In Table 5.9, I report the lines of code for each subject. Especially for the programs from *GNU coreutils*, the source code repositories do contain additional code. I counted only lines of code that are in functions that are reachable from the main method of the respective program.

5.4.3 System Testing

In this section, I answer research question 1, namely, how BASILISK compares to another system test generator.

For this experiment, BASILISK and FUZZILISK are integrated with RADAMSA. My BASILISK and FUZZILISK implementation than invoked RADAMSA, to generate

subject	config	test	Slowdown mean	Slowdown std	runtime mean	runtime std
grep 6d952be	Carving	0	10.22	0.64	0.05	0.00
grep 6d952be	Carving	1	1.00	0.00	0.50	0.00
grep 6d952be	Carving	total	5.61	4.64	0.28	0.23
grep 6d952be	Coverage	0	1.45	0.71	0.01	0.00
grep 6d952be	Coverage	1	1.00	0.01	0.50	0.00
grep 6d952be	Coverage	total	1.22	0.55	0.26	0.25
grep 6d952be	Uninstrumented	0	1.00	0.43	0.01	0.00
grep 6d952be	Uninstrumented	1	1.00	0.01	0.50	0.00
grep 6d952be	Uninstrumented	total	1.00	0.30	0.25	0.25

Table 5.5: Mean of the slowdown for all configurations on grep 6d952be.

10 additional system tests for each seed test. Then, they carve unit tests from all system tests. Afterwards I select the unit test for the function that still contains most uncovered code, and the process of parameterizing, unit-level analysis and lifting is applied to this test. Once this is done, the next unit test will be selected and processed. Generated system tests are executed immediately.

In my experiments, I used a time limit of 15 minutes. My prototype runs each system test directly after creating it, so the output is coverage information. Radamsa only generates test inputs. For that reason, comparing directly is not fair. While, for my tool, the time limit includes test executions, Radamsa needs additional time to execute the generated system tests.

In order to mitigate this difference, I had Radamsa generate system tests in batches of 10 tests each and executed each batch before generating the next one, until the time limit was exceeded. This means that for Radamsa, as for Basilisk and Fuzzilisk, system test execution time is included in the time limit. I repeated each run 5 times, with 5 different random seeds. I supplied all three tools with the same, hand-written seed tests.

For Fuzzilisk, both the unit-level analysis and system-level fuzzing is randomized. For Basilisk, unit-level analysis is deterministic, but system-level testing is still randomized. For all subjects, I measure the branch coverage over time.

Coverage

Table 5.10 summarizes the branch coverage obtained, as a mean over five runs. In all subjects but `dc`, the coverage achieved by Basilisk improves over Radamsa alone, showing a clear benefit of the boosting of system-level tests with unit-level test generators.

> For all subjects but `dc`, the Basilisk booster of system-level test generation (Radamsa) and unit-level test generation (Klee) increases coverage over

subject	config	test	Slowdown mean	Slowdown std	runtime mean	runtime std
grep 8f9106c	Carving	0	10.18	0.78	0.05	0.00
grep 8f9106c	Carving	3	1.16	0.15	0.13	0.02
grep 8f9106c	Carving	total	5.67	4.56	0.09	0.04
grep 8f9106c	Coverage	0	1.39	0.52	0.01	0.00
grep 8f9106c	Coverage	3	1.00	0.04	0.12	0.00
grep 8f9106c	Coverage	total	1.20	0.42	0.06	0.05
grep 8f9106c	Uninstrumented	0	1.00	0.47	0.01	0.00
grep 8f9106c	Uninstrumented	3	1.00	0.13	0.12	0.01
grep 8f9106c	Uninstrumented	total	1.00	0.34	0.06	0.06

Table 5.6: Mean of the slowdown for all configurations on grep 8f9106c.

> RADAMSA alone between 5% (`cut`) and 59% (`sed`).

`dc`, being an arbitrary-precision calculator, stores numbers as strings of bytes, which my implementation cannot associate with input fragments and thus neither map nor lift. This claim is further supported by Table 5.11, showing that BASILISK analyzes just one unit test for `dc`. All other tests had no parameterizable inputs.

> Non-standard data representations of strings and numbers, as in `dc`, need special implementations for mapping and lifting.

Coverage Over Time

Does the increased coverage also translate into time savings? Figure 5.6 shows a plot of the branch coverage achieved over time for `paste`. The diagrams list time on the x-axis, and the coverage achieved so far by BASILISK or RADAMSA respectively on the y-axis. This is a typical representative; the other subjects show similar plots. All plots an be found in Appendix B.

BASILISK initially has worse coverage than RADAMSA. This is due to the extra overhead of carving. However, for all subjects, there is a point in time when BASILISK outperforms RADAMSA. Typically, this happens at about 8 minutes. Note that BASILISK's coverage after 8 minutes also outperforms RADAMSA coverage after 15 minutes. This means that with BASILISK, one can achieve the same coverage in about half of the time. Again, `dc` is the exception.

> In my experiments, BASILISK achieves the same coverage as RADAMSA in about half the time.

Figure 5.7 shows coverage over time for `sed`. This is the only case where coverage does not increase slow and steady, but in a massive jump. `sed` accepts an

subject	config	test	Slowdown		runtime	
			mean	std	mean	std
grep c32c042	Carving	0	8.29	0.77	0.04	0.00
grep c32c042	Carving	4	11.59	4.52	0.06	0.02
grep c32c042	Carving	total	9.94	3.63	0.05	0.02
grep c32c042	Coverage	0	1.45	0.53	0.01	0.00
grep c32c042	Coverage	4	1.27	0.64	0.01	0.00
grep c32c042	Coverage	total	1.36	0.59	0.01	0.00
grep c32c042	Uninstrumented	0	1.00	0.58	0.01	0.00
grep c32c042	Uninstrumented	4	1.00	0.76	0.01	0.00
grep c32c042	Uninstrumented	total	1.00	0.67	0.01	0.00

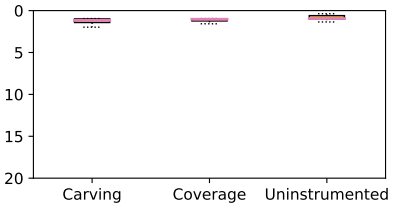
Table 5.7: Mean of the slowdown for all configurations on grep c32c042.

subject	config	test	Slowdown		runtime	
			mean	std	mean	std
grep 70e2361	Carving	0	9.77	0.50	0.05	0.00
grep 70e2361	Carving	2	16.10	9.05	0.10	0.05
grep 70e2361	Carving	total	12.94	7.14	0.07	0.04
grep 70e2361	Coverage	0	1.36	0.50	0.01	0.00
grep 70e2361	Coverage	2	1.27	0.69	0.01	0.00
grep 70e2361	Coverage	total	1.32	0.61	0.01	0.00
grep 70e2361	Uninstrumented	0	1.00	0.67	0.01	0.00
grep 70e2361	Uninstrumented	2	1.00	1.35	0.01	0.01
grep 70e2361	Uninstrumented	total	1.00	1.06	0.01	0.01

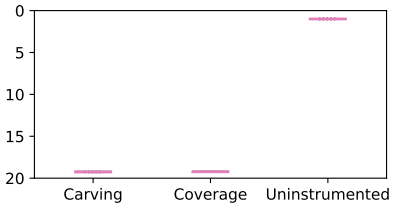
Table 5.8: Mean of the slowdown for all configurations on grep 70e2361.

Table 5.9: Evaluation subjects

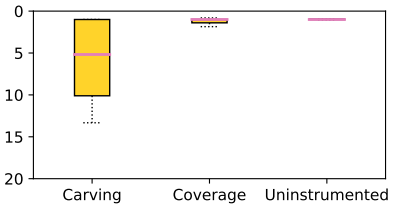
Subject	**LoC**	**Functions**	
b2sum	1228	115	checksum calculation
paste	662	79	text processor
tac	987	111	text processor
dc	1997	136	arbitrary-precision calculator
cut	1346	127	text processor
sed	2715	215	text processor



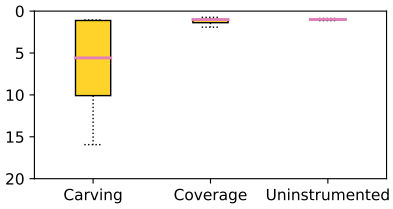
(a) calculator



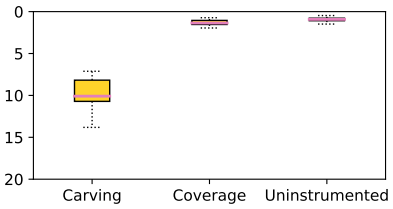
(b) jq



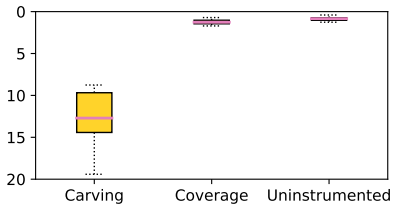
(c) grep 6d952be



(d) grep 8f9106c



(e) grep c32c042



(f) grep 70e2361

Figure 5.5: Slowdown for each subject and configuration.

Table 5.10: Coverage reached by system tests for each subject.

Subject	BASILISK	FUZZILISK	RADAMSA	Improvement (Basilisk)
b2sum	41.57%	30.60%	34.87%	1.19
paste	41.93%	38.32%	37.28%	1.12
tac	33.55%	31.98%	29.92%	1.12
dc	19.13%	19.13%	35.73%	0.54
cut	22.58%	21.36%	21.61%	1.05
sed	30.62%	24.36%	19.24%	1.59

Table 5.11: Functions covered in unit tests.

Subject	# functions	BASILISK	FUZZILISK
b2sum	204	17	12
paste	152	21	9
tac	177	18	9
dc	154	1	3
cut	205	37	7
sed	290	38	12

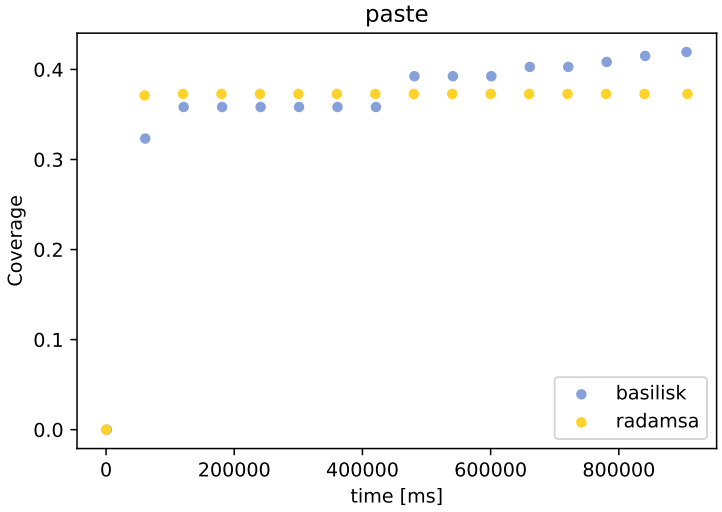


Figure 5.6: Coverage over time for Basilisk and Radamsa on `paste`

input language, which consists of several commands. The jump happens when Basilisk (or to be precise, Klee) discovers the `q` command, which terminates `sed`. It proceeds to generate a lot of tests which contain `q` and thereby covers a lot of new code. Radamsa did not manage to trigger the `q` command in my experiments.

Symbolic Testing vs. Random Testing

Fuzzilisk, with a unit-level random fuzzer, outperforms Radamsa only on `paste` and `tac`, and is never better than Basilisk, with Klee inside; it also covers fewer functions. This shows a clear benefit of using symbolic testing techniques at the unit level.

> At the unit level, symbolic testing (Basilisk with Klee) consistently covers more code than random testing (Fuzzilisk).

5.4.4 Lifting Performance

Table 5.12 shows how many tests were lifted to the system level within Fuzzilisk. This number is rather low, less than 1% of the tests in total. The percentage of effective lifts is even lower. I counted a test as *effective* if it increases coverage in the program under test. This means that Fuzzilisk in fact tries many execution paths that do not lead to new coverage, or crashes. It would be impractical to test all those execution paths with system tests. These numbers shed a light on the usefulness (or better: non-usefulness) of unguided random tests at the function level, and motivate their lifting into a system test to validate their results.

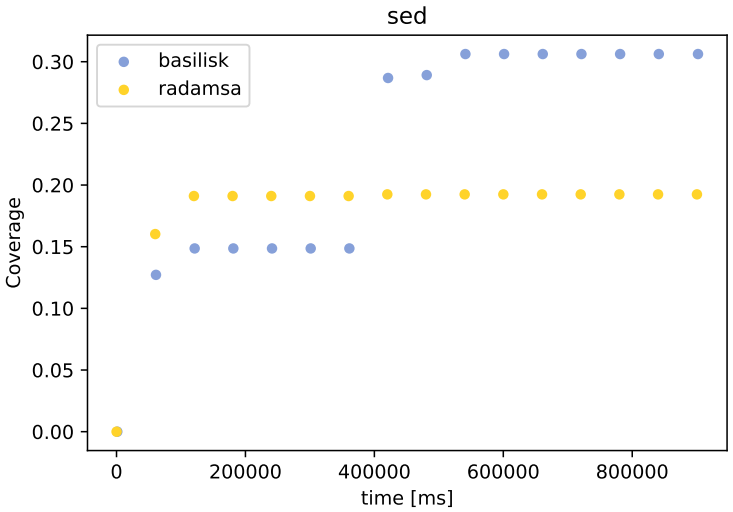


Figure 5.7: Coverage over time for Basilisk and Radamsa on sed

Table 5.12: Unit Tests selected for lifting in Fuzzilisk

Subject	# Unit Tests	# Lifted Tests	% lifted	% effective
b2sum	9005075.20	53.40	0.0007%	0.0001%
paste	4654504.60	37.60	0.0008%	0.0002%
tac	2543487.80	35.20	0.0014%	0.0003%
dc	475426.60	14.20	0.0030%	0.0000%
cut	1167711.00	29.60	0.0026%	0.0002%
sed	4656782.00	105.20	0.0023%	0.0008%
total	3750497.87	45.87	0.0012%	0.0003%

Table 5.13: Effectiveness of lifting in Basilisk

Subject	# Lifted Tests	% effective
b2sum	876.40	3.9049%
paste	757.80	3.6158%
tac	870.60	2.2169%
dc	266.60	0.0000%
cut	393.40	2.5420%
sed	670.00	12.3893%
totals	639.13	4.5322%

Table 5.14: Speed advantage of System Tests vs. Unit Tests

Subject	Median Runtime[ms]		Mean
	System Tests	Unit Tests	speed-up
b2sum	39	2	1165.00
paste	114	0	1196.30
tac	160	4	1393.43
dc	1620	0	5005.40
cut	32	0	281.21
sed	98	0	1684.92
total	73	0	495.06

> In my experiments with RADAMSA and a unit-level fuzzer, less than 0.001% of random function invocations that increase coverage or cause failures also do so when lifted back to the system level.

Being a symbolic tester, KLEE always executes multiple program path at the same time, and only reports those that trigger a bug or cover new code. Thereby, the full number of unit tests, as given in Table 5.12, is not available for the combination of RADAMSA and KLEE (which I call BASILISK). Table 5.13 thereby only gives the number of lifted tests (path reported by KLEE), and the effectiveness.

Also, BASILISK gives a different picture here: Except for `dc`, all subjects show a lifting effectiveness above 2%, with a maximum of 12% for `sed`. This again provides evidence for the hypothesis that FUZZILISK's performance is mainly a problem with the unit-level fuzzing tool.

> In my experiments with RADAMSA and KLEE, 4.53% of paths at unit level that increase coverage or cause failures also do so when lifted back to the system level.

5.4.5 Unit Testing

Unit tests typically run much faster than system tests, as they execute much less code. Table 5.14 shows how much faster they are within my FUZZILISK prototype. On average, a unit test is **495 times faster** than a system test for the same program. This means that if I want to test one single function only (say, because it has recently changed), I can save a huge amount of time, or spend the same time on more thorough testing.

> The carved unit tests run several orders of magnitude faster than the original system test.

This result is important. In Section 5.4.1, I found that:

For reference

Basilisk's instrumentation leads to a slowdown of more than 20x.

See Section 5.4.1

Still, Basilisk outperforms Radamsa. The reason is the speed difference between system-level and unit-level analysis: Klee can examine tens of unit-level path in the time that Radamsa needs for one system test. Therefore, the tests that are lifted, and actually executed on the system-level, are more effective than what Radamsa does. As long as the speed up of unit-level analysis, compared to system test executions, is larger than the slowdown for instrumentation, Basilisk is effective.

5.5 Communicating Interest in Input Parts

In this section, I will attempt to integrate Basilisk into Alhazen. As described within Section 5.1, the idea is that Basilisk can identify decision borders, and inform Alhazen, by providing test cases which exactly mark the border of a decision. In order to do so, Basilisk needs to focus on those parts of the input which are considered relevant by Alhazen. Those parts will be considered as *symbolic* within the symbolic execution later on.

Alhazen expresses interest in input parts by making them part of the hypothesis. In Section 3.1, I showed how Alhazen's hypothesis, the decision trees, are turned into predicate sets. Each predicate set corresponds to one decision path within the tree. A decision tree learner only generates tree path which belong to an observed sample. That is because the learner only forms a hypothesis about parts of the input space where at least one witness, that is one sample, exists. Therefore, I can identify one sample for each predicate set. This only holds if the generation of additional predicate set to explore beyond the known parts of the search space, described in Section 3.1.3, is not used. For this reason, I do not use this process when I use Basilisk.

Basilisk subsequently uses the grammar to obtain a parse tree for the sample. As in the entire thesis, this parse tree may be ambiguous. If it is, I choose one of the parse trees at random. Then, I identify the non-terminals that are used within the predicate set for this sample. Next, I obtain the leaf word of the tree, and mark every part that was derived from a node labeled with a non-terminal present in the predicate set as symbolic.

The parse tree for the running example, `"sqrt(900)"`, is shown in Figure 5.8. The predicate set for this sample is $\{\text{max-numeric}(\langle number \rangle) >= 445\}$. Therefore, the only relevant non-terminal symbol is $\langle number \rangle$. As can be seen in the parse tree in Figure 5.8, the leaf word for $\langle number \rangle$ is `"900"`. Therefore, this part of the input sample is marked as symbolic.

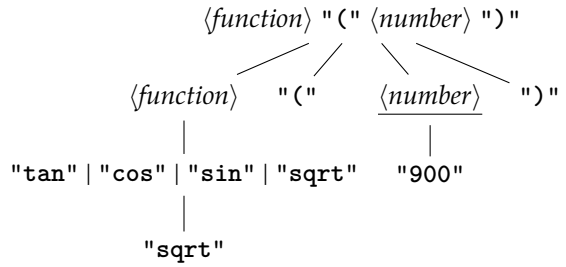
5.5.1 Search Space Exploration

At first, one might get the impression that this means that Basilisk won't be able to explore those areas. However, Basilisk only receives information about the non-terminal symbols used. Basilisk does not learn about the concrete predicate.

Predicate set: {max-numeric(⟨*number*⟩) >= 445}

Input: `"sqrt(900)"`

Parse tree:



Symbolized input: `"sqrt(900)"`

Figure 5.8: This example shows which parts of the input become symbolic for a given input and predicate set. Symbolic parts are underlined.

Listing 5.5: A unit test for `msqrt`, carved from the calculator in Listing 5.2.

```
1  void state_based_test() {
2        msqrt(900);
3  }
```

Therefore, Basilisk will explore all variations of this value, and therefore generate values outside the area defined by the tree.

5.5.2 Limitations of Basilisk

When combined with Alhazen, Basilisk occupies the role of the generator presented in Chapter 3. It is used within the feedback loop to generate new samples to learn from.

In Section 3.1.3 I explained how Alhazen needs samples *outside* the known parts of the search space. In this section, I reported that I do not use the mechanism that was introduced to generate those. At first glance, this is a disadvantage for Basilisk.

However, there is an important difference between Basilisk and the generator from the previous section: While the previous generator made use of the entire hypothesis, including values, Basilisk looks at the used nonterminals only. That is, for a predicate sets {max-numeric(⟨*A*⟩) < 10}, Alhazen's generator would

Listing 5.6: The unit test in Listing 5.5, with `"900"` made symbolic.

```
1  void state_based_test() {
2     int32_t i = symbolic_int();
3        msqrt(i);
4  }
```

For reference

Definition 14: Coincidental Correlation
If, due to properties of the generator algorithm, the values of two features always correlate, I call this *coincidental correlation*.

See Section 2.7.3

only generate samples with $\langle A \rangle$ smaller than 10, and the additional predicate set helps to explore samples with $\langle A \rangle$ larger than 10 as well. Basilisk ignores the comparison, and generates values for $\langle A \rangle$ that exercise all branches it discovers in the code. Introducing a predicate set $\{\text{max-numeric}(\langle A \rangle) \geq 10\}$, as I did for Alhazen, would therefore be unneccessary. Basilisk, which is deterministic and ignores the comparison operator and constant, would generate the same samples twice.

This, however, also poses a threat. If Alhazen, in the next iteration, changes nothing but the constants and operators in the tree, Basilisk generates the same samples again. This may mean that Alhazen encounters a dead end: It receives no further evidence to refine its hypothesis, just more data it already knew.

5.6 Evaluation

In this section, I will evaluate Basilisk as a part of Alhazen. This means that Alhazen uses a different set of training samples within the feedback loop. Instead of following the structure of the tree, sample generation follows the structure of the code now. The effect of this is unclear: In the introduction, I showed how the relevant sample can be found *earlier*, therefore, it can be expected that Alhazen comes up with a correct hypothesis *faster*. This leads to my first research question:

1. Do Basilisk-generated samples lead to faster hypothesis learning?

The samples generated with Basilisk, in contrast to the ones generated with Alhazen's own generator, outline the decision boundaries within the code. This means that it is possible that the hypothesis does as well, and precision and accuracy improve. Therefore, my second and third research question are:

2. Do Basilisk-generated samples lead to more precise hypothesis?

3. Do Basilisk-generated samples lead to more accurate hypothesis?

Alhazen originally used the same generator algorithm within the feedback loop and for its generator. This means that coincidental correlations affect both processes. If the feedback loop learned a coincidental correlation, rather than a real cause, the generator would generate the same coincidental correlation, leading to high precision and accuracy for Alhazen as a geneartor. Therefore, I examine whether Alhazen still does well as a generator when Basilisk is used within the feedback loop.

4. Can Alhazen serve as a generator when Basilisk was used in training?

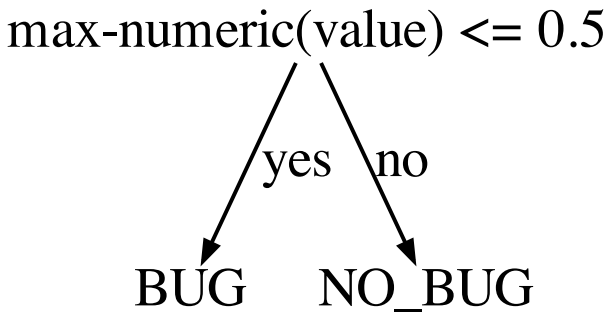


Figure 5.9: Final tree for ALHAZEN with BASILISK as a generator.

5.6.1 Evaluation Setup

I ran ALHAZEN with BASILISK as a generator with the same parameters as in Section 3.3. I also re-ran experiments with ALHAZEN's own generator, because the subjects are slightly different (see Section 5.4.2). In contrast to the experiments in the previous chapter, I removed duplicate samples in between iterations. That is, when BASILISK generated a sample that was identical to a sample generated in a previous iteration, I removed that sample. The timeout was again 1 hour or 40 iterations, whatever happened first. I also terminated when no new sample was generated in an iteration.

5.6.2 Looking at the Calculator

In this section, I will do a close analysis of the results for one subject, the calculator. The calculator is not a real subject, but was written as an example. It should be familiar to the reader, as it has been used as an example throughout the thesis.

Running on the calculator, ALHAZEN terminates after 2 iterations, because the second iteration generated only samples that were known from previous iterations. The final tree is shown in Figure 5.9. It, correctly, points out that the crash happens when the value is smaller or equals 0.5. The calculator accepts only integers, so 0.5 is as close to the real value as ALHAZEN can get. However, there is a crucial fact missing: The crash only happens when the function is `sqrt`.

It is easy to understand why: The initial samples provided where:

```
1       sqrt(-900)
2       sin(4)
```

ALHAZEN's initial hypothesis was that the crash happens when ⟨*value*⟩ is smaller than -445. As only ⟨*value*⟩ is used in the predicate set, BASILISK marks only the value as a parameter. There are two path in the tree, and therefore two predicate sets, which means that BASILISK attempts to modify both samples. The newly generated samples are

```
1       sqrt(0)
2       sqrt(3)
3       sqrt(1)
4       sqrt(-900)
5       sin(4)
6       sqrt(-900000)
7       sqrt(2)
```

While there are quite a few new samples for `sqrt`, there is no new sample for `sin`. That is because the code for `sin` does not contain branching. The existing sample reaches all code reachable via `sin`, and therefore BASILISK, or more precisely KLEE, doesn't feel it necessary to generate more samples. But then, ALHAZEN does not learn that there are no crashes with `sin` and a small value, and therefore it does not include this in its hypothesis.

The next hypothesis is the final hypothesis already, which again contains predicates over ⟨*value*⟩ only. BASILISK again uses only the value as a parameter, and cannot generate new information.

This is exactly the problem I expected in Section 5.5.2. However, in this section I suspected that it happens due to the fact that BASILISK does not use values. This example would not profit from using values: BASILISK found the decision boundary perfectly, it got ALHAZEN from the hypothesis $\langle value \rangle <= -445$ to $\langle value \rangle <= 0.5$ within just one iteration. Further refinement of the hypothesis requires to look at a different non-terminal, rather than more values.

Despite the dead end BASILISK encountered, it reached the correct decision boundary with just one iteration. This is a lot faster than with ALHAZEN's own generator.

> BASILISK can help to find numeric boundary values faster, but runs the risk of missing other constraints.

5.6.3 Do BASILISK-generated samples lead to faster hypothesis learning?

With the result from the previous section, let's have a look at how many iterations ALHAZEN with BASILISK uses in general. While the research question asked about runtime, I am looking at the number of iterations instead. BASILISK, as explained before, is not a highly optimized implementation of the underlying idea. Looking at execution time therefore would likely not yield a relevant result: Instead of recognizing merit of the idea, I would rediscover shortcomings of the implementation. Therefore, I decided to compare the number of iterations, rather than the execution time.

subject	configuration	basilisk		search-based	
		mean	std	mean	std
calculator	kpath	2.50	1.00	40.00	0.00
calculator	sets	2.75	0.96	35.00	10.00
calculator	twoInputs	2.50	0.58	31.75	13.38
grep 3220317a	kpath	2.00	0.00	2.50	0.58
grep 3220317a	sets	1.00	0.00	11.50	2.08
grep 3220317a	twoInputs	2.00	0.00	15.50	7.90
grep 5fa8c7c9	kpath	3.50	1.00	34.00	6.98
grep 5fa8c7c9	sets	1.00	0.00	39.50	1.00
grep 5fa8c7c9	twoInputs	2.00	0.00	38.50	3.00
grep 7aa698d3	kpath	2.00	0.00	3.00	1.41
grep 7aa698d3	sets	1.00	0.00	5.00	1.83
grep 7aa698d3	twoInputs	2.00	0.00	27.75	13.30
grep c96b0f2c	kpath	2.25	0.50	11.25	11.95
grep c96b0f2c	sets	1.50	1.00	4.00	0.82
grep c96b0f2c	twoInputs	2.00	0.00	20.50	12.87
jq	kpath	2.50	0.58	40.00	0.00
jq	sets	1.00	0.00	26.50	3.32
jq	twoInputs	2.00	0.00	32.00	10.61
total	kpath	2.46	0.78	21.79	17.66
total	sets	1.38	0.82	20.25	14.98
total	twoInputs	2.08	0.28	27.67	12.38

Table 5.15: Number of iterations per subject and configuration.

The data can be found in Table 5.15. All values are averaged over 4 runs. For the search-based generator, ALHAZEN performs between 20.25 (sets) and 27.67 (twoInputs) iterations on average. In this experiment, I stopped the feedback loop as soon as no new samples were generated. The sets configuration, which receives the highest number of samples from the start, reaches this point first. This does not mean that those runs exercised the entire search space. The search spaces in this experiment are that large that they cannot be examined entirely. It means that ALHAZEN exercised a sufficient number of samples from the relevant part of the search space.

BASILISK performs on average between 1.38 (sets) and 2.46 (kpath) iterations. This is smaller by a factor of almost 10 when compared to the search-based generator. This confirms what I observed on the calculator in the previous section.

> BASILISK generates no new information after just a few iterations.

It remains to see whether all subjects behave as the calculator does, and miss large parts of the search space.

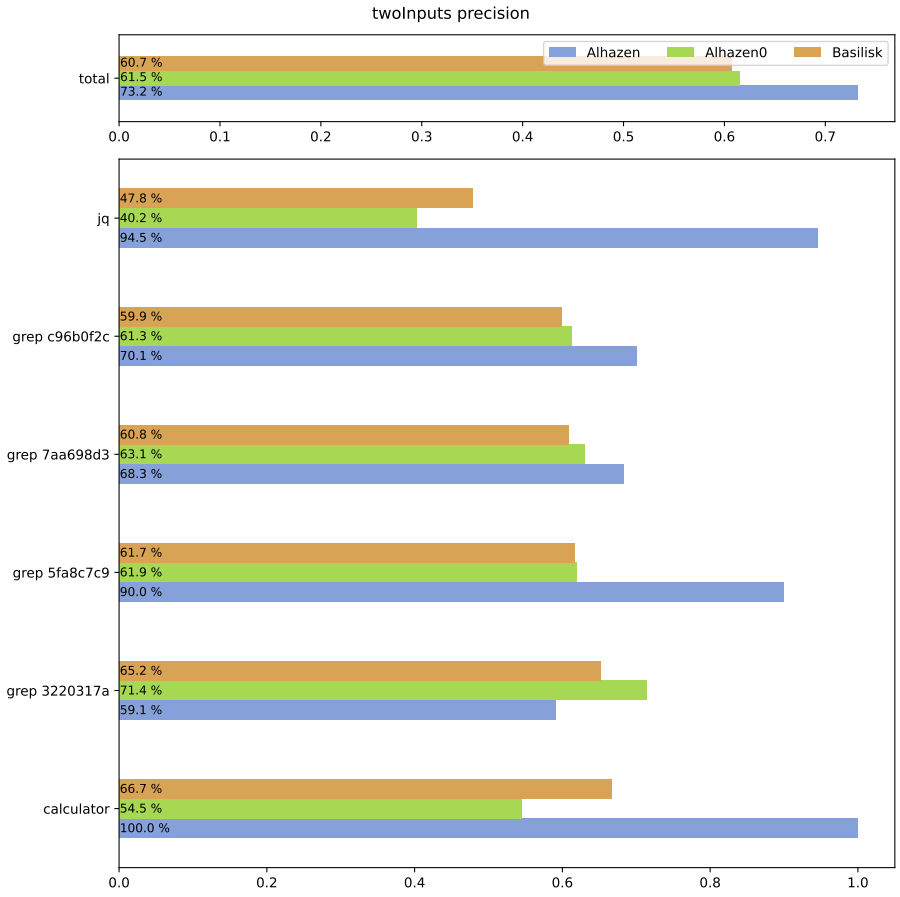


Figure 5.10: Precision of ALHAZEN in the twoInputs configuration.

5.6.4 Precision and Accuracy of BASILISK-generated hypothesis

In the previous section, I found that ALHAZEN requires less iterations with BASILISK as a generator. However, it was not clear whether it performs better, or just faster. Therefore, I need to have a look at the data for the next research questions.

Figure 5.10 shows precision for all subjects. Even for ALHAZEN, those values are not as good as those in Section 3.3.1. The reason is the new termination criterion. If there is no new sample in an iteration, the version of ALHAZEN I used in Chapter 3 still performs another iteration. ALHAZEN has no new information for the next iteration, but the decision tree learner receives a new random seed [7]. Therefore, it generates a different decision tree, and has a new chance to generate additional samples. Within this section, I terminated the feedback loop as soon as an iteration generated no new samples. This leads to less data to learn from, and lower precision. Except for `grep 3220317a`, ALHAZEN still

[7] A new random seed is derived from a master seed for each iteration, different runs of ALHAZEN use different master seeds

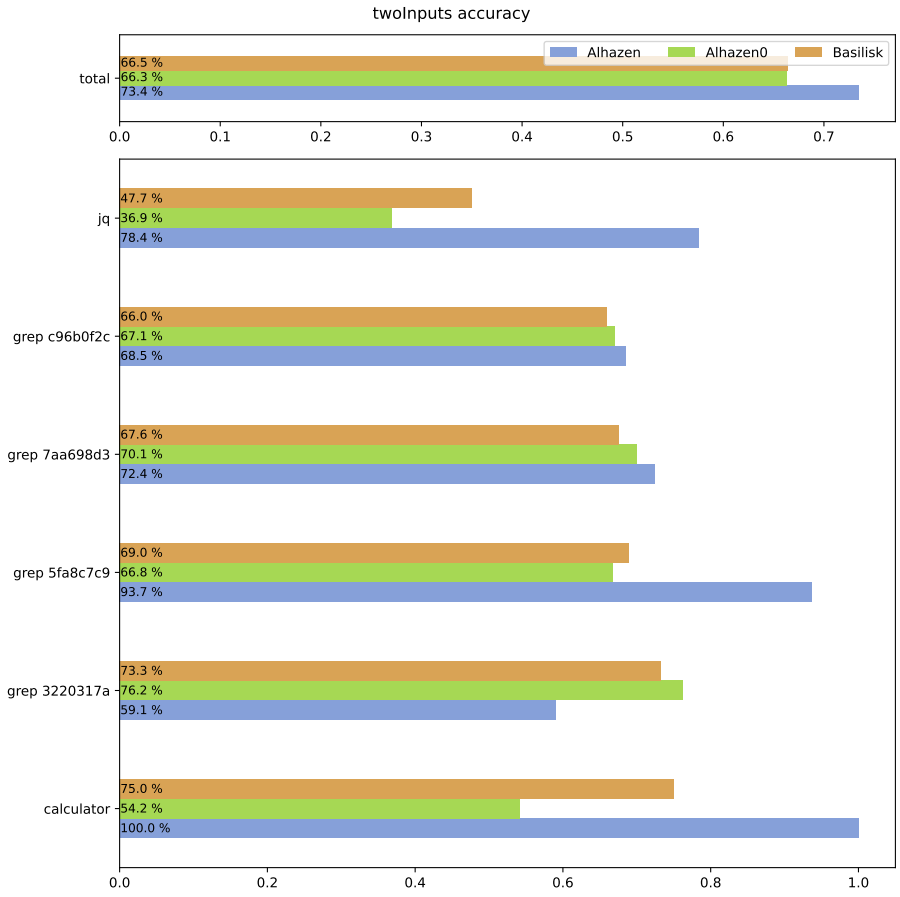


Figure 5.11: Accuracy of ALHAZEN in the twoInputs configuration.

outperforms ALHAZEN0, so the idea of a feedback loop still works. On this subject, BASILISK even outperforms ALHAZEN. For all other subjects, BASILISK performs worse than ALHAZEN. Even ALHAZEN0 is better than BASILISK in most cases. This means that samples generated with BASILISK solidify wrong believes within the tree, rather than helping to disproof them.

Data on accuracy, in Figure 5.11, confirms this result. Except for `grep 3220317a`, ALHAZEN outperforms both other tools. BASILISK outperforms ALHAZEN0 on two subjects, `jq` and `grep 5fa8c7c9`, proofing that it can generate valuable information. But then, ALHAZEN0 still outperforms on three subjects.

> BASILISK on its own is not a suitable generator for ALHAZEN.

However, for `jq`, BASILISK performs at least better than ALHAZEN0. With this result, I can conform my observation on the calculator example: BASILISK can generate some information fast, but misses other important pieces of information. Likely, a tool which uses BASILISK, and switches to ALHAZEN's search-based generator as soon as BASILISK generates no new samples any more would combine

advantages of both approaches. However, evaluating this approach is out of scope for this thesis.

5.7 Conclusion

In this section, I will take a final look at the results achieved by Basilisk. The first section will take a closer look at the implementation, before the results are summarized in the last section.

5.7.1 The Design of Basilisk

The slowdown observed in Section 5.4.1 shows that my Basilisk implementation is not fit for real-world usage. However, if it is decided which functions to carve out before the test is executed, it would not be necessary to dump everything. Events could only be emitted for relevant program parts. In theory, this leads to a massive speed-up for Basilisk, I cannot tell whether it does in practise, because I never tried.

An even more optimal implementation could stop the execution of the system test, similar to a breakpoint in a debugger, at the invocation of a function and perform unit tests within the context of the original program. This would eliminate the need to carve altogether, and is done by tools such as RedQueen[5] or Driller[68]. It would, however, also mean that developers cannot inspect the carved unit tests manually.

5.7.2 Summary

In this chapter, I introduced Basilisk. Basilisk realizes the idea of a system test booster: It *carves* system tests from unit test executions, performs *unit-level analysis* of the carved tests, and *lifts* the results to the system level. In doing so, it reaches more code than system-level random testing. Moreover, it does so faster. However, there is a catch: If program-internal datastructures are beyond what Basilisk can understand, Basilisk fails.

When combined with Alhazen, Basilisk results are ambiguous. While individual examples show that the basic idea works, the possible advantages do not materialize. That is, Basilisk analyses individual units and lifts the analysis results successfully, but Alhazen can not learn from them. The reason is that Basilisk focuses on a specific part of the input space, and ignores everything else. In doing so, it misses important information. Future work should investigate whether Basilisk can be combined with Alhazen's own generator.

Chapter 6

Closing Remarks

In this section, I'll conclude my thesis by reflecting on the insights, and commenting on my contributions to the questions raised in the introduction. In Section 6.1, I will compare my contributions to work others have done in the same field.

6.1 Related Work

In this section, I'll take a look at work that is related to Alhazen. The related work is organized into six sections:

1. Section 6.1.1 looks at work which attempts to automatically generate grammars. Those could be used with Alhazen, and therefore simplify the setup.
2. One component of Alhazen is a grammar-based input generator. Section 6.1.2 looks at other approaches for grammar-based input generation.
3. A possible use case for Alhazen is to use Alhazen as a debugging aid. Section 6.1.3 looks at other approaches which can be used this way. The work on abstract failure-inducing inputs by Gopinath et al. [25] is closely related to Alhazen, and therefore has its own section.
4. Section 6.1.5 looks at specification mining, a technique which generates models for program behavior.
5. Lastly, I'll consider techniques which are related to Basilisk's approach of symbolic analysis of subprograms.

6.1.1 Grammar Mining

Alhazen utilises a grammar for dissecting the program inputs. Features are defined based on parse trees, and therefore grammars. As for other grammar-based approaches, finding a suitable grammar — or writing one from scratch — may be difficult. Also, a grammar may need to be updated when the program under test evolves and the input format changes. Research on *Grammar Mining*

attempts to extract grammars automatically. It therefore simplifies the setup of ALHAZEN.

Bastani et al. [6] use a *black-box approach*. They construct a context-free language which contains all observed input words. Afterwards, they apply a number of operations, adding quantifications or turning literals into more permissive regex, which generalize the candidate language. Afterwards, grammar-based test generation is used to check whether the relaxed grammar still describes the program behavior. Clearly, the obtained grammar can be used with ALHAZEN, however, there is also conceptual overlap between their tool and ALHAZEN. Both tool use a feedback loop to refine their models, and both tools generate a model which describes the observations. However, the details are quite different. While ALHAZEN uses its feedback loop to refine a model over an existing grammar, Bastani et al. [6] refine the grammar itself. Thus their approach, using a context-free grammar as model, offers different expressiveness than ALHAZEN, which expresses its hypothesis in terms of features. Also, they attempt to generate an input grammar for the entire program, rather than just explaining part of its behavior.

White-box approaches take a different approach to the same problem: They observe internal program behavior, namely control-flow (e.g. AUTOGRAM [35]) or dynamic data-flow(e.g. MIMID [23]), and derive a grammar from those interactions. Control-flow based algorithms in this group work with parsers that are written as recursive-descent parsers, and use the control-flow properties of those parsing algorithms to derive a grammar. Data-flow based algorithms use the structure of program internal data structures, and assume that it has some resemblance to the input structure. Both are dynamic analysis, and therefore require input samples. Mathis et al. [47][1] present a generator algorithm which uses data-flow, the same information that AUTOGRAM uses, to guide sample generation. Those samples reach good coverage of the parser under test, and are therefore particularly well suited for white-box approaches to grammar mining.

As with Bastani et al. [6], those approaches generate a grammar, and therefore offer different expressiveness than ALHAZEN. For both approaches, it is unclear whether they can be restricted to the analysis of failing inputs. They rely on the analysis of internal data structures or control-flow. If program execution terminates before internal data structures are initialized, or the control-flow happened, both approaches cannot generate a model. Therefore, their models always describe the entire input space. They are a good fit for obtaining a grammar for a program under test, and applying ALHAZEN with this grammar for bug diagnosis.

6.1.2 Grammar-based Input Generation

One component of ALHAZEN is a grammar-based input generator, which generates inputs that fulfill given predicates. This section examines previous work on grammar-based input generation.

Havrikov and Zeller [31] defined a grammar-based coverage metric, k-path, and developed an algorithm which generates test suites that fulfill this criterion. The inner search of my algorithm, presented in Section 3.2.4, was heavily influenced by this work. Havrikov and Zeller [31] use the grammar graph and

[1] The author of this thesis is one of the "et al."

the distance within the grammar graph to reach targets just as I do, however, I extended the algorithm to be able to handle max-char(), max-qu-length() and max-char-length() predicates, as well as the ability to handle negated exists() predicates. Also, I added the outer search to solve cases that the approach cannot handle on its own.

Pavese et al. [55] use probability distributions to sample context-free grammars for input samples. I used their approach — with a different probability distribution — to generate training data for ALHAZEN. The details are described in Section 2.5.3.

There is more work which generates input samples from a context-free grammar. As an example, Hodován, Kiss, and Gyimóthy [33] generate from the grammar randomly. This is suprisingly difficult, as one needs to make sure that the random process terminates eventually.

All three works present grammar-based generation algorithms. However, to my knowledge, my algorithm was the first one which can fulfill predicates over the grammar while generating samples.

Shortly after my work, Gopinath, Nemati, and Zeller [24] published their work on Input Algebras. Input Algebras formalize operations on grammar. Therefore properties like "The phone number needs to start with +1" can be expressed as formulas, and those formulas can be applied as a grammar transformation, which yields a grammar that generates samples that fulfil this property only. This approach is interesting, and if ALHAZEN's features can be expressed as formulas like this, it offers a different option for sample generation. However, Input Algebras can only express syntactic properties. Expressing a property like "The number needs to be below 121" would be hard, as it is hard to come up with a grammar which can parse numbers below 121 only[2].

Another interesting approach is the work by Steinhoefel and Zeller [67]. They present a language, called ISLA, which allows to express syntactic properties on top of a context-free grammar. They also provide a solver, using an SMT solver within, to fulfil this kind of constraints. Again, this is an interesting alternative to ALHAZEN's generator.

6.1.3 Debugging Aid

In Chapter 4, I positioned ALHAZEN as a tool which can help in debugging. This section examines related work in this area.

A popular approach to aid developers in debugging is *input reduction*. The main idea is that a smaller input leads to a shorter program run, which is easier to analyse with traditional debugging tools such as breakpoints and manual analysis of program states. The goal of those approaches therefore is to determine

[2] But possible:

⟨*number*⟩	→	`"1"` ⟨*s21*⟩ | ⟨*num2*⟩
⟨*num2*⟩	→	⟨*zero-digit*⟩ | ⟨*digit*⟩ ⟨*zero-digit*⟩
⟨*s21*⟩	→	⟨*digit1*⟩ ⟨*zerodigit*⟩ | `"2"` ⟨*digit1*⟩
⟨*digit1*⟩	→	`"1"` | `"0"`
⟨*digit*⟩	→	`/[1-9]/`
⟨*zero-digit*⟩	→	`/[0-9]/`

Beware of this grammar, I have not tested it.

a subset of the input which still reproduces the failure. I used a similar idea when I looked at input space reduction in Section 4.1.

Zeller and Hildebrandt [78] uses a binary search-like process. They remove half of the input in each step, until removing any more character from the input means that the failure does not occur any longer. Later approaches combine this idea with grammars[50, 69], using information from the grammar to determine which parts of the input can be left out.

There are some similarities between ALHAZEN and this work. Both work on system-level inputs, and both use repeated testing, ALHAZEN within its feedback loop and delta debugging-based approaches as a means to test whether an input can be reduced. However, there are also some differences. First of all, ALHAZEN is quite different in the output it provides. Delta debugging provides a reduced input. That is, exactly one input, and this input, while smaller, may still not contain obvious clues to the reason of the bug. ALHAZEN provides a *model*. This can be used to generate more inputs, and also, it may highlight the important properties of those inputs. Most notably, if there are several inputs with the same size which trigger the bug, ALHAZEN can discover all of them and include their information content in the model. A delta debugging-based approach cannot do that. Another difference between ALHAZEN and delta-debugging is that ALHAZEN can also work *without* the feedback loop, relying on preexisting inputs only. While performance is lower, this may still be useful in situations where the program under analysis cannot be executed with generated inputs easily.

Statistical fault localization[65] attempts to highlight the code locations which are most likely to contain the bug. That is, they execute failing test cases and monitor which lines of code get executed. The programming mistake needs to be somewhere in those lines, as a bug cannot manifest unless the lines containing it are executed. Statistical fault localization tools attempt to establish a statistical correlation between code locations and program failures. There is some discussion about whether this is useful for programmers[53], but some works show the usefulness of those approaches for automated program repair[42, 73]. ALHAZEN also uses observations from many program runs to establish statistical associations with program failures. However, while statistical fault localization tools associate failures with code locations, ALHAZEN associates input features with failures. Input features may be easier to interpret than code locations, especially if the user lacks context knowledge about the program under test. A second difference lies in ALHAZEN's feedback loop. ALHAZEN can systematically generate tests which indicate whether a hypothesis is correct. It uses this to refine its hypothesis.

Havrikov, Kampmann, and Zeller [30] shows how an extension of ALHAZEN can be used to link input features to code locations. In future work, this approach and unmodified ALHAZEN could be combined to identify code locations for a bug.

All statistical debugging tools need a large number of test cases to establish meaningful statistical associations, and, also in future work, ALHAZEN might be used to generate those test cases.

Rößler et al. [61] also use test generation to come up with evidence that refutes a pre-existing believe, but their hypothesis is based on internal program state, rather than input features. While ALHAZEN gives a diagnosis such as "The failure occurs if x is smaller than 0", Rößler et al. [61] only point to locations

⟨*Expression*⟩	→	⟨*UnaryExpression*⟩ | ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩
⟨*UnaryExpression*⟩	→	⟨*Literal*⟩ | ⟨*Invocation*⟩
⟨*Invocation*⟩	→	⟨*Function*⟩ `"("` ⟨*Expression*⟩ `")"`
⟨*Function*⟩	→	`"sqrt"` | `"cos"` | `"tan"` | `"sin"`
⟨*Literal*⟩	→	`/[1-9][0-9]*/`

Figure 6.1: A more complex grammar for a calculator. This is the same grammar as in Figure 3.2.

within the program code. Therefore, the advantage of their tool over statistical debugging is just the embedded test generator. It might be interesting to see whether the tests generated by ALHAZEN are equally good at pinpointing the bug location when they are used with a statistical debugging tool. All in all, I think that software developers will be grateful for any piece of information they can get, so combining ALHAZEN with statistical debugging, or the tool by Rößler et al. [61], may be beneficial.

There are approaches which work with concrete models of why a program fails. Johnson, Brun, and Meliou [37] attempt to find tests which refute a hypothesis on what causes a software behavior, but their approach to test generation is purely random. They just hope that by chance a meaningful test is generated. In contrast, ALHAZEN can explore systematically, and create inputs with the required properties directly.

Chen et al. [9] use a decision tree learner to determine which component in a large web application causes a specific failure. While this uses the same model as ALHAZEN, a decision tree, it has a different objective: Instead of predicting what is the cause of the bug, they predict which component is likely to contain the failure.

6.1.4 Abstracting Failure-Inducing Inputs

The approach which is most closely related to ALHAZEN is DDSET[25], which the author of this thesis contributed to. DDSET starts with a parse tree, and replaces subtrees in the parse tree with trees that are rooted in the same control form. Therefore, the trees still fulfill the requirements laid out by the grammar. If the behavior of interest still occurs, DDSET marks this node as *abstract*. If the behavior does not occur any longer, it marks the node as *concrete*. After repeating this process several times, all nodes are marked as concrete or abstract. Within the tree of concrete nodes, abstract nodes can be replaced. Therefore, the tree captures the structure of behavior-triggering inputs.

Let's illustrate this with an example. I will again use a calculator which reads inputs as described by the grammar in Figure 6.1. This calculator fails on the input `"tan(90) + 5 + 7"`. The parse tree for this input can be found in Figure 6.2.

DDSET starts at the root node. It replaces the existing node with a new node which is labeled with the same control form. For the example, assume that it goes with the ⟨*UnaryExpression*⟩ that has `"9"` as a leaf word. Now, the leaf word

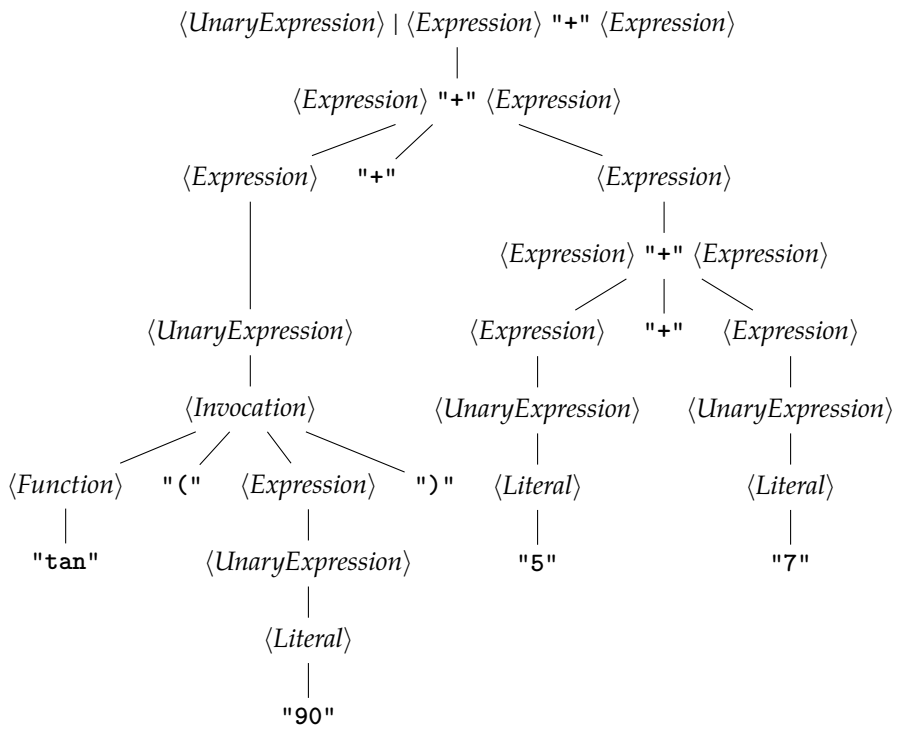


Figure 6.2: The parse tree for `"tan(90) + 5 + 7"` within Figure 6.1. For the sake of readability and size, this figure leaves out some nodes that would be labeled with alternations or concatenations.

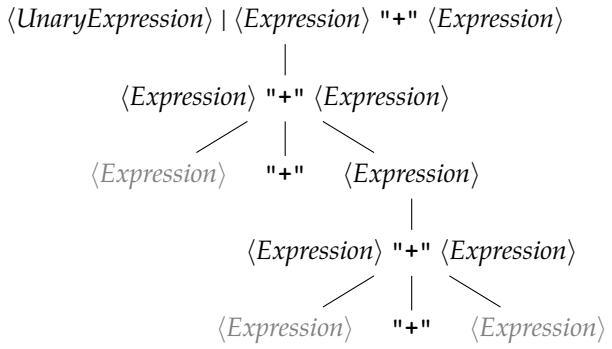


Figure 6.3: The abstract failure inducing input derived by DDSET. Abstract nodes are gray.

of the entire tree is `"9"`[3]. The behavior of interest does not occur with `"9"`, so DDSET marks the root node as concrete. Next, DDSET attempts to replace the children of this node one after another. It starts with the left-most child, and replaces `"tan(90)"` with `"7"`. This is possible because both expressions have a parse tree which is rooted in ⟨*Expression*⟩. The bug still occurs, so DDSET marks the left child as abstract. The algorithm goes on to replace the remaining children, one after another, and observes that it cannot replace ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩, but both children of this node. In the end, it removes the children of all abstract nodes, and obtains the tree in Figure 6.3. While this is an incomplete parse tree, it can also be written as a string: ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩ `"+"` ⟨*Expression*⟩. The authors of DDSET call such a string an *abstract failure-inducing input*. Concrete failure-inducing inputs can be obtained from such an abstract failure-inducing input by filling in the non-terminals with some valid leaf word.

Similar to ALHAZEN, DDSET provides a model which describes a set of behavior-triggering inputs. In this, DDSET and ALHAZEN share their motivation. Both perform program executions to test preliminary versions of their models, and use the observations to refine them. However, their expressiveness is complementary.

Let's assume we would apply ALHAZEN to the example I just presented. ALHAZEN can provide the information that a `"+"` needs to be in a behavior-triggering input, however, it cannot express that there need to be two `"+"`. ALHAZEN fails at structural properties like this. At the same time, consider what happens when DDSET is applied to my initial example for ALHAZEN. That example was a calculator which fails for the input `"sqrt(0)"`. ALHAZEN would provide the information that the calculator fails if within `"sqrt(`x`)"`, it is $x \leq 0$. DDSET cannot interpret `"0"` numerically, it can only try to replace it with arbitrary subtrees. Therefore, DDSET's abstract failure-inducing input would in the end be `"sqrt(0)"`, which is not abstract at all. ALHAZEN and DDSET deal with different properties of the input: DDSET excels at analysing *structural properties*. At the same time, ALHAZEN works best for *semantic properties*. In future work, both approaches should be combined.

6.1.5 Specification Mining

Specification mining attempts to provide a specification for the analysed program, which is similar to ALHAZEN, as a specification can be seen as an explanation for the behavior of the program. This section will look at some of those approaches in more detail.

One of the first approaches in specification mining was DAIKON[17]. DAIKON infers *dynamic invariants*, that is, properties which hold for every observed program run. If, within a set of program runs, the argument x to sqrt(x) is always positive, DAIKON infers the invariant $x \geq 0$. DAIKON's invariants are derived from predefined templates: It starts by instantiating those templates with the variables in the program, and then discards those which are not fulfilled by one of the observations. ALHAZEN also derives a model which describes all observed program runs. While DAIKON is limited by the pre-defined patterns, ALHAZEN is limited by the expressiveness of decision trees. Also, ALHAZEN works at a different level. Instead of function arguments, ALHAZEN reasons about properties

[3] DDSET replaced the root, after all

of the program input. This means that ALHAZEN's model can be used to infer additional program inputs. This is rather difficult with DAIKON invariants, as the relationship of variable values to program inputs is unknown.

Galeotti et al. [21] extend the idea of DAIKON for dynamic loop invariant detection. They assume that there is a postcondition for a function, and are looking for an invariant for a loop within this function. In this situation, they relax DAIKON's reliance on predefined templates for the invariant, by using *syntactic mutations* of the postcondition as invariant candidates. Their work also features a feedback loop: They attempt to generate additional tests, rejecting more of the invariant candidates, by running a test generator. This idea is conceptually similar to ALHAZEN's feedback loop, but does not attempt to provide something human-readable. On the contrary, they explicitly want to use those loop invariants in automated verification.

6.1.6 Unit Analysis of Subprograms

The key idea for the BASILISK generator is to limit symbolic execution to subprograms, and make the input parts which are relevant to ALHAZEN's hypothesis symbolic. The first part, limiting symbolic execution to subprograms, has been explored by other authors before.

When generating random inputs, or even when using a coverage-based feedback-driven mechanism, it is often difficult to solve tests for precise values, such as checksums or magic numbers. Aschermann et al. [5] overcome this problem by stopping program execution at this point in time, and finding symbolic solutions. Just as BASILISK, they use correspondence between input values and values observed in the program run to decide which parts are to be made symbolic. However, in difference to BASILISK, they do not generate unit tests. Instead, they solve symbolic constraints directly. This also means that the symbolic part of their execution rarely solves more than one comparsion. In constrast, BASILISK solves an entire program unit symbolically.

Stephens et al. [68] separates the program in compartments, and uses concolic execution, that is, symbolic execution in parallel to non-symbolic execution, to solve conditions at the compartment boundaries, while traditional fuzzing is used within the compartments. Similar to REDQUEEN, this allows to combine the strength of symbolic analysis and fuzzing, the differences lie mostly in the technical implementation. While REDQUEEN solves individual comparisons, and takes the decision what is symbolic based on input correspondance, DRILLER uses tracing to find out which values need to be considered as symbolic. As mentioned in Section 5.7.1, both approaches avoid the overhead of generating an execution graph, which gives them a better performance than BASILISK.

6.2 Threats to Validity

As all scientific work, the work presented in this thesis has some threats to validity.

The first question is whether results can be generalized to other subjects. Our subjects are written in two programming languages, Java and C, and read different input formats. So my evaluation set is quite diverse. However, ALHAZEN can only diagnose bugs if the diagnosis can be expressed in terms of its features.

The thesis showed this in Section 2.7.1, when I observed that the condition for Rhino 385 and Closure 2808 cannot be expressed, because they are context-sensitive. This is a general limitation of Alhazen. In Section 6.3, I discuss how to extend the set of features Alhazen supports.

Also, the results may depend on the grammar used. In Section 4.1.2, I described how properties of the grammar, in this case whether repetitions are written as quantifications or not, influence the results. Similar properties of the grammar could have an effect as well. All my grammars come from fuzzing campaigns, mostly the one by Havrikov and Zeller [31]. So similarities of the grammars cannot be ruled out.

The second question concerns internal validity. In Section 2.5.3, in tandem with the grammar rewrite in Section 2.4, I described how the shortest derivation close-off can, under some circumstances, lead to more behavior-triggering samples than the probabilistic generator itself would generate. In Section 2.7.3, I describe how the same effect leads to better performance for Alhazen0 in the sets configuration. The bad results for Alhazen0 as a generator, presented in Section 2.7.2, proves that the effect is sizeable. In fact, if Alhazen0's own generator is modified to use a shortest-derivations for tie-breaking (rather than minimal length), the generator precision of Alhazen0 as evaluated in Section 2.7.2, increases by 40%! This is an example for coincidental correlation: Within the probabilistic generator, coincidental correlation means that some failing samples do not fail due to probabilistic generation, but due to the chosen close-off. If Alhazen's generator has the same coincidental effect, and favors the same control forms, those effects add up and lead to unrealistic high accuracy and precision. This is especially true for the sets configuration, which has the same bias in training and verification sets.

Within the original paper on Alhazen[39], only results for the sets configuration where given. The paper lists a possible bias within the training data as a possible threat to validity, however, at the time the authors, which include the author of this thesis, did not realize how severe the problem in fact was. Most notably, Alhazen used a shortest derivation length close-off at this point in time, so even samples generated by Alhazen's own generator had the aforementioned bias. When I realized how severe the bias is, I replaced the close-off within Alhazen with a close-off based on minimal length, as described in Section 3.2.4. The results in Section 3.3 were obtained with the new close-off, and are therefore free of this bias.

This bias is quite visible as soon as one starts to perform a qualitative analysis, as in Section 2.7.3 and Section 3.3.3. I discovered no evidence of further biases when I did so. However, I cannot be sure whether there are more, yet undiscovered, effects of coincidental correlation that lead to better-than-deserved results.

As with all work that develops new tools, programming mistakes in my code may lead to wrong results. I took great care to rule out mistakes. My code includes 216 automated test cases, which I used to check for correctness after every code change. I carefully analysed logfiles for each run, and checked for evidence of unexpected behavior. The code was reused by Havrikov, Kampmann, and Zeller [30], and their analysis helped to find and fix further implementation hickups.

All in all, I took all reasonable measures to ensure that the results are valid and sound.

6.3 Future Work

My work on ALHAZEN opens the door for tools which automatically find the reason behind a program failure. Still, ALHAZEN itself is just a prototype which showcases the idea. Improvements are necessary for a real-world deployment.

The first obvious improvement is the feature selection: Right now, ALHAZEN is incapable of expressing relationships between different input parts. I described that in Section 2.7.3, when I observed that ALHAZEN0 cannot diagnose the RHINO 385 bug, because it requires two variables with the same name. While ALHAZEN can, for example, express that a specific character within a variable name causes a problem, it cannot express that two input parts need to be identical. This limitation needs to be lifted: There are too many bugs which require something like this in their explanation for ALHAZEN to be useful if this restriction persists. This problem can be tackled in two ways: With a stronger machine-learning approach, or with more advanced features.

Some extensions to the features used by ALHAZEN are straight forward: I already have max-char-length(), introducing a feature min-char-length(), which argues about the minimal length of a control form, rather than the maximal length, would be easy.

As discussed in Section 6.1.4, DDSET[25] and ALHAZEN are complementary in the input properties they argue about: DDSET is restricted to structural properties, while ALHAZEN excels at semantic properties. A connection between ALHAZEN and DDSET is therefore a natural extension: One could use patterns as features, use ALHAZEN to learn semantic properties as soon as the pattern has been found, or train both approaches in tandem on the same training data.

However, there are more advanced methods to phrase features. Within the machine-learning community, there is some work on graph embeddings[60]. As parse trees are essentially graphs, those approaches may lend themselves as a better way to learn from parse trees.

The generator presented in this thesis is extendable: If a new feature is introduced, a new r' needs to be defined, and integrated into r_P. Then, the generator can generate samples with specific values for those features. However, the generator has a clear drawback, and that is feature interaction. If one predicate asks for a property for a subtree of the generated parse tree, and another predicate asks for another property in the same subtree, I often see problems: Both predicates are greedy *independently*. They don't know about each other, and each tries to push the inner search in its preferred direction, rather than agreeing on a direction. This means that the outer search needs to come in action, and explore several options for this part of the tree. Adding more feature types, and more predicate types, could intensify this problem, and mean that the generator cannot find samples within reasonable time any longer. Future work should check whether Gopinath, Nemati, and Zeller [24]'s Input Algebras or Steinhoefel and Zeller [67]'s ISLA provide preferable behavior in this type of situations. BASILISK, the approach presented in Chapter 5, should be considered as well.

It should also be investigated whether decision trees are the best machine-learning approach for my use case. Decision trees are easy to understand, but not very powerful. It stands to reason that another approach, for example a SVM[70] or a neural network[26] could be more successful. This is, however, in no way guaranteed: As described in Section 2.6.3, Decision Trees are a natural

fit for the kind of data ALHAZEN has. So maybe another tree-based approach would be the right fit. Clearly, experimentation is required to find out which machine-learning approach performs best in the context of ALHAZEN.

However, replacing the machine-learning approach used in ALHAZEN is not an easy task: Section 3.1 describes how I extract the predicates for the generator from the decision tree. This approach relies on the structure of the tree. Therefore, a machine-learning approach which does not provide such a structure cannot be used within ALHAZEN. A different generator may be able to use the internal representation of another machine-learning approach, or work without access to the internal representation at all. However, this does not apply to any of the work mentioned previously. Luckily, there is quite some work on explainable machine learning. This work aims to provide an explanation for why a model provides the prediction it does provide, even if the model itself doesn't. A tool like SHAP[46] or LIME[72] could be used to derive predicate sets that the generator can then solve.

6.4 Conclusion

In this thesis, I presented ALHAZEN. ALHAZEN is a tool for automated debugging: Based on as little as two failing samples, it generates a hypothesis for why the program fails.

Chapter 1 presented three use cases for ALHAZEN. The tool can predict program behavior, generate additional inputs and help to understand program behavior.

I developed ALHAZEN in two steps: In Chapter 2, I presented ALHAZEN0. This tool can already generate reasonable explanations for program behavior, but requires a huge number of input samples.

In Chapter 3, I extended ALHAZEN0 to ALHAZEN. The newly added component is a sample generator. Together with the hypothesis learner, this generator *generates* the samples that ALHAZEN learns from. This means that ALHAZEN no longer requires such an extensive sample set. It can now learn from just two input samples.

With respect to predicting program behavior, I can conclude that

For reference

> Learning from just two inputs, ALHAZEN achieves a precision of 90.8% and an accuracy of 88.6% over all subjects.
>
> See Section 3.3.1

This is an excellent result: ALHAZEN recognizes a large part of the behavior-triggering samples correctly. Therefore, ALHAZEN could be used to filter inputs before they reach the program under test, and potentially avoid some failures. Be aware, however, that ALHAZEN is not perfect: Such a solution should only be used as a temporary measure. A software developer still needs to provide a real fix for the problem.

For generating additional inputs, I conclude

For reference

89.2% of the samples generated trigger or don't trigger the behavior as requested. 65.2% of the samples generated trigger the behavior, if requested.

See Section 3.3.2

ALHAZEN can, therefore, be used to generate additional input samples. Those can be used as a regression test suite to verify a fix, or as additional samples for some automated program repair technique. Just having more samples available may make debugging a lot easier.

The findings about ALHAZEN's usefulness in debugging are inconclusive: The user studies performed in Chapter 4 have too little participants. They can, however, inform the design of a user study. I proposed such a design in Section 4.3.

Chapter 5 explores whether unit-level program analysis can be sued to generate more information for ALHAZEN. While the combination of system-level and unit-level analysis is shown to be effective, the combination with ALHAZEN gives mixed results: It works in some situations, but not in general. More work is required to find a working combination of BASILISK and ALHAZEN's own generator.

All in all, ALHAZEN shows the huge potential of machine learning within program understanding. Relations between program input and program behavior can be discovered and learned from observed behavior. Future work should, and is already, explore better machine learning approaches, and improve upon the generator used within ALHAZEN.

In ten years from now, no software developer should debug without an artificially intelligent helper.

Appendix A

Bug Classification

Bug reports for ansible can be found at

`https://github.com/ansible/ansible/pull/<bug_no>`,

where ⟨*bug_no*⟩ should be replaced with the number of the bug. For django, bug reports can be found at

`https://github.com/django/django/pull/<bug_no>`.

The following list of bug reports I examined is ordered by the categories in the final classification.

Special Case Missing

Project	Bug No.	One-line Summary
ansible	73947	missing special case in string parsing
ansible	73809	searching wrong subdir, basically missed a special case
ansible	73776	forgot a case where something does not have the expected type
ansible	73765	forgot handling of a special case
django	14182	added an assertion, negative number does not work
django	14179	missing special case, missing host
django	14170	missing special case
django	14148	incorrect handling of a special case (empty string)
django	14144	incorrect handling of a special case (empty string)
ansible	74030	missing condition for different modes of execution

Too Much Code

Project	Bug No.	One-line Summary
ansible	73710	removes a special case
django	14140	removed unnecessary if
django	14152	removed dead code

Error Handling

Project	Bug No.	One-line Summary
ansible	73963	missing error handling
ansible	74022	added missing error handling
ansible	73804	go on after first error, basically exiting a loop to early due to error handling
django	14151	added missing error handling

Type Errors

Project	Bug No.	One-line Summary
django	14196	type error, bytes vs. string
django	14195	type error in combination with error handling
django	14190	wrong access mode when opening a file
django	14149	changed expected type for a parameter

Default Values

Project	Bug No.	One-line Summary
ansible	73961	wrong default value
ansible	73924	also a variant of wrong default value
django	14167	changing tests only, wrong default values
django	14166	adding a default value
django	14169	changed a string constant

Due to Interesting Python Syntax

Project	Bug No.	One-line Summary
ansible	73808	forgot to empty a list, and therefore didn't obey depth limit, dirs[:] = vs. dirs =
ansible	73806	forgot to mutate instead of assign, dirs[:] = vs. dirs =
ansible	74029	wrong indentation

Incorrect Communication with Other Program Parts

Project	Bug No.	One-line Summary
ansible	73881	improper state update after performing a task
django	14139	fixed incorrect setup (missing call)
django	14124	avoid an additional update (method call too much)

Performance

Project	Bug No.	One-line Summary
ansible	73951	performance, try to avoid blocking
django	14198	readability, performance optimization
django	14175	test performance

Output Formatting

Project	Bug No.	One-line Summary
django	14191	re-ordering for better readability of output
django	14174	changing error messages
django	14155	better output formatting

Changes to Comments only

Project	Bug No.	One-line Summary
ansible	73718	identical to #73808
ansible	73714	changes comments only
django	14192	changes comments only
django	14176	comments only

Miscellaneous

Project	Bug No.	One-line Summary
django	14189	adds missing feature
django	14177	changing test discovery
django	14164	incorrect language code formats
django	14163	fixing tests only
django	14153	fixed a key for a cache
django	14205	adds a test
django	14203	refactoring
django	14199	applies a previous fix to an additional location?
ansible	73863	full qualified name vs. simple name of a plugin

Configuration only

The ansible repository contains configuration files which describe properties of various plugins. Some of the merge requests made changes to those configuration files only.

Project	Bug No.	One-line Summary
ansible	73839	fix is in a config file
ansible	73825	fix is in a config file

Documentation Only

Ansible labels merge requests which change only the documentation as such. Therefore, I could exclude those merge requests from my analysis. For django, those merge requests have no special label, and therefore I could not exclude them automatically.

Project	Bug No.	One-line Summary
django	14194	documentation only
django	14183	documentation only
django	14181	documentation only
django	14172	documentation only
django	14171	documentation only
django	14160	documentation only
django	14143	documentation only
django	14133	documentation only
django	14129	documentation only

Appendix B

Coverage over Time for Basilisk

Coverage on all Basilisk subjects over time.

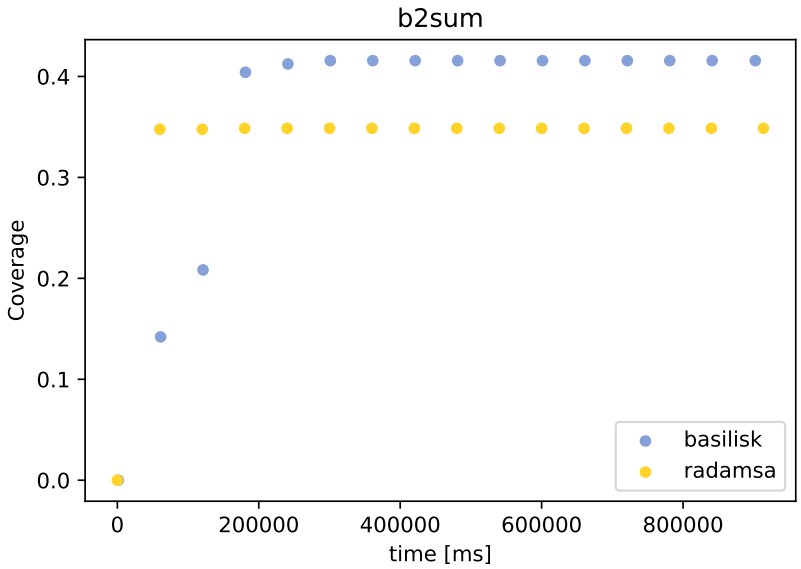


Figure B.1: Coverage on `b2sum`

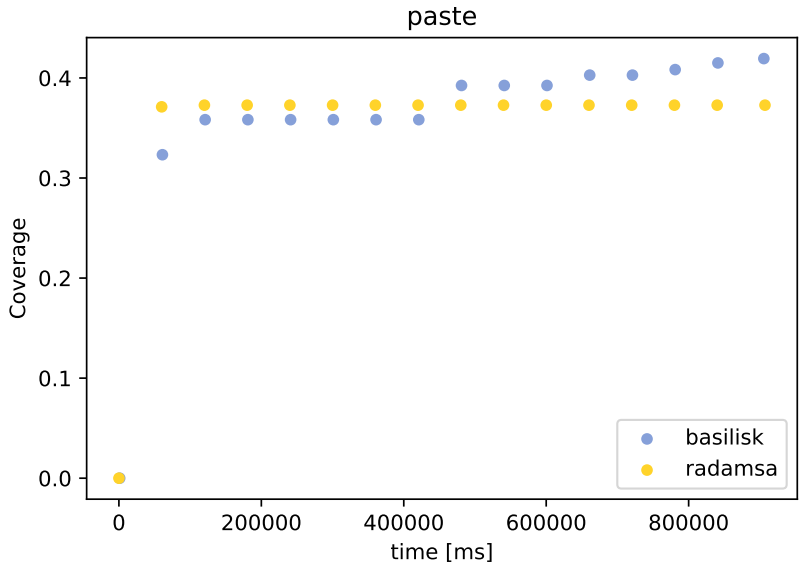


Figure B.2: Coverage on `paste`

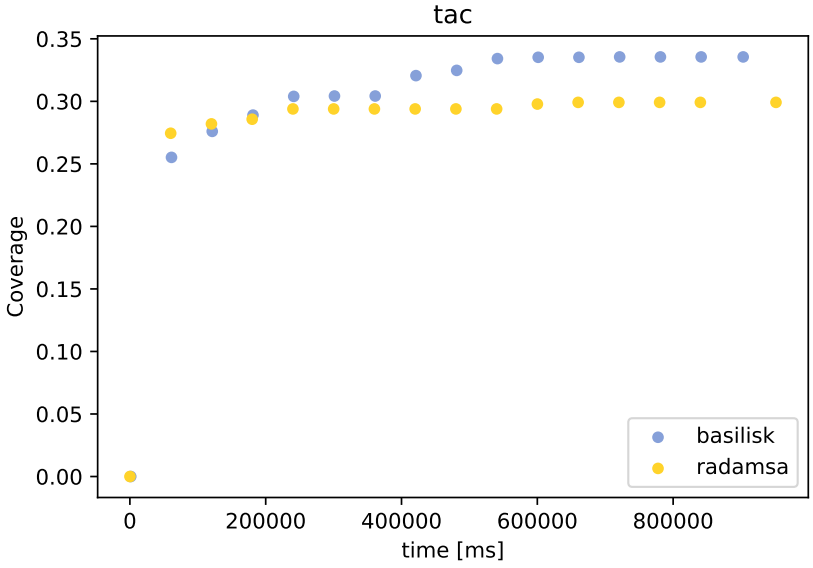


Figure B.3: Coverage on `tac`

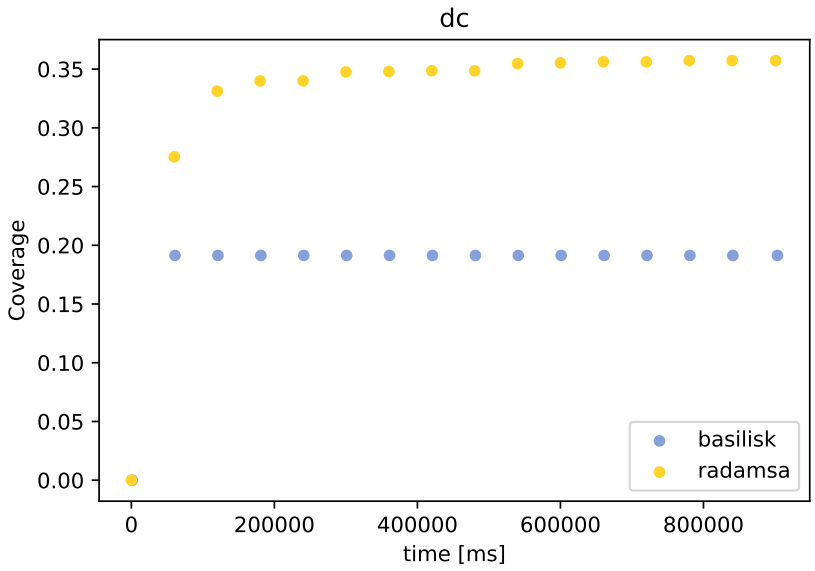


Figure B.4: Coverage on `dc`

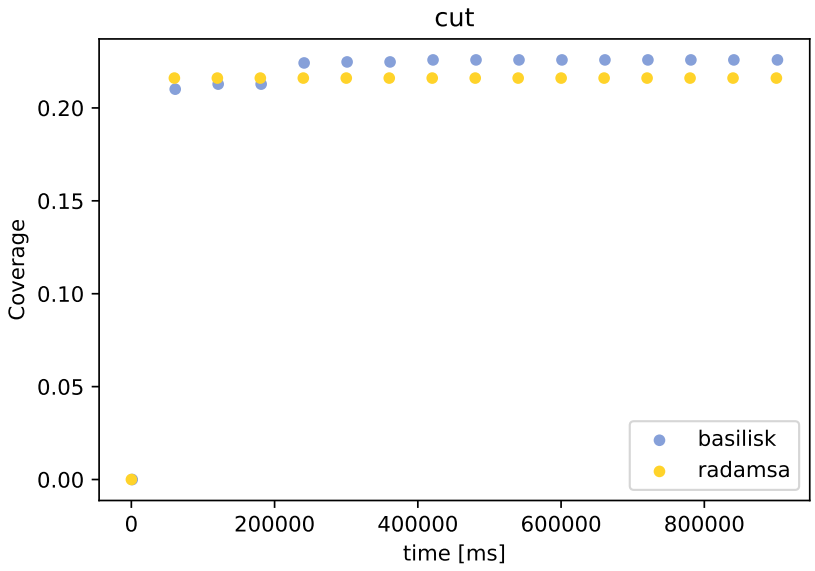


Figure B.5: Coverage on `cut`

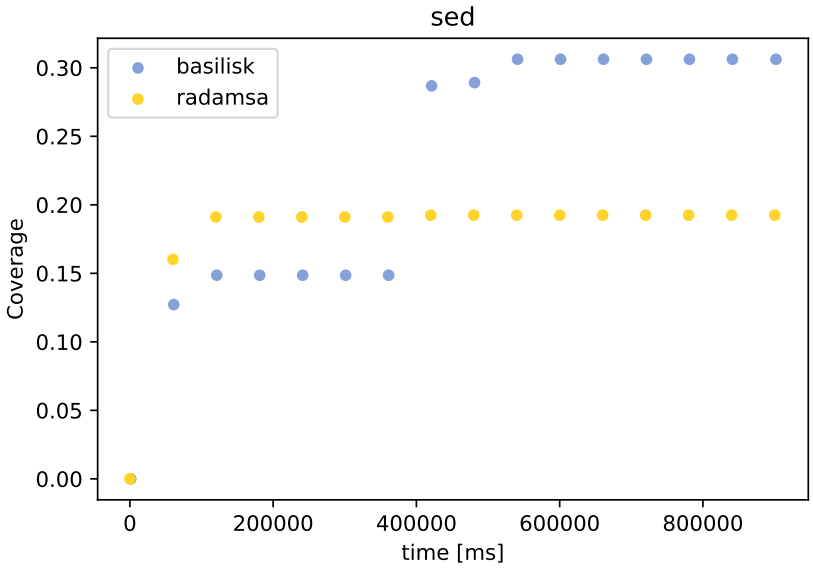


Figure B.6: Coverage on `sed`

Bibliography

[1] Twitter @vm_call. *Carl Schou on Twitter: After joining ...* June 22, 2021. URL: https://twitter.com/vm_call/status/1405937492642123782.

[2] M AdelsonVelskii and Evgenii Mikhailovich Landis. *An algorithm for the organization of information*. Tech. rep. JOINT PUBLICATIONS RESEARCH SERVICE WASHINGTON DC, 1963.

[3] Akasurde. *Amazon: Fix distribution facts for older release by Akasurde · Pull Request #73947 · ansible/ansible · GitHub*. Nov. 28, 2021. URL: https://github.com/ansible/ansible/pull/73947.

[4] Red Hat Ansible. *Ansible is Simple IT Automation*. Nov. 28, 2021. URL: https://www.ansible.com/.

[5] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[6] Osbert Bastani et al. "Synthesizing program input grammars." In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 95–110.

[7] Marcel Böhme et al. "How Developers Debug Software—The DBGBENCH Dataset." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 244–246.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[9] Mike Chen et al. "Failure diagnosis using decision trees." In: *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE. 2004, pp. 36–43.

[10] cjerdonek. *Fixed #32578 – Fixed crash in CsrfViewMiddleware when a request with Origin header has an invalid host. by cjerdonek · Pull Request #14179 · django/django · GitHub*. Nov. 28, 2021. URL: https://github.com/django/django/pull/14179.

[11] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. 1977. ISBN: 978-0-12-179060-8.

[12] NVIDIA Corporation. *CUDA LLVM Compiler | NVIDIA Developer*. Nov. 29, 2021. URL: https://developer.nvidia.com/cuda-llvm-compiler.

[13] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[14] Fred D Davis, Richard P Bagozzi, and Paul R Warshaw. "User acceptance of computer technology: A comparison of two theoretical models." In: *Management science* 35.8 (1989), pp. 982–1003.

[15] Jay Earley. "An efficient context-free parsing algorithm." In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[16] Sebastian Elbaum et al. "Carving and Replaying Differential Unit Test Cases from System Test Cases." In: *IEEE Transactions on Software Engineering* 35.1 (2009), pp. 29–45.

[17] Michael D Ernst et al. "The Daikon system for dynamic detection of likely invariants." In: *Science of computer programming* 69.1-3 (2007), pp. 35–45.

[18] Sven Christian Fackert. "A User Study on the Effectiveness of Test-Based Debugging Diagnoses." MA thesis. Saarland University, 2021.

[19] George Fink and Matt Bishop. "Property-based testing: a new approach to testing for assurance." In: *ACM SIGSOFT Software Engineering Notes* 22.4 (1997), pp. 74–80.

[20] Django Software Foundation. *The web framework for perfectionists with deadlines | Django*. Nov. 28, 2021. URL: `https://www.djangoproject.com/`.

[21] Juan P Galeotti et al. "Automating full functional verification of programs with loops." In: *CoRR, abs/1407.5286* (2014).

[22] Erich Gamma et al. *Elements of reusable object-oriented software*. Vol. 99. Addison-Wesley Reading, Massachusetts, 1995.

[23] Rahul Gopinath, Björn Mathis, and Andreas Zeller. "Mining Input Grammars from Dynamic Control Flow." In: *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*. 2020.

[24] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. "Input Algebras." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 699–710.

[25] Rahul Gopinath et al. "Abstracting failure-inducing inputs." In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 237–248.

[26] Kevin Gurney. *An introduction to neural networks*. CRC press, 2018.

[27] Mark Harman. "Automated patching techniques: the fix is in: technical perspective." In: *Communications of the ACM* 53.5 (2010), pp. 108–108.

[28] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[29] Nikolas Havrikov. *NullPointerException · Issue #386 · mozilla/rhino · GitHub*. July 28, 2021. URL: `https://github.com/mozilla/rhino/issues/386`.

[30] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. "From Input Coverage to Code Coverage: Systematically Covering Input Structure with k-Paths." In: *ACM Transactions on Software Engineering and Methodology* (Feb. 2022). URL: `https://publications.cispa.saarland/3572/`.

[31] Nikolas Havrikov and Andreas Zeller. "Systematically covering input structure." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 189–199.

[32] Aki Helin. *Radamsa*. https://gitlab.com/akihe/radamsa.

[33] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: a grammar-based open source fuzzer." In: *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 2018, pp. 45–48.

[34] Jörg Hoffmann and Bernhard Nebel. "The FF planning system: Fast plan generation through heuristic search." In: *Journal of Artificial Intelligence Research* 14 (2001), pp. 253–302.

[35] Matthias Höschele and Andreas Zeller. "Mining Input Grammars from Dynamic Taints." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, 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.

[36] ECMA International. *ECMAScript; 2022 Language Specification*. Nov. 20, 2021. URL: https://raw.githubusercontent.com/tc39/ecma262/master/spec.html.

[37] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. "Causal Testing: Understanding Defects' Root Causes." In: *Proceedings of the 2020 International Conference on Software Engineering*. 2020.

[38] Project Jupyter. *Project Jupyter*. Nov. 28, 2021. URL: https://jupyter.org/.

[39] Alexander Kampmann et al. "When does my Program do this? Learning Circumstances of Software Behavior." In: *ESEC/FSE 2020*. June 2020. URL: https://publications.cispa.saarland/3107/.

[40] Dongsun Kim et al. "Automatic patch generation learned from human-written patches." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 802–811.

[41] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization." *See* http://llvm.cs.uiuc.edu. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[42] Claire Le Goues et al. "Genprog: A generic method for automatic software repair." In: *Ieee transactions on software engineering* 38.1 (2011), pp. 54–72.

[43] *libFuzzer: a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. LLVM Compiler Infrastructure.

[44] Rensis Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932).

[45] Kenneth Lorber. *NetHack 3.6.6: NetHack Home Page*. Dec. 23, 2021. URL: https://www.nethack.org/.

[46] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions." In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[47] Björn Mathis et al. "Parser-directed fuzzing." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 548–560.

[48] Phil McMinn. "Search-based software test data generation: a survey." In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156.

[49] Barton P. Miller, Louis 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: http://doi.acm.org/10.1145/96267.96279.

[50] Ghassan Misherghi and Zhendong Su. "HDD: hierarchical delta debugging." In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 142–151.

[51] Hoang Duong Thien Nguyen et al. "Semfix: Program repair via semantic analysis." In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 772–781.

[52] Carlos Pacheco and Michael D Ernst. "Randoop: Feedback-directed Random Testing for Java." In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. Vol. 2. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297902. URL: http://doi.acm.org/10.1145/1297846.1297902.

[53] Chris Parnin and Alessandro Orso. "Are automated debugging techniques actually helping programmers?" In: *Proceedings of the 2011 international symposium on software testing and analysis*. 2011, pp. 199–209.

[54] Terence Parr et al. *GitHub - antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions.* Nov. 20, 2021. URL: https://github.com/antlr/grammars-v4.

[55] Esteban Pavese et al. "Inputs from Hell: Generating Uncommon Inputs from Common Samples." In: *arXiv preprint arXiv:1812.07525* (2018).

[56] Karl Pearson. "VII. Note on regression and inheritance in the case of two parents." In: *proceedings of the royal society of London* 58.347-352 (1895), pp. 240–242.

[57] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[58] Yuhua Qi et al. "The strength of random search on automated program repair." In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 254–265.

[59] Baishakhi Ray et al. "On the" naturalness" of buggy code." In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 428–439.

[60] Arnold L Rosenberg. "Issues in the study of graph embeddings." In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 1980, pp. 150–176.

[61] Jeremias Rößler et al. "Isolating failure causes through test case generation." In: *Proceedings of the 2012 international symposium on software testing and analysis*. 2012, pp. 309–319.

[62] Harold Sackman, Warren J Erikson, and E Eugene Grant. "Exploratory experimental studies comparing online and offline programming performance." In: *Communications of the ACM* 11.1 (1968), pp. 3–11.

[63] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge, 2008.

[64] Skipper Seabold and Josef Perktold. "statsmodels: Econometric and statistical modeling with python." In: *9th Python in Science Conference*. 2010.

[65] Higor A de Souza, Marcos L Chaim, and Fabio Kon. "Spectrum-based software fault localization: A survey of techniques, advances, and challenges." In: *arXiv preprint arXiv:1607.04347* (2016).

[66] Charles Spearman. "The proof and measurement of association between two things." In: (1961).

[67] Dominic Steinhoefel and Andreas Zeller. "Specifying Input Constraints."

[68] Nick Stephens et al. "Driller: Augmenting fuzzing through selective symbolic execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[69] Chengnian Sun et al. "Perses: Syntax-guided Program Reduction." In: ICSE '18. Gothenburg, Sweden: ACM, 2018, pp. 361–371. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180236. URL: http://doi.acm.org/10.1145/3180155.3180236.

[70] Shan Suthaharan. "Support vector machine." In: *Machine learning models and algorithms for big data classification*. Springer, 2016, pp. 207–235.

[71] Philip H Swain and Hans Hauska. "The decision tree classifier: Design and potential." In: *IEEE Transactions on Geoscience Electronics* 15.3 (1977), pp. 142–147.

[72] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "" Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: *arXiv e-prints* (2016), arXiv–1602.

[73] Westley Weimer et al. "Automatically finding patches using genetic programming." In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 364–374.

[74] Jooyong Yi et al. "A correlation study between automated program repair and test-suite metrics." In: *Empirical Software Engineering* 23.5 (2018), pp. 2948–2979.

[75] Michał Zalewski. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/.

[76] Andreas Zeller. *The Debugging Book*. Retrieved 2021-10-13 13:24:19+02:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.debuggingbook.org/ (visited on 10/13/2021).

[77] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[78] Andreas Zeller and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.