When Does My Program Do This? Learning Circumstances of Software Behavior

Alexander Kampmann
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
alexander.kampmann@cispa.saarland

Nikolas Havrikov
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nikolas.havrikov@cispa.saarland

Ezekiel O. Soremekun
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
ezekiel.soremekun@cispa.saarland

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

A program fails. Under which circumstances does the failure occur? Our Alhazen approach starts with a run that exhibits a particular behavior and automatically determines input features associated with the behavior in question: (1) We use a *grammar* to parse the input into individual elements. (2) We use a decision tree learner to *observe* and *learn* which input elements are associated with the behavior in question. (3) We use the grammar to *generate additional inputs* to further strengthen or refute hypotheses as learned associations. (4) By repeating steps 2 and 3, we obtain a *theory* that explains and predicts the given behavior. In our evaluation using inputs for `find`, `grep`, `NetHack`, and a JavaScript transpiler, the theories produced by Alhazen *predict* and *produce* failures with high accuracy and allow developers to *focus* on a small set of input features: "`grep` fails whenever the `--fixed-strings` option is used in conjunction with an empty search string."

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Grammars and context-free languages*; *Oracles and decision trees*; Active learning.

KEYWORDS

debugging, error diagnosis, machine learning, software behavior

ACM Reference Format:
Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. When Does My Program Do This? Learning Circumstances of Software Behavior. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409687

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409687

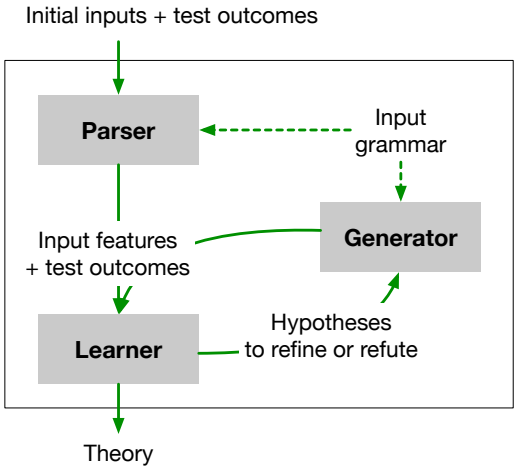

Figure 1: How Alhazen works. Given a set of initial inputs and their test outcomes which determine whether the behavior in question is present or not, we *parse* the input into its elements using a given input grammar, A *learner* then determines the associations of input properties and outcomes, producing hypotheses on the circumstances under which the behavior occurs. By producing inputs from the grammar, we *generate* additional tests to further refine or refute hyoptheses, eventually obtaining a *theory* that explains and predicts when the behavior in question occurs.

1 INTRODUCTION

When diagnosing why a program fails, one of the first steps is to precisely understand the *circumstances* of the failure—that is, when the failure occurs and when it does not. Such circumstances are necessary for three reasons. First, knowing the circumstances is necessary to precisely *predict* when the failure takes place; this is important to devise the severity of the failure. Second, one needs them to design a *precise fix:* A fix that addresses only a subset of circumstances is incomplete, while a fix that addresses a superset may alter behavior in non-failing scenarios. Third, one can use them to create *test cases* that reproduce the failure and eventually validate the fix.

In this paper, we introduce ALHAZEN—an approach that *automatically determines the circumstances under which some program behavior of interest takes place.*[1] As all program behavior is determined by its inputs, we see failure circumstances as properties of the program input; our aim is thus to determine input features that would be associated with the behavior in question.

As an example of how ALHAZEN works and what it produces, assume some program P to evaluate mathematical functions; the input `sqrt(4)`, for instance, produces the output 2. Given the input `sqrt(-900)`, however, P hangs. At this point, the astute reader already may have an idea on the circumstances of the failure; but we want to determine these *automatically*. To do so, ALHAZEN makes use of three key ingredients, illustrated in Figure 1:

Parsing. We use a *grammar* to *parse* program inputs into individual elements. This allows us to express fine-grained relationships between input elements (and their features) and program behavior (i.e. presence or absence of a failure).
Figure 2 lists the input grammar for P. This grammar will allow us to express failure circumstances by means of the ⟨*function*⟩ being used and the ⟨*number*⟩ being passed.

Learning. We use a *decision tree* to learn which features of input elements are associated with the program behavior in question. By default, the features used in ALHAZEN test whether a particular element occurs in the input or not; in our failure-inducing input, `sqrt` is present, whereas `sin` is not. If some element has a numerical interpretation (such as ⟨*number*⟩), it also uses its maximum value as feature.
The decision tree learner produces a tree that explains and predicts when the behavior in question occurs based on a subset of the input features. Figure 3 shows the initial decision tree learned from the passing input `sqrt(4)` and the failing input `sqrt(-900)`. The initial hypothesis is that the failure occurs when the largest [2] ⟨*number*⟩ is less than or equal to -445.5—a predicate chosen by the decision tree learner as a feature that correctly distinguishes all observations so far.

[1] Ḥasan Ibn al-Haytham (Latinized as *Alhazen;* ~965–~1040) was an Arab researcher of the Islamic Golden Age. His *Kitāb al-Manāẓir* "Book of Optics"(1011–1021) was one of the first embodiments of the modern scientific method, proving hypotheses through reproducible experiments that vary the experimental conditions in a systematic manner [34].
[2] In the example, there cannot be more than one number, but ALHAZEN would be able to handle it if there were.

```
⟨start⟩ → ⟨function⟩ "(" ⟨number⟩ ")";
⟨function⟩ → "sqrt" | "sin" | "cos" | "tan";
⟨number⟩ → "-"? /[1-9][0-9]*/ ("." /[0-9]+/)?;
```

Figure 2: A grammar for evaluating functions.

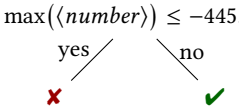

Figure 3: ALHAZEN's initial hypothesis in the sqrt example.

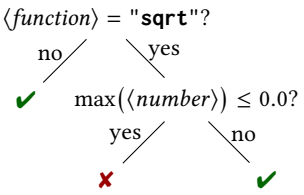

Figure 4: Final decision tree after iteration 29.

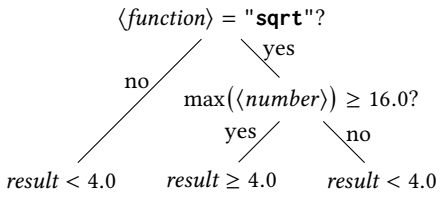

Figure 5: Circumstances for the result being 4.0 or more.

Generating. To precisely capture the failure circumstances, we need further experiments. To this end, ALHAZEN uses the grammar as a *producer* of inputs and systematically explore *alternatives* to the inputs observed so far. For each decision branch in the tree, ALHAZEN generates further inputs to refine or refute the association with the predicted outcome.
In our example, ALHAZEN would generate more inputs for each branch in Figure 3. These satisfy the given conditions from the tree, but otherwise are randomly chosen from the grammar—say, `cos(-444.5)` for the left branch and `cos(-446.5)` for the right branch. Since both pass, the original decision tree is inadequate. Instead, ALHAZEN refines the failure hypothesis such that ⟨*number*⟩ must be less than -673.25. Note that this hypothesis is consistent with all observations so far.
As ALHAZEN generates further inputs for all branches, it eventually learns that the failure depends on `sqrt()` being called. After 29 iterations, ALHAZEN delivers Figure 4, which correctly describes the failure conditions: The ⟨*function*⟩ `"sqrt"` is used, and the ⟨*number*⟩ is less than or equal to 0.

Beyond just pass and fail predicates, ALHAZEN can be applied to obtain explanations and predictions for *arbitrary predicates* over the program execution. For instance, one can use it to determine the circumstances under which a specific output is produced; Figure 5 shows the circumstances for the output being 4 or more. (Note that the trigonometric functions return values in the range $[-1, 1]$.)

Since it requires no program analysis, ALHAZEN scales to arbitrary large programs. *NetHack* is an adventure games, consisting of 240424 lines of code. In January 2020, it was found that NetHack was vulnerable to a buffer overflow [11]. Using a .ini grammar to parse its configuration file, ALHAZEN easily determines that the failure occurs as soon as some line in the configuration file has more than 619 characters (Figure 6).

ALHAZEN can be seen as a full automation of the scientific method, creating, refining and refuting hypotheses from observations over specifically constructed experiments to eventually produce a *theory* of when the program exhibits a specific behavior.

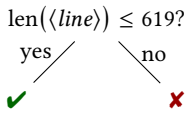

Figure 6: Decision tree for a NetHack failure.

The grammar serves as parser and producer of inputs; the decision tree captures the circumstances that distinguish program behavior.

The structure of this paper follows its three main contributions:

Input elements as features. (Section 2) Using a grammar to parse inputs into fine-grained elements, we can associate the presence or absence of such elements with observed program behavior. This makes these elements *features* for machine learners that can thus infer precise models of program behavior from observed runs. To the best of our knowledge, ours is the first approach to combine general-purpose parsing and machine learning in software engineering.

Creating hypotheses for program behavior. (Section 3) Using a decision tree learner, we can extract associations between input features and program behavior. Decision tree learners are not very precise, but they provide very good explanations to humans—in our case, predicates over input features that capture the circumstances of the behavior. To the best of our knowledge, this is the first use of machine learners over general-purpose input features for predicting, fixing, and producing failures.

Refining and refuting hypotheses. (Section 4) Using the grammar, we can produce additional test cases to refine or refute hypotheses as produced from the learner; we thus combine the explainability of decision trees with the production power of grammars. As the grammar allows us to systematically test alternatives, this active learning approach makes the resulting diagnosis much more precise. To the best of our knowledge, the production of additional inputs to satisfy and refine decision tree constraints is novel, making ours the first automated debugging approach producing a theory over syntactic features.

In Section 5, we evaluate the models generated by Alhazen for their accuracy. Applied on a variety of real-world bugs in standard programs, including `grep` and `find`, we find that the resulting models precisely capture failure circumstances. Applied on JavaScript and its processors, Alhazen is able to isolate nontrivial conditions over elements that lead to failure. After discussing related work (Section 6), Section 7 closes the paper with conclusion and future work, as well as links to code and data.

2 INPUT ELEMENTS AS FEATURES

Alhazen associates properties of the input with program behavior. Those properties are derived from a context-free grammar of the input language. We use presence and absence of non-terminal symbols in the grammar, the length of individual nodes in the path tree, the code point of characters in nodes and the numeric interpretation of parse tree nodes as features. The following section describes the extraction of these features from an input.

⟨*start*⟩ → ⟨*empty*⟩ | ⟨*start*⟩ ⟨*suffix*⟩;
⟨*suffix*⟩ → `"a"`;
⟨*empty*⟩ → `""`;

Figure 7: A grammar with a loop.

2.1 Context-Free Grammars

A context-free grammar consists of a start symbol and a set of production rules. A production rule $\langle P \rangle \rightarrow \alpha$ consists of a *non-terminal symbol* $\langle P \rangle$ on the left and a control form α on the right. A control form can be one of the following:

Terminal symbol. A quoted string.
Non-terminal symbol. A symbol name in angle brackets.
Concatenation. A sequence of control forms.
Quantification. A control form, called the subject, annotated with one of `+`, `*` or `?`.
Alternation. A sequence of control forms, separated by `|`.

When writing grammars, we use regular expressions delimited with slashes as control forms for better readability (e.g. in the ⟨*number*⟩ production of Figure 2). This is possible because any regular expression can be transformed into an equivalent context-free grammar. We use parentheses to avoid confusion about precedence.

If a production rule has an alternation as its right-hand side, we call the control forms within this alternation the alternatives of the non-terminal. In a production rule $\langle P \rangle \rightarrow \alpha|(\beta(\gamma|\delta))$, $\langle P \rangle$ has the alternatives α and $\beta(\gamma|\delta)$. Note that γ and β are not considered alternatives of $\langle P \rangle$ on their own.

A *producer* generates a parse tree from a grammar. A simple base algorithm for a producer is to generate the nodes of the tree in pre-order. Contrary to most standard text books, we have a node for each control form, so a derivation for ⟨*function*⟩ → `"sqrt"` | `"cos"` has three nodes in total, one for ⟨*function*⟩, one for the alternation and one for the chosen alternative.

We call the sequence of control forms in the order the nodes were produced the *derivation sequence*. One possible derivation sequence for the parse tree of `sqrt(-900)` is (1) ⟨*start*⟩ (2) Concatenation of ⟨*function*⟩ `"("` ⟨*number*⟩ `")"` (3) ⟨*function*⟩ (4) alternation of `"sqrt"` | `"tan"` | … (5) `"sqrt"` (6) `"("` (7) ⟨*number*⟩ (8) `"-900"` (9) `")"`. This is a pre-order traversal of the parse tree.

There is one catch to look out for when implementing this algorithm. Assume we want to generate a parse tree for the grammar in Figure 7, and we want the leaf word to contain `"a"`. Within a node for ⟨*start*⟩, we need to decide which alternative we want. We choose the second, as this allows us to generate `"a"`. In a pre-order traversal, we need to generate another node for ⟨*start*⟩ now. As we did not yet generate `"a"`, it is quite easy to take the same decision again, and run into an endless loop. We therefore allow the algorithm to create the child nodes for a concatenation in any order, and add them to the parent node in the required order.

However, this means that there are several derivation sequences for the same leaf word. As an example, (1) ⟨*start*⟩ (2) Concatenation of ⟨*function*⟩ `"("` ⟨*number*⟩ `")"` (3) `"("` (4) `")"` (5) ⟨*function*⟩ (6) alternation of `"sqrt"` | `"tan"` | … (7) `"sqrt"` (8) ⟨*number*⟩ (9) `"-900"` is a possible derivation sequence for `sqrt(-900)` just as well.

In some cases, there can be different parse trees for the same word. In this case, we call the grammar *ambiguous*. Ambiguities in

Before the rewrite

⟨*start*⟩ → ⟨*function*⟩ "**(**" ⟨*number*⟩ "**)**";
⟨*function*⟩ → "**sqrt**" | "**tan**" | "**sin**" | "**cos**";
⟨*number*⟩ → "**-**"? /[1-9][0-9]*/ ("." /[0-9]+/)?;

After the rewrite

⟨*start*⟩ → ⟨*function*⟩ "**(**" ⟨*number*⟩ "**)**" | "**sqrt(-900)**";
⟨*function*⟩ → "**sqrt**" | "**tan**" | "**sin**" | "**cos**";
⟨*number*⟩ → "**-**"? /[1-9][0-9]*/ ("." /[0-9]+/)? | "**-900**";

Figure 8: The last alternatives for *start* and *number* are added by the rewrite step.

grammars usually stem from the fact that disambiguation relies on properties not reflected in a context-free grammar, a poor quality of the formalization of the input language, or a mixture of both.

Throughout this paper, we need a notion of whether a control form ⟨*Q*⟩ is *reachable* from a control form ⟨*P*⟩. The *distance* from a control form ⟨*P*⟩ to a control form ⟨*Q*⟩ is the minimal number of operations required to create a node labeled ⟨*Q*⟩, after the creation of a node labeled ⟨*P*⟩, in the subtree of ⟨*P*⟩. If there can be a node labeled with ⟨*Q*⟩ in the subtree of a node labeled with ⟨*P*⟩, we call ⟨*Q*⟩ *reachable* from ⟨*P*⟩. Otherwise, ⟨*Q*⟩ is *not reachable* from ⟨*P*⟩, and the distance from ⟨*P*⟩ to ⟨*Q*⟩ is infinite.

2.2 Grammar Transformation

The behaviors we want to explain are triggered by complex input structures. While all inputs are words of the grammar, the grammar is often too fine-grained to capture the essence of what causes a bug. Therefore, we perform a *rewrite step* which adds additional alternatives that capture more complex structures. To this end, for all non-terminal symbols in the grammar, we determine the word derived by this symbol in the bug-triggering input, and add those words as alternatives to the symbol. Figure 8 shows the rewritten grammar for the calculator example.

In the rewritten grammar, "**-900**" is added as alternative to ⟨*number*⟩. Also, the full string is added as an alternative to the start symbol. We do not add "**sqrt**" as an alternative to ⟨*function*⟩, because it is already there. Note that the rewrite step makes all our grammars more ambiguous as they always have at least two parse trees for the input we started with.

3 CREATING HYPOTHESES FOR PROGRAM BEHAVIOR

We use a decision tree learner [32] to learn associations between program behavior and input features. In each iteration, Alhazen trains a learner on all known input samples, and uses the obtained tree to generate more inputs, which help to refine the tree in the next iteration.

Decision tree learners express associations in terms of predicates over numeric *features*, i.e. max(⟨*number*⟩) ≤ 0.0. As we want to reason about program inputs, we need to extract numeric features from program inputs. We do so by *parsing* each input, and extracting features from the parse tree. For each production rule and each alternative, we consider the following features:

Existence. This feature has a value of 1 iff the production rule was used in the derivation sequence for an input at least once. We write the existence feature for the production ⟨*start*⟩ as `exists(`⟨*start*⟩`)`. For alternatives, we have an existence feature for the non-terminal (e.g. `exists(`⟨*function*⟩`)`) and individual existence features for each alternative (e.g. `exists(`⟨*function*⟩ `== "sqrt")`).

Length. If for a production ⟨*P*⟩, ⟨*P*⟩ itself is reachable from ⟨*P*⟩ or a quantification is reachable from ⟨*P*⟩, we use the number of characters in the word derived by ⟨*P*⟩ as a feature. For the production ⟨*number*⟩, we write this feature as `len(`⟨*number*⟩`)`. If the right-hand side of the production rule for ⟨*P*⟩ is a quantification, we instead introduce a feature `qu-len(`⟨*P*⟩`)`, which gives the number of child nodes of this quantification. If ⟨*P*⟩ is used multiple times in the derivation, we use the maximum value for both `len` and `qu-len`.

Maximal Code Point. For all productions ⟨*P*⟩ that have more than one derivation, we introduce a feature `max-char(`⟨*P*⟩`)` for the *maximal code point*—that is, the maximal integer representation for all characters in the word derived by ⟨*P*⟩. If there are multiple words derived by ⟨*P*⟩, we use the maximum code point across all words.

Numeric Interpretation. If a production ⟨*P*⟩ only derives words composed of the characters `0-9`, `.` and `-`, we introduce a feature `max(`⟨*P*⟩`)`, which interprets the word as a floating-point number. Again, we use the maximum value for multiple production uses.

All those features are derived from the parse tree of an input. Due to the ambiguity in our grammars, we need to consider all possible parse trees. Therefore, we use an Earley Parser [15], which gives us all possible parse trees, rather than just one.

Table 1 shows the feature values for `sqrt(-900)`. The ⟨*start*⟩ rule is used, and so is our newly-introduced alternative, so `exists(`⟨*start*⟩`)` and `exists(`⟨*start*⟩ `== "sqrt(-900)")` both have a value of 1. The length of this word is 11 characters, and the maximal code point is 116 (which corresponds to 't'). If we had just one parse tree, that would have been all. However, we can also see the alternative parse tree, which uses the pre-existing rule for ⟨*start*⟩. In this parse tree, we have `exists(`⟨*function*⟩`)` and `exists(`⟨*function*⟩ `== "sqrt")` as 1, but ⟨*function*⟩ `== "cos"` as 0. We can again see maximum code point features for ⟨*function*⟩ and ⟨*number*⟩, as well as the numeric interpretation for ⟨*number*⟩, which is -900.

4 GENERATING TESTS TO REFINE HYPOTHESES

As shown in Figure 1, Alhazen uses a *feedback loop* to systematically refine or refute hypotheses. To this end, we *generate tests* that explore the various paths from the decision tree.

4.1 Extracting Prediction Paths

In a decision tree, each internal node contains a predicate $f \leq v$, where f is a feature. Leaves are labeled with the program behavior.

When a decision tree learner classifies a sample s, it traverses its internal structure in the following way: Starting at the root node, the predicate in the node is checked against the (features of) the sample. If it is fulfilled, the "yes" branch is examined next, otherwise the traversal continues at the "no" branch. As soon as the traversal

Table 1: All feature values for *sqrt(-900)* and the transformed subgrammar in Figure 8

Feature	Value
max-char(⟨*start*⟩)	116
len(⟨*start*⟩)	11
⟨*function*⟩ == "**sqrt**"	1
⟨*function*⟩ == "**cos**"	0
⟨*function*⟩ == "**sin**"	0
exists(⟨*start*⟩)	1
exists(⟨*start*⟩) == "**sqrt(-900)**")	1
⟨*function*⟩ == "**tan**"	0
max-char(⟨*function*⟩)	116
⟨*number*⟩	1
⟨*number*⟩ == "**-900**"	1
max-char(⟨*number*⟩)	57
max(⟨*number*⟩)	-900
len(⟨*number*⟩)	4

reaches a leaf, the label of this leaf is the prediction. That is, each prediction traverses a path from the root to a child node of the tree—the *prediction path* for this sample.

Each path in the tree, from root to leaf, can be written as a sequence of predicates of the form $f_i \leq v$ or $f_i > v$.

To generate test inputs, for all paths in the tree, we take all subsets of predicates on the path and *negate* them. For instance, for a path with the predicates $f_1 \leq v_1$ and $f_2 > v_2$ we would generate (1) $f_1 \leq v_1 \wedge f_2 > v_2$, (2) $f_1 > v_1 \wedge f_2 > v_2$, (3) $f_1 \leq v_1 \wedge f_2 \leq v_2$, and (4) $f_1 > v_1 \wedge f_2 > v_2$.

Let us now generate samples which fulfill these sets of predicates. We then proceed in three steps:

(1) We *slice* the grammar into a subset that does not contain productions prohibited by the tree predicates (Section 4.2).
(2) We eliminate predicate sets that are *infeasible* within the grammar (Section 4.3).
(3) We produce solutions for feasible predicates (Section 4.4), which we repeat until the best possible candidate is found within a time budget (Section 4.5).

4.2 Slicing the Grammar

We start by generating a *subset* of the grammar without productions that would be prohibited by the existence predicates—that is, it excludes all productions or alternatives where the predicate states that the existence feature is < 1. As an example, if the predicate exists(⟨*number*⟩ == "**-900**") ≤ 0.5 is in the predicate set, we would rewrite the production rule for ⟨*number*⟩ as ⟨*number*⟩ → "**-**"? /[1-9][0-9]*/ ("**.**" /[0-9]+/)?.

Due to ambiguity, a production may implicitly use a different production in another derivation sequence for the same word. In the transformed grammar for our example (Figure 8), using ⟨*start*⟩ == "**sqrt(-900)**" means that ⟨*number*⟩ == "**-900**" is used implicitly, via a different parse tree for the same word. For all productions and alternatives which derive the same word in all parse trees (that is, the right-hand side contains only terminal symbols or nonterminal symbols with just one production that recursively always derives the same word), we precompute the set of productions that are used implicitly. We also remove a production if this set contains a prohibited production. In the example, the predicate exists(⟨*number*⟩ == "**-900**") ≤ 0.5 would lead to removal of both "**-900**" and "**sqrt(-900)**". This addresses the ambiguity we introduced in the grammar transformation, but not necessarily all ambiguities in the grammar.

4.3 Feasibility Check

In our next step, we identify and eliminate predicate sets that are *infeasible* within the grammar:

Existence. Productions and alternatives corresponding to existence features with $f > 0.5$ predicates are required by the predicate set. We check whether those are reachable within the grammar without prohibited features.
Length. For length features, we check reachability only, as with the existence features.
Maximal Code Point. We check reachability of the production rule, and we check whether there is a terminal symbol that contains the required code point reachable from the production.
Numeric Interpretation. We try to parse the string of the required value starting at the production of the feature.

As an example, let's assume we want to fulfill the predicate set max-char(⟨*number*⟩) == 55. 55 is the ASCII value for 7, and "**7**" can be contained in "**-**"? /[1-9][0-9]*/ ("**.**" /[0-9]+/)?. So the predicate set passes the first test. Next, we check reachability within the sliced grammar. ⟨*number*⟩ is reachable via the ⟨*start*⟩ production, and therefore this test is passed as well.

If a predicate set fails one of those tests, it is infeasible and will not be considered.

4.4 Producing Inputs

In the next step, we produce candidates for derivation sequences that fulfill the given predicates.

To generate an input, we produce the nodes of a parse tree in preorder as described in Section 2.1. During this process, the algorithm needs to make three decisions:

(1) For a concatenation, decide the order of the children.
(2) For a quantification, decide how many children to add.
(3) For an alternation, decide which control form to use.

Each of those decisions corresponds to an element in the derivation sequence, and every time there is more than one possible choice. Each derivation sequence is split into a *prefix* and a *postfix*. When this process creates a new sequence, it lists all possible choices for the first decision in the postfix, and generates one derivation sequence for each of those, by appending each one of those to the prefix of this sequence. For each new derivation sequence, it uses a greedy approach to finish off the sequence. The part that was generated greedily is the new postfix. In the following, we describe this greedy approach.

Each feature is associated with a control form. When the greedy approach has to take a decision, we choose an option such that the label in the new (or next) child node minimizes the distance to the closest of those control forms. When this control form is reached, most features require other heuristics for the subtree of this node.

- For code point features, there is always a terminal symbol that contains the required code point, and we can use it as a target for the distance check.
- For numeric interpretations, we parse the required value, and try to reach the root of this parse tree and the same children as in the known parse tree below the root.
- For length predicates, we can use the distance to the production that the length requirement belongs to, and we can then try to use longer or shorter derivations.

If there is no predicate which influences a decision, we use the alternative which allows for the shortest derivation sequence.

4.5 Searching the Best Derivation Sequence

The process in Section 4.4 can generate different candidates, which need to be *ranked* and *refined*.

The search process maintains a list L of already analyzed derivation sequences. We rank those sequences based on how many predicates they fulfill, and choose the current-best derivation sequence for modification. The newly generated derivation sequences are added to the list, and may be chosen for refinement in the input production from Section 4.4. As soon as a derivation sequence fulfills all predicates, the algorithm returns this input as a solution and terminates.

This search process will not terminate if the feature set is infeasible, that is, if it contains a combination of predicates that cannot be fulfilled. If we could not generate a sample within the timeout of two minutes, we consider a predicate set infeasible.

Within Alhazen, we always have a list of predicate sets when we start the search. While we use only one set for rating derivation sequences, and start with empty L for the next predicate set as soon as we find a solution, we check each derivation sequence against all predicate sets and output all matching inputs for each sequence.

5 EVALUATION

We evaluate Alhazen in three different scenarios:

Predictor. Can Alhazen be used to predict whether an input triggers the bug? (Section 5.2)

Producer. Can Alhazen be used to produce more inputs that trigger the bug? (Section 5.3)

Debugging Aid. Does Alhazen reduce the search space in debugging? (Section 5.4)

As we are not aware of other approaches that act as predictors or producers, we evaluate the *accuracy* of Alhazen in these scenarios. By assessing the quality of decision trees both as predictors and producers, we ensure that they neither overspecialize (which would make them accurate producers, but inaccurate predictors) nor overgeneralize (which would make them accurate predictors, but inaccurate producers). For the third scenario, we evaluate whether Alhazen separates relevant from irrelevant input features, allowing developers to focus on a subset of the input language.

5.1 Evaluation Setup

5.1.1 Subjects. For any predicate over observable program behavior, Alhazen can explain the circumstances that trigger this behavior in terms of input features. In our evaluation, we focus on explaining undesired program behavior (bugs). Table 2 lists our subjects and predicates.

Table 2: Subjects and Predicates of Interest

Bug ID	Predicate of Interest	Bug ID	Predicate of Interest
calculator.1	error message	find.24bf33c0	property
Closure.1978	exception	find.b445af98	regression
Closure.2808	exception	find.e1d0a991	regression
Closure.2842	exception	find.ff248a20	timeout
Closure.2937	exception	grep.c96b0f2c	property
Closure.3178	exception	grep.2be0c659	regression
Closure.3379	exception	grep.3220317a	crash oracle
rhino.385	exception	grep.3c3bdace	crash oracle
rhino.386	exception	grep.55cf7b6a	regression
genson.120	exception	grep.5fa8c7c9	timeout
find.07b941b1	crash oracle	grep.7aa698d3	regression
find.091557f6	crash oracle	grep.c1cb19fe	regression
find.dbcb10e9	crash oracle		

Using the same fuzzer as [18], we found nine bugs in the Google Closure Compiler [10], the Mozilla Rhino JavaScript Runtime [9] and the Genson JSON parser [6].

Those bugs are a good fit for Alhazen because they are triggered by a specific input (by construction, fuzzing generates inputs). All three subjects are written in Java, and report the exception type, file and line number if an error occurs. We used this information for our predicate of interest in the same way as [18] did.

As fourth and fifth subject, we took the `grep` and `find` command line utils from the dbgbench benchmark [13]. Dbgbench provides the means to compile and execute old versions of `grep` and `find`, and documents the bugs that were present in those old versions. We used different predicates of interest here.

Crash. We check whether the program crashes.

Timeout. We check whether the program terminates within 500ms.

Regression. We check whether more recent versions of `grep` or `find` respectively show the same behavior.

Property. `grep` only ever outputs a substring of the input, and find only ever outputs path to existing files. We use checks for those properties as oracles.

Table 2 lists our subjects and the related predicate type.

5.1.2 Evaluation Grammars. In our evaluation, we use a grammar for each subject:

- For the Google Closure Compiler, Mozilla Rhino, and Genson we adapted grammars found in the popular GitHub repository for ANTLR grammars [7]. (ANTLR [28] is a widely known parser generator.)

For `grep` and `find`, we wrote grammars ourselves.

- 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.

5.1.3 Generating Data Sets. As other machine learning approaches, evaluating our approach requires a large set of input data. We could just generate samples randomly, but it is very unlikely to generate a behavior-triggering sample with a pure random producer. Having no behavior-triggering samples in the data set makes it useless.

To avoid this, we use a modified version of the "more of the same" approach taken by Pavese et al. [29]. A word is derived from the grammar by replacing non-terminals with the right-hand side of one of their production rules, until there is no non-terminal left in the resulting sequence. If a production rule has multiple alternatives, a random producer is employed to select which alternative is chosen. The input sample is a word in the grammar, therefore it has a sequence of derivations that generate it. In our sample generation, we increase the probability of choosing the same alternative as in the initial bug-triggering input.

For a word w, let $\#_w(\langle P \rangle)$ be the number of occurrences of $\langle P \rangle$ within w's derivation sequence, and let $\#(\langle P \rangle \to \alpha)$ be the number of occurrences of the alternative $\langle P \rangle \to \alpha$ within this sequence. For ambiguous grammars, let $\#_w(\bullet)$ be the sum of those counts for all possible parse trees.

Using those counts to calculate probabilities directly would generate the same sample over and over again. For a production rule $\langle Q \rangle \to \alpha|\beta|\gamma$, we therefore define a smoothed count, $s(\langle Q \rangle \to \alpha)$ with $s(\langle Q \rangle \to \alpha) = \#(\langle Q \rangle \to \alpha) + 1$. Further, $s(\langle Q \rangle)$ is the sum over the smoothed counts for all alternatives of $\langle Q \rangle$. For our grammar-based fuzzing, the probability to choose the alternative $\langle Q \rangle \to \beta$ (over $\langle Q \rangle \to \alpha$ or $\langle Q \rangle \to \gamma$) is $P(\langle Q \rangle \to \gamma) = \frac{s(\langle Q \rangle \to \alpha)}{s(\langle Q \rangle)}$.

This approach may (still) generate the same sample over and over again, so we remove duplicates, and re-run until we have 1000 unique, behavior-triggering samples (the number of non behavior-triggering samples usually is larger than 1000 at this point). We stopped with a smaller number of samples if 20 re-runs could not generate enough behavior-triggering samples, or a timeout of 1 hour was exhausted. Table 3 gives the number of bug-triggering and non bug-triggering samples for each subject. Please note that we ran this algorithm with the transformed grammars (see Section 2.2).

It is clearly visible that some bugs are harder to trigger then others. For `find.07b941b1`[1] and `find.24bf33c0`[2], it seems to be even easier to trigger the bug than generate a benign input samples. On the other hand, some bugs are particularly hard to trigger. For `grep.7aa698d3`[4], we have just 25 bug-triggering input samples. This bug requires a multibyte character in the input, and a regex matching this multibyte character as an argument to `grep`. The probabilities do not model relations between different parts of the input, so the producer generates this structure only by chance.

Then, we split the generated samples into sets. $\frac{1}{4}$ of the bug-triggering samples (the benign samples, if there were less benign than bug-triggering samples) will be used as training set, and the remaining $\frac{3}{4}$ of them will be the test set. Afterwards, we randomly select benign samples for the training set, such that the training set has the same number of benign and bug-triggering samples.

Table 3: Number of bug-triggering vs. non bug-triggering inputs after generating inputs.

Subject	All Samples		Training Samples	
	benign	bug-triggering	benign	bug-triggering
calculator.1	7163	1041	260	260
closure.1978	8295	868	217	217
closure.2808	3952	1173	293	293
closure.2842	8186	75	19	19
closure.2937	6041	1076	269	269
closure.3178	9558	638	159	159
closure.3379	2915	1152	288	288
rhino.385	4066	1079	270	270
rhino.386	2930	1139	285	285
genson.120	15455	1046	261	261
find.07b941b1	808	1260	202	202
find.091557f6	3487	578	144	144
find.24bf33c0	574	1475	143	143
find.b445af98	3020	76	19	19
find.dbcb10e9	1839	1228	307	307
find.e1d0a991	2758	285	71	71
find.ff248a20	3358	736	184	184
grep.2be0c659	2861	239	60	60
grep.3220317a	3625	475	119	119
grep.3c3bdace	1904	1377	344	344
grep.55cf7b6a	2250	751	188	188
grep.5fa8c7c9	4829	543	136	136
grep.7aa698d3	3075	25	6	6
grep.c1cb19fe	4922	179	45	45
grep.c96b0f2c	3147	50	12	12

Next, we split the remaining samples into sets such that each set is as large as the training set, each set has the same number of benign and bug-triggering samples and each sample is contained in at least one set.

5.2 Alhazen as a Predictor

To evaluate whether Alhazen can predict whether an input is bug-triggering, we generated sample sets with a different approach (see Section 5.1.3), and calculated precision and accuracy on those.

We ran Alhazen on the training set with two different seeds for the random producer, and evaluated each run on all the sets. Within these runs, we performed at most 40 iterations of the feedback loop, and stopped if we did not generate any new samples in an iteration. The results are reported in Table 4. Precision and accuracy numbers are averages over two runs for each set.

We see that Alhazen works very well as a predictor:

> *Used as predictor, Alhazen classifies 92% of all inputs correctly.*

Besides demonstrating the high accuracy of the decision trees produced by Alhazen, this also has some practical value. Most importantly, it means that Alhazen can be used for *automatic workarounds*, diverting potentially failure-inducing input before it reaches the program in question—a feature that would be especially valuable if the failure of interest is a vulnerability. Since Alhazen runs fully automatically, such workarounds can be deployed as soon as a failure is detected.

Table 4: Precision and accuracy when using ALHAZEN as a predictor. All values are averages over 2 runs.

Bug	Precision	Accuracy	Bug	Precision	Accuracy
calculator.1	100.0%	100.0%	find.ff248a20	99.1%	97.6%
closure.1978	97.2%	86.4%	genson.120	100.0%	98.6%
closure.2808	99.6%	96.2%	grep.2be0c65	78.0%	66.0%
closure.2842	98.6%	96.7%	grep.3220317	99.7%	99.4%
closure.2937	99.5%	92.3%	grep.3c3bdac	99.6%	98.9%
closure.3178	96.1%	87.6%	grep.55cf7b6	90.9%	90.6%
closure.3379	94.0%	89.1%	grep.5fa8c7c	100.0%	99.5%
find.07b941b	100.0%	100.0%	grep.7aa698c	79.4%	84.1%
find.091557f	96.7%	95.5%	grep.c1cb19f	87.4%	86.6%
find.24bf33c	87.7%	91.5%	grep.c96b0f2	82.0%	74.8%
find.b445af9	96.1%	96.2%	rhino.385	100.0%	92.6%
find.dbcb10e	100.0%	100.0%	rhino.386	100.0%	96.4%
find.e1d0a99	97.3%	93.6%			
Total				95.0%	92.0%

The only case where ALHAZEN has an accuracy of less than 80% is `grep.2be0c659`[3], where the failure occurs if a given regex matches the input—a property not modelled by our features. While ALHAZEN can check for features which make such a match more likely (e.g. a '.' in the regex), the predictive power suffers.[3]

5.3 ALHAZEN as a Producer

Let us now examine how well ALHAZEN performs as a *producer* for more samples. We ran ALHAZEN on the training sets we generated in Section 5.1.3, and obtained the predicate sets from the final tree. As before, we generated samples for all paths, and generated variations of those subsets as described in Section 4. We then checked whether the prediction of the tree matches actual program behavior.

Table 5 gives the results for this experiment. The "Failing Inputs" column lists the absolute number of new failure-inducing inputs generated. The final decision tree may have multiple paths that lead to the prediction of a failure. ALHAZEN generates a new sample for each of these paths, however, if it runs into a sample that fulfills all predicates on one path while solving another, this sample will also be reported. Hence, a value of 4 means either that the tree had four paths; or it had two paths, and three solutions for one of them were discovered while searching for a solution for the other one.

> *For the large majority of subjects, ALHAZEN produced several new failure-inducing inputs.*

Such additional inputs that trigger the bug can be very valuable in practice. In manual debugging, they can serve as a test set to ensure the bug has actually been fixed. For automated repairs, they can ensure that all aspects of a bug have been fixed, and not only the symptoms of the single failure in question.

The "Precision" column shows the percentage of these failure-inducing inputs within the entire set of inputs. We see that in total, about two thirds of all produced inputs actually trigger the failure.

[3]In practice, what would be helpful here is a more domain-specific feature such as "regex matches". For this evaluation, however, we stick to the generic features introduced in Section 3, which we chose well before starting the evaluation. Over-specialization in the set of features is a real risk for evaluation: In the extreme, a hypothetical "will fail" feature would always yield perfect results.

Table 5: Precision and accuracy when using ALHAZEN as a producer. Precision and accuracy are averages over 2 runs.

Bug	Failing Inputs	Precision	Accuracy
calculator.1	1	100%	100%
closure.1978	1	16.7%	95.5%
closure.2808	11	52.4%	84.4%
closure.2842	0	0.0%	1.000
closure.2937	1	4.5%	72.0%
closure.3178	8	22.9%	80.0%
closure.3379	0	0.0%	100.0%
find.07b941b1	6	100.0%	100.0%
find.091557f6	49	81.7%	90.7%
find.24bf33c0	14	100.0%	77.3%
find.b445af98	20	87.0%	98.4%
find.dbcb10e9	20	100.0%	100.0%
find.e1d0a991	61	58.1%	78.5%
find.ff248a20	94	68.6%	87.1%
genson.120	4	80.0%	94.1%
grep.2be0c659	26	70.0%	90.4%
grep.3220317a	12	97.9%	98.0%
grep.3c3bdace	30	100.0%	100.0%
grep.55cf7b6a	64	100.0%	94.9%
grep.5fa8c7c9	7	100.0%	100.0%
grep.7aa698d3	22	81.5%	96.3%
grep.c1cb19fe	13	41.9%	59.2%
grep.c96b0f2c	0	0.0%	90.9%
rhino.385	8	100.0%	100.0%
rhino.386	13	92.9%	98.1%
Total	3093	68.5%	92.3%

> *On average, 68.5% of the inputs produced by ALHAZEN as failure-inducing actually trigger the failure.*

For programs where a bug is easily triggered, this number indicates a high efficiency of test generation. Even if a test unexpectedly passes, one can simply repeat it with the next input; a precision of 68.5% means that few tests need to be repeated.

For some programs, however, the conditions to trigger a bug are hard to meet, and even harder to model. Indeed, for some subjects, ALHAZEN does not generate any new failure-inducing input at all. For bug `grep.c96b0f2c`[5], ALHAZEN needs to generate an input that contains an empty line, and a regex which matches an empty line; for `closure.2937`[8], the bug is triggered only by a specific nesting of syntax elements. Both regex matching and element nesting are not reflected by our generic input features.

On the other hand, if one uses ALHAZEN to produce passing inputs, a failure is very unlikely. This is reflected in the "Accuracy" column, where we see how many of the inputs produced by ALHAZEN as passing and failing actually are passing and failing. The total shows the overall very high accuracy of ALHAZEN as a producer.

> *On average, 92.5% of the inputs produced by ALHAZEN as passing or failing actually pass and fail as produced.*

5.4 Alhazen as a Debugging Aid

We already have seen that using Alhazen as predictor and producer can be very useful in debugging. It may also be interesting how to use the trees directly. However, it is not yet clear to us, and out of scope for this work, how to present those trees to developers.

Most published automated debugging techniques are evaluated for their *fault localization* capability, that is, their ability to predict where a bug should be fixed. Focusing on code is not appropriate for Alhazen, as it does not predict a bug location; actually, as it treats the program under test as a black box, it neither has nor needs nor produces any concept of a fault location. (This makes Alhazen especially useful if the program in question, say a neural network, has no concept of a fault location either.)

Other automated debugging techniques allow developers to focus on the relevant parts of the *input*; delta debugging [37], for instance, automatically reduces the input to a minimum in which all characters are relevant for producing the bug. The amount by which the search space is reduced, however, depends more on the input (which may contain more or less relevant characters) than the actual approach.

This is more suitable for Alhazen, as Alhazen also works on input representations. However, we do not minimize an existing input, as delta debugging does, instead we report which parts of input structure are relevant. We do not yet know how to present this information to developers, but we assume that a model which reports a small part of the underlying grammar allows the developer to *focus* much more. Smaller is less complicated, and therefore easier to interpret.

For evaluating how much Alhazen can help in reducing the search space, we therefore introduce a measure that is independent of an implementation *and* independent of concrete inputs. In Table 6, we have evaluated how many of the non-terminal symbols and alternatives from the grammar occur in the decision tree. The idea behind this is that each nonterminal and alternative in the grammar stands for a specific concept; the fewer such concepts a programmer has to deal with, the easier it will be for her to capture the specifics of the bug, and eventually to fix it.

If the tree uses exists(⟨*number*⟩) in one node and max(⟨*number*⟩) in another one, this will be counted as one nonterminal symbol, ⟨*number*⟩, the programmer will have to deal with as a relevant concept—in contrast to ⟨*string*⟩, ⟨*loop*⟩ and several more that do not occur in the tree and thus are deemed irrelevant for the bug.

We see that on average, the decision tree makes use of only 3.62% of nonterminals, and only 4.86% of alternatives in the respective grammar. In other words, whatever happens with 96.38% of nonterminals is irrelevant for the respective failure to occur. This means that programmers can indeed focus on a small percentage of relevant input features.

> *The decision trees produced by Alhazen allow programmers to focus on less than 5% of input features.*

The actual percentage highly depends on the size of the grammar. For `calculator.1`, which uses the grammar in Figure 2, 33.3% of the grammar are marked as relevant; however, with such a small grammar, this means two non-terminal rules. For the largest grammar in our selection, JavaScript (used with Closure and Rhino), Alhazen can reduce the relevant elements to 1.67% on average.

Table 6: Tree size and % of grammar used per subject.

Bug	#nodes	#leaves	% of grammar used non-terminals	alternatives
calculator.1	5.0	3.0	33.3	25.0
closure.1978	29.0	15.0	2.0	1.4
closure.2808	13.0	7.0	0.7	0.8
closure.2842	11.0	6.0	1.5	0.3
closure.2937	23.0	12.0	2.4	0.8
closure.3178	38.0	19.5	2.9	1.7
closure.3379	25.0	13.0	1.5	1.4
find.07b941b1	3.0	2.0	0.5	0.1
find.091557f6	28.0	14.5	4.1	0.8
find.24bf33c0	20.0	10.5	4.1	0.5
find.b445af98	19.0	10.0	4.1	0.4
find.dbcb10e9	3.0	2.0	0.0	0.1
find.e1d0a991	33.0	17.0	6.0	0.8
find.ff248a20	35.0	18.0	4.1	1.0
genson.120	18.0	9.5	23.3	23.7
grep.2be0c659	41.0	21.0	6.5	1.2
grep.3220317a	18.0	9.5	2.2	0.7
grep.3c3bdace	11.0	6.0	2.2	0.3
grep.55cf7b6a	33.0	17.0	4.7	1.1
grep.5fa8c7c9	7.0	4.0	0.9	0.2
grep.7aa698d3	11.0	6.0	2.2	0.3
grep.c1cb19fe	23.0	12.0	5.6	0.5
grep.c96b0f2c	16.0	8.5	2.2	0.3
rhino.385	11.0	6.0	1.3	0.4
rhino.386	13.0	7.0	1.1	0.6
Average	19.48	10.24	3.62	4.86

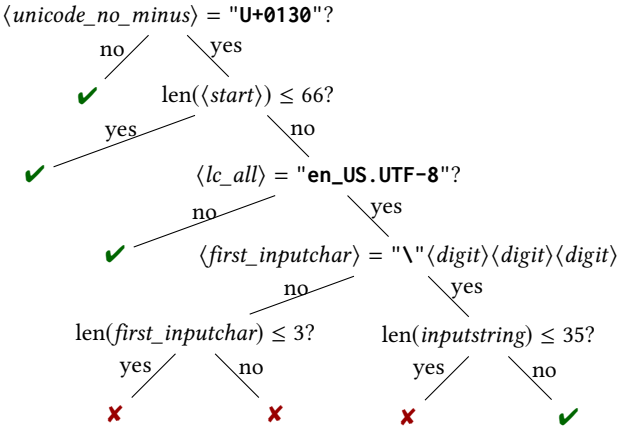

Figure 9: Decision tree for grep.7aa698d3.

5.5 Limitations

While allowing programmers to focus on specific aspects, the inferred decision trees can still be complex. This reflects the complexity of the underlying bugs, which in turn also shows the limits of our approach. In fact, the bugs in our evaluation have non-trivial

descriptions even in natural language, and this complexity is also reflected in the decision trees: As we see in the first two columns of Table 6, the average decision tree has about 20 nodes and 10 leaves.

The tree in Figure 9 for `grep.7aa698d3` is a typical example reflecting complex conditions. The actual bug occurs with all characters where the unicode representation of the lower case variant has fewer bytes than the representation of the upper case variant.

Since our tree can only use numeric comparisons of unicode code points, it cannot fully capture this complex condition (short of listing all characters with this property; note that Unicode characters with this property are not in a continuous area of the Unicode representation.). Instead, the tree checks for the single unicode character "**U+0130**" (I-with-dot) for which the above condition holds. This, of course, is an overspecialization.

The notation "****" ⟨*digit*⟩ ⟨*digit*⟩ ⟨*digit*⟩ is an alternative for how unicode characters can be encoded, so the tree re-iterates that there should be a unicode character in the input.[4]

To more precisely capture the failure condition, as above, one would again have to provide Alhazen with specific features to check for—in our case, a vocabulary over external and internal Unicode properties. But even with the tree being imperfect, it clearly points to the correct features—namely the one important Unicode character as well as the Unicode context. Both of these are very relevant features (out of several hundred in the grep input grammar) for understanding the circumstances of the failure, and provide important hints for fixing it.

> *For best results, the set of input features used by Alhazen should be adapted to the functionality of the program under test.*

5.6 Threats to Validity

In Section 5.3 we use the same algorithm to generate inputs as in Alhazen's iterations. If there is some property of the producer that leads to properties of the generated input that are not described in the decision trees, this would have a positive influence on our results. One such property could be that our producer always tries to minimize the derivation sequences. We are not aware of any other producer that could generate samples from a grammar and a predicate, so there is currently no alternative to this evaluation. At the same time, our claim is that Alhazen can help to generate more inputs which trigger the desired behavior, which is true even if it works only with our producer.

6 RELATED WORK

Grammars and Grammar Mining. The key ingredient to Alhazen is a grammar, used for (1) extracting features from the input by *parsing* it; and (2) *generating* additional inputs for refining and refuting hypotheses. The double usage of grammars as parsers and producers is well-known in the literature. What is new in Alhazen, though, is the generic usage of a grammar to learn features for machine learning and debugging, as well as the combination of parsing and producing as embodiment of the scientific method. Recent developments in *mining grammars* from programs [12, 17, 19] might considerably reduce the effort of writing the required grammars. Parser-directed test generation [25] can eliminate the need for sample inputs to learn grammars from.

Input Reduction. Input Reduction refers to techniques that automatically determine a subset of the input that still reproduces the failure; such simplification is an important prerequisite for debugging. *Delta debugging* [37] is the earliest and simplest technique for reducing inputs; going through a number of tests, it reduces any input to a minimum in which removing any character no longer causes the failure. Later variants of input reduction combine delta debugging with *grammars* for faster reduction [26, 31] or are set up to simplify complex input languages [36].

Alhazen shares a number of ideas with input reduction, notably (1) the goal of eliminating circumstances that are irrelevant for the failure; (2) the concept of working on system input; and (3) the idea of refining or refuting hypotheses via generated tests. There are two core differences, though. First, Alhazen can create theories *from observations only,* without the need for executing additional tests. Second, Alhazen *generalizes* over reduction techniques in that the result is not one single simplified input, but a model for a *set of inputs* that explains and reproduces the failure.

Statistical Fault Localization. Statistical fault localization [21, 24, 35] searches for *statistical associations* between program failures and program runtime failures, notably the execution of specific code locations. Given a sufficiently large number of executions, a small set of lines executed only in failing runs may be determined, making these natural candidates for further investigation or even fixes. While the usefulness of statistical fault localization for programmers is disputed [27], the given locations make important starting points for automated repair techniques [23, 33].

Chen et al.[14] use a decision tree to learn which component in a large internet site causes a specific failure. This is close to our approach, in that it uses decision trees, but still a (kind of) fault localization, as a specific component within a multi-component system is identified.

Just like statistical fault localization, Alhazen creates associations involving program failures. However, the Alhazen associations refer to features of the input, which is an important conceptual difference. Since input features refer to the problem domain and are independent of a given implementation, they may be easier to understand than code locations without any context.

A second important conceptual difference is that Alhazen allows for refining or refuting hypotheses through test generation; this is possible as it uses its grammar as producer. In practice, this means that Alhazen can start with a single failing run only. A similar feature for statistical fault localization would require the ability to generate tests that execute or do not execute a particular line, which is hard in practice and undecidable in general.

Holmes[20] also uses test generation to create more tests similar to a failing test, but does so on pure luck: There is no systematic exploration of hypothesis. Rößler et al. [30] combine statistical fault localization with test case generation, and therefore systematically

[4] What we also see are three len predicates in the tree. The first one captures the fact that you need a minimum length of 66 in our setting to have a unicode character passed as an argument. The other two are cases of *coincidental correlation*—that is, features that happen to match all observations so far, but which have not been refuted by our generation algorithm yet. These features do not significantly impede prediction or production accuracy, however, and would be eliminated with an increasing number of iterations.

test hypothesis. They, however, still work on source code, rather than inputs.

Dynamic Invariants. Dynamic invariants are properties inferred over observations in a given set of program runs. If the argument `x` to `sqrt(x)` is always non-negative, for instance, a dynamic invariant detector like DAIKON [16] can infer the precondition candidate $x \geq 0$. DAIKON achieves this by starting with a large set of potential invariants, keeping only those that apply in all runs.

Like dynamic invariants, ALHAZEN generates abstractions that apply in a set of runs; its predicates, however, apply to input elements rather than function arguments and return values; this also gives ALHAZEN the ability to generate additional tests as needed, which is not easily possible for dynamic invariant generation. However, the ALHAZEN predicates at this point only involve the presence of specific elements or production alternatives. A wider set of features as with DAIKON, including arithmetic, set, and string properties over input elements, could dramatically improve the diagnostic capabilities of ALHAZEN—albeit at the expense of making test generation more difficult.

7 CONCLUSION AND FUTURE WORK

Learning how input features determine program behavior, as ALHAZEN does, opens new perspectives for program understanding and debugging—not only characterizing the circumstances under which a program fails, but also predicting failures for given inputs as well as producing additional inputs that cause failures. Our evaluation shows that ALHAZEN performs all of these tasks with high accuracy, demonstrating the potential of the approach.

We see ALHAZEN as a big step towards better debugging, but also as a platform and opportunity for lots of further research. Future work includes:

Domain-specific features. The "vocabulary" that ALHAZEN can use to characterize failure circumstances is intentionally limited to the very syntactical and numerical basics. Adding more features that cater to the domain of the program at hand could yield much crisper, and possibly even more precise failure characteristics. The challenge is to strive a balance between generality and specificity.

Explainable AI. As the program under test can be arbitrary large or obscure, ALHAZEN can also be used to produce explanations for the behavior of artificial intelligence systems; again, one would need domain-specific features that help distinguishing behavior.

Efficient refinement. We are exploring more sophisticated methods for testing hypotheses that systematically cover language features.

Intercorrelated features. In a grammar, several features intercorrelate with each other: In our expression example, a ⟨*number*⟩ can only occur if a ⟨*function*⟩ occurs as well. The learner can settle on either of these to distinguish passing from failing runs; such choices, however, may impact performance and diagnostic quality of the resulting trees.

Alternate learners. While decision trees can be easily read by humans, other machine learners, such as SVMs or neural networks, could capture failure circumstances much more precisely. The challenge will be to use these learners to generate additional inputs to refine hypotheses, and to extract human-readable descriptions of failure circumstances.

Program analysis. Guidance from static or dynamic program analysis could greatly enhance hypothesis forming and testing.

Beyond failures. The diagnostic capabilities of ALHAZEN easily extend to arbitrary program behaviors—such as the circumstances under which a particular resource is accessed, a data flow takes place, a function is covered, memory is exhausted, and many more.

ALHAZEN and all experiments described in this paper are available for replication and extension. For review purposes, we have compiled a replication package[22] with all code and data at

https://zenodo.org/record/3902142

REFERENCES

[1] 2017. DbgBench - find.07b941b1. https://dbgbench.github.io/find.07b941b1.report.txt
[2] 2017. DbgBench - find.24bf33c0. https://dbgbench.github.io/find.24bf33c0.report.txt
[3] 2017. DbgBench - grep.2be0c659. https://dbgbench.github.io/grep.2be0c659.report.txt
[4] 2017. DbgBench - grep.7aa698d3. https://dbgbench.github.io/grep.7aa698d3.report.txt
[5] 2017. DbgBench - grep.c96b0f2c. https://dbgbench.github.io/grep.c96b0f2c.report.txt
[6] 2017. Genson. https://github.com/owlike/genson. Version 1.4.
[7] 2018. ANTLR Grammars. https://github.com/antlr/grammars-v4/.
[8] 2018. INTERNAL COMPILER ERROR: assigning a class extending class expression #2937. https://github.com/google/closure-compiler/issues/2937
[9] 2018. Mozilla Rhino. https://github.com/mozilla/rhino. Version 1.7.8.
[10] 2019. Google Closure. https://github.com/google/closure-compiler. v20180101.
[11] 2020. CVE-2020-5214. https://nethack.org/security/CVE-2020-5214.html
[12] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110.
[13] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017)*. 1–11.
[14] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 36–43.
[15] Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (1970), 94–102.
[16] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
[17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.
[18] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 189–199.
[19] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321
[20] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proceedings of the 2020 International Conference on Software Engineering*.
[21] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.
[22] Alexander Kampmann, Ezekiel Soremekun, Nikolas Havrikov, and Andreas Zeller. 2020. When does my Program do this? Learning Circumstances of Software Behavior. https://doi.org/10.5281/zenodo.3902142
[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.
[24] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *ACM Sigplan Notices* 40, 6 (2005), 15–26.

[25] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 548–560.

[26] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[27] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[28] Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[29] Esteban Pavese, Ezekiel Soremekun, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2018. Inputs from Hell: Generating Uncommon Inputs from Common Samples. *arXiv preprint arXiv:1812.07525* (2018).

[30] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *Proceedings of the 2012 international symposium on software testing and analysis*. 309–319.

[31] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided Program Reduction *(ICSE '18)*. ACM, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

[32] Philip H Swain and Hans Hauska. 1977. The decision tree classifier: Design and potential. *IEEE Transactions on Geoscience Electronics* 15, 3 (1977), 142–147.

[33] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[34] Wikipedia. 2020. Ibn_al-Haytham. https://en.wikipedia.org/wiki/Ibn_al-Haytham

[35] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[37] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.