Semantic Debugging

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Dominic Steinhöfel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
dominic.steinhoefel@cispa.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@hu-berlin.de

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

ABSTRACT

Why does my program fail? We present a novel and general technique to automatically determine failure causes and conditions, using logical properties over input elements: "The program fails if and only if $\text{int}(\langle \textit{length} \rangle) > \text{len}(\langle \textit{payload} \rangle)$ holds—that is, the given ⟨*length*⟩ is larger than the ⟨*payload*⟩ length." Our AVICENNA prototype uses modern techniques for inferring properties of passing and failing inputs and validating and refining hypotheses by having a constraint solver generate supporting test cases to obtain such diagnoses. As a result, AVICENNA produces crisp and expressive diagnoses even for complex failure conditions, considerably improving over the state of the art with diagnoses close to those of human experts.

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

program behavior, debugging, behavior explanation, testing

ACM Reference Format:
Martin Eberlein, Marius Smytzek, Dominic Steinhöfel, Lars Grunske, and Andreas Zeller. 2023. Semantic Debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616296

1 INTRODUCTION

When software fails, one needs to *debug* it—find the error in the code that causes the failure and fix it. Before digging into the code, however, one must first identify the *circumstances* under which the failure occurs. Such circumstances give important hints on the failure cause, and thus how and where to fix the bug; provide insights into how severe the problem is; and help producing *exact fixes,* preventing patches that only fix a part of the problem.

$\langle \textit{heartbeat-request} \rangle$ ::= `0x1` $\langle \textit{length} \rangle$ $\langle \textit{payload} \rangle$ $\langle \textit{padding} \rangle$
$\langle \textit{heartbeat-response} \rangle$ ::= `0x2` $\langle \textit{length} \rangle$ $\langle \textit{payload} \rangle$ $\langle \textit{padding} \rangle$
$\langle \textit{length} \rangle$ ::= $\langle \textit{int} \rangle$
$\langle \textit{payload} \rangle$ ::= $\epsilon \mid \langle \textit{byte} \rangle\ \langle \textit{payload} \rangle$
$\langle \textit{padding} \rangle$ ::= $\epsilon \mid \langle \textit{byte} \rangle\ \langle \textit{padding} \rangle$

Figure 1: Syntax of TLS Heartbeat exchanges

Let us illustrate the role of failure circumstances referring to the well-known *Heartbleed* problem. In versions between 2012 and 2014, TLS servers were vulnerable to the Heartbleed attack, in which an attacker could extract internal memory contents from a server. The attack was based on the TLS *Heartbeat* protocol, in which a client checks whether a server is still alive by sending it some *payload* string and expecting the same payload to be returned.

The elements of a Heartbeat client request and server response are shown in Figure 1. The client sends a `0x1` byte, followed by the length of the payload, and then the payload itself; extra padding bytes are used to extend the request to the data frame length. The server responds with a `0x2` byte, followed by the same payload, indicating that it has received the request.

The *Heartbleed* attack now consisted of having the *declared* payload length differ from the *actual* payload length. After sending a ⟨*length*⟩ value of, say, 4,000, and a five-character payload of `"Hello"`, the server would reply with `"Hello"`—but followed by another 3,995 bytes that would happen to reside in its memory behind the payload string. Such "over-read" bytes can contain arbitrary information about the server state, including sensitive information such as unencrypted passwords and certificates.

Heartbleed was found by *fuzzing* TLS servers in 2014 [26]. Indeed, simply feeding the grammar from Figure 1 into any grammar-based fuzzer (e.g., [2, 6, 10, 11, 20, 37, 43, 44]) immediately produces a request where ⟨*length*⟩ and the length of the ⟨*payload*⟩ differ, say

$\langle \textit{attack-request} \rangle$ ::= `0x1 0x0123 "hello" 0x0` ...

Sending ⟨*attack-request*⟩ to a server with a memory sanitizer enabled would instantly reveal the invalid memory access. The vulnerability has been present since 2012, and administrators all over the world rushed to patch and update the server software.

To fix the problem, we need to know the exact *circumstances* under which the problem occurs—in our case, something along the lines of "The given payload length is different from the actual payload length"—such that we can characterize, locate, and fix the failure. Recently, two novel approaches for obtaining such circumstances *automatically* have been presented:

- ALHAZEN by Kampmann et al. [19] uses repeated experiments to determine whether specific *properties* of input elements correlate with failure. The set of properties is fixed to existence, length, maximal code point, and numeric interpretation, and ALHAZEN can do a good job if a conjunction of individual properties causes a failure. ALHAZEN, however, does not check for *relationships* between properties, such as $\langle length \rangle$ and the length of $\langle payload \rangle$. The failure circumstances produced by ALHAZEN therefore only relate to the $\langle payload \rangle$ length in isolation:
$$\mathrm{len}\big(\langle payload \rangle\big) \leq 16357 \tag{1}$$
While Equation (1) ("The failure occurs if the $\langle payload \rangle$ has less than 16,357 characters") is a correct *necessary* condition (if $\langle payload \rangle$ is longer, the data frame becomes invalid), it is not *sufficient* for the failure to occur. Nor does it give hints on how to fix the failure.
- *ISLearn* by Steinhöfel and Zeller [38] learns *semantic properties* over input elements that hold for all inputs observed. For this purpose, it checks inputs for *patterns* of these properties that match all observed inputs. In our example, the pattern
$$\mathrm{int}\big(\$1\big) > \mathrm{len}\big(\$2\big) \tag{2}$$
instantiated with $\$1 = \langle length \rangle$ and $\$2 = \langle payload \rangle$ applies to *all* failing *Heartbleed* inputs and would be returned by ISLearn as a common input property:
$$\mathrm{int}\big(\langle length \rangle\big) > \mathrm{len}\big(\langle payload \rangle\big) \tag{3}$$
Equation (3) precisely captures the failure circumstances. However, it is buried in hundreds[1] of additional *coincidental* instantiations that *also* hold for the given failing inputs, such as $\mathrm{len}\big(\langle padding \rangle\big) > \mathrm{len}\big(\langle payload \rangle\big)$ or $\mathrm{len}\big(\langle padding \rangle\big) > \mathrm{int}\big(\langle length \rangle\big)$. In contrast to ALHAZEN, ISLearn has no mechanism to refine diagnoses through experiments.

In this paper, we present AVICENNA[2], a *precise, general, and extensible* approach to determine failure circumstances automatically (Figure 2). AVICENNA builds on the idea that one can decompose the input into individual elements using a grammar and that properties of these input elements can precisely capture failure circumstances. In addition, AVICENNA makes three novel contributions, extending the state of the art:

Quickly determining relevant elements. To narrow down the search space in the (potentially large) set of input elements, AVICENNA uses *Shapley values* [25], a mechanism used to explain AI decisions, to determine *which input elements and derived properties contribute most to the occurrence of failures.* Only these are considered for deriving failure hypotheses, allowing for efficient use even of complex patterns.
In our example, AVICENNA quickly determines that $\langle length \rangle$ and $\mathrm{len}\big(\langle payload \rangle\big)$ contribute most to failure occurrence. $\langle payload \rangle$ by itself contributes little (other than for its length), $\langle padding \rangle$ not at all. To the best of our knowledge, AVICENNA is the first debugging approach to *determine the relevance of input elements for the failure using techniques from explainable AI.*

Reasoning over passing and failing runs. AVICENNA makes use of ISLearn to infer failure-inducing patterns. AVICENNA, however, makes use of both *passing* and *failing* inputs and thus learns input properties that hold for failing runs, but *not* for passing runs. Hence, *coincidental* input properties that hold for *all* inputs are eliminated in the first place, narrowing down the set of candidate properties towards those related to failures, including Equation (3). To the best of our knowledge, AVICENNA is the first debugging approach *to detect arbitrary failure-related patterns in inputs.*

Logical refinement. To further assess diagnosis candidates, AVICENNA uses the ALHAZEN approach to *generate additional test cases.* But while the ALHAZEN generator is limited to its four hard-coded input properties, AVICENNA makes use of the ISLa *constraint language and solver* [38], allowing to express and solve even complex conditions over input elements. This way, AVICENNA can generate hundreds of test inputs satisfying the candidate diagnoses learned by ISLearn. Only Equation (3) reliably produces inputs that cause the failure and is therefore retained; the other "coincidental" properties do not and are thus eliminated.
To the best of our knowledge, AVICENNA is the first debugging approach to *determine diagnoses by logical reasoning and experimentation over input elements.* We therefore call AVICENNA a *semantic* debugging approach in contrast to *lexical/syntactical* approaches like input reduction [12, 39, 41] and *ML-based* approaches like ALHAZEN.

The *Heartbleed* failure circumstance as isolated by AVICENNA and validated by hundreds of test inputs thus reads:
$$\mathrm{int}\big(\langle length \rangle\big) > \mathrm{len}\big(\langle payload \rangle\big) \tag{4}$$
This failure circumstance is 100% precise—the *Heartbleed* failure (as a memory overread) occurs if and only if this condition is met. It could also be used by an *input sanitizer* to precisely predict and prevent *Heartbleed* attacks. This is in contrast to ALHAZEN's diagnosis in Equation (1), which cannot separate attacks from legitimate requests, and the ISLearn diagnosis, embedding the correct property in a myriad of coincidental properties.

With this single failure condition and the test cases, AVICENNA

(1) provides important hints on the nature of the problem;
(2) makes sure that any fix will be well-tested and validated;
(3) supplies a condition for input checkers to detect attacks;
(4) ensures a great start for locating and fixing the fault.

To achieve all this, AVICENNA leverages recent advances in input specification, input generation, and input inference—notably, the ISLa language and input generator [38], which generates inputs

[1] 450 in our experiments; see Table 4.
[2] Ibn Sīnā (Latinized as *Avicenna*; 980–1037) was one of the most significant physicians, astronomers, philosophers, and writers of the Islamic Golden Age. He was one of the earliest proponents of the scientific method of experimentation: In his "Book of Healing" (Kitāb al-Shifā) on science and philosophy, he explained that the ideal situation is when one finds that a "relation holds between the terms, which would allow for absolute, universal certainty." [47]

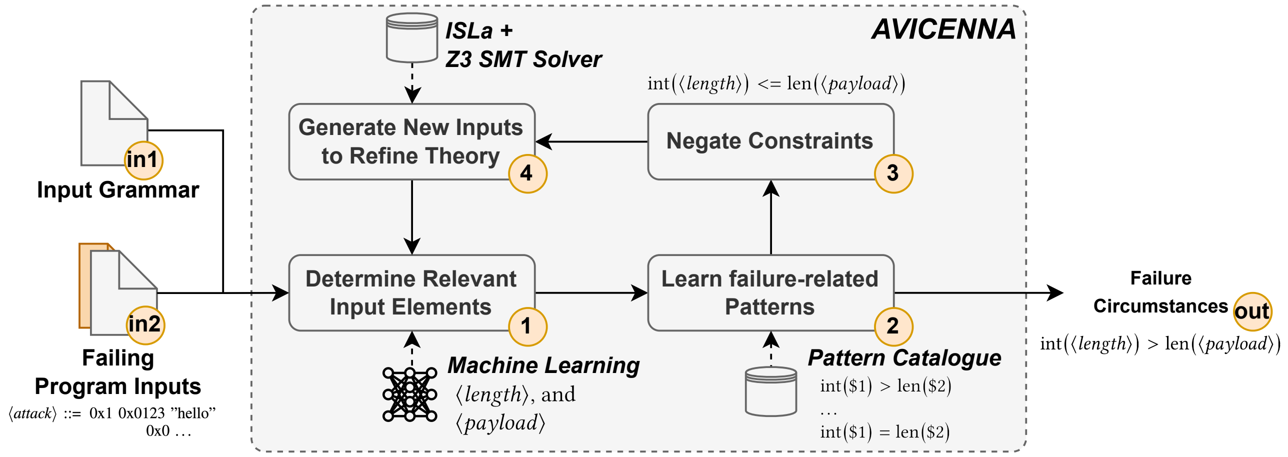

Figure 2: How AVICENNA works. Starting with an input grammar (in_1) and a failing input (in_2), AVICENNA automatically determines the failure circumstances. Then, AVICENNA iteratively refines its failure hypothesis through repeated experiments. To learn the failure-inducing input constraints, AVICENNA leverages both generative models and predictive models to satisfy constraints over grammar elements and to detect relations of input elements, respectively. Ultimately, AVICENNA obtains a theory of the failure circumstances (*out*) that explains and predicts when the behavior in question occurs.

satisfying constraints over grammar elements, and the ISLearn input invariant learner [38], which detects ISLa properties in given inputs. But while ISLa and ISLearn were designed to produce semantically valid inputs, AVICENNA shows that they also enable powerful automated debugging approaches. In our evaluation, AVICENNA determines crisp failure conditions that are much shorter yet more precise than ALHAZEN or ISLearn, closely matching diagnoses collected from human experts. AVICENNA and all experimental data is available as open source (Section 8).

2 BACKGROUND AND RELATED WORK

Automated debugging collectively refers to localization [18, 23, 28, 30, 45, 48, 52], understanding [12, 16, 19, 21, 22, 34, 38, 39, 41, 51], explaining [19, 38], and fixing [13, 24, 27, 35, 49] a faulty system. Automated tools that guide developers toward correcting erroneous program behavior can significantly reduce the cost of software development and improve the overall quality of the software.

2.1 Program Slicing

One of the first seminal automated debugging techniques is *program slicing*, introduced by Weiser [45]. A *slice* is the set of program statements that can be influenced from a given statement (forward slice) or that may have influenced a specific statement (backward slice): "The NULL value in Line 20 comes from Line 18, which executed because of the condition in Line 10." In debugging, a backward slice from a failing statement helps narrow down possible causes in the program code—also by eliminating those parts that could *not* have contributed to the failure. Program slices, however, explain failures in terms of program code—not as (input) circumstances that exist independently of statements and variables.

2.2 Statistical Debugging

Statistical debugging techniques [18, 23, 52] identify execution features that correlate with failure, such as individual code lines mostly executed in failing runs: "Lines 10, 11, and 50 are executed only in failing runs." The resulting statistical models thus expose relationships between specific program behavior and eventual success or failure. However, there may be multiple execution features correlating with failures, and then the developer must determine which of these may be in error. Also, to prevent overfitting, statistical debugging may need *comprehensive sets* of passing and failing runs, which may not exist in practice.

2.3 Delta Debugging

Automated debugging techniques can be made much more precise if they can *generate* additional inputs to narrow down possible failure causes. One seminal example of such *experimental* techniques is Delta Debugging [51], a strategy to effectively reduce failure-inducing inputs: "The failing input can be reduced to the two characters '.' and 'x'." Delta Debugging assumes an *automated test* that determines whether a (reduced) input still produces the failure and whose outcome thus guides the reduction process. In contrast to the above techniques, its result does not refer to the code but, instead, a reduced input that still reproduces the failure.

2.4 Leveraging Input Syntax

Knowing the *input structure* can make experimental debugging techniques far more efficient and also provide better diagnoses. To describe input languages, *Context-Free Grammars (CFGs)* are the most popular formalism, well-studied in theoretical computer science, compiler design, and linguistics [17]. Using input grammars, one can extend reduction to *structured* inputs with Perses [39], C-reduce [34], HDD [16] and HDDr [22, 41]. All these are significantly faster (and often more precise) than lexical Delta Debugging.

While the above approaches still reduce a given failing input into a shorter failing input, the DDSET algorithm [12] aims to find a *pattern* that characterizes the failure. In the pattern, *nonterminals* describe sequences that can take any value as defined by the grammar, thus abstracting over specific contents: "The failure occurs for any input of the form $\langle expr \rangle$ * $\langle expr \rangle$".

2.5 Learning Relevant Input Properties

ALHAZEN [19] requires a *set* of labeled inputs to determine the circumstances of a program's failure automatically. It uses a grammar to parse the inputs into individual *syntactical features*, such as the length of an element or the presence of specific elements. To form the first debugging hypotheses, ALHAZEN needs at least one initial failing input. This hypothesis is shaped by first deconstructing the initial inputs into features with the help of the grammar and utilizing these features to learn a *decision tree.* Next, ALHAZEN uses the tree to learn *associations* between the input features and program failure. Then, ALHAZEN attempts to refine the first hypothesis in an *iterative process*: The decision tree—representing the current explanatory theory—and the grammar are used to construct new inputs to probe the program. These new samples can then be considered to refine the hypothesis in the decision tree.

Eventually, ALHAZEN presents its final theory relating input features with the faulty program behavior: "The failure occurs whenever the length of the $\langle command\text{-}line \rangle$ element exceeds 264". This way, ALHAZEN extends the state of the art beyond DDSET, not only generalizing the syntactic parts of the input but also determining which input features contribute to the failure.

2.6 Dynamic Invariants

During debugging, having a specification of the correct behavior of the program is beneficial. If such a specification is *formal*, automated debugging techniques can leverage it to guide the process. Formally specified *pre-* and *postconditions* as well as *data invariants* can significantly reduce debugging effort—if one knows, for instance, that during an execution, the precondition of some function was satisfied, but not its postcondition, then we can narrow down the search to the execution of said function.

After all, how can we obtain such specifications? One seminal work in specification mining is the DAIKON dynamic invariant detector [7], which takes a set of (passing) runs and for all function arguments and returns, determines from a pattern library whether there are specific properties that hold for these arguments and returns. If a function `y = sqrt(x)`, for instance, is always called with positive values of `x`, it can deduce the invariants $x > 0$, $y > 0$, and even $x = y * y$. Today's *program synthesis* techniques [15] extend over DAIKON by synthesizing complex *formulas* that capture relationships between input and output variables.

However, one downside of all such dynamic techniques is that the mined specifications may *overfit* to the *given* runs. Assuming that $x = 0$ is a valid argument for `sqrt(x)`, for instance, the above invariants overspecialize. Only if DAIKON and the like see an invocation `sqrt(0)` will we get the correct invariants $x \geq 0$ and $y \geq 0$. One may attempt to mitigate this problem by *generating test inputs.* Still, then we run into the problem of potentially violating the very preconditions we want to mine in the first place: From an invocation of `sqrt(-1)`, DAIKON and the like may have lots of undefined behavior from which to learn.

2.7 Input Invariants

Recently, the concepts of specifying and learning invariants were extended to the *system* level, which allows expressing pre- and postconditions over system inputs. The ISLa language [38] combines a context-free grammar with *constraints* as predicates over nonterminals. Given the grammar in Figure 1, for instance, the ISLa constraint "$\text{int}(\langle length \rangle) = \text{len}(\langle payload \rangle)$" expresses that the $\langle length \rangle$ field should hold the length of the $\langle payload \rangle$ field. This way, ISLa leverages the simplicity of CFGs while significantly extending their expressiveness.

The ISLa tool allows to *produce* valid inputs that satisfy the given constraints (using an SMT solver). It can also *check* given inputs against an ISLa specification; *mutate* inputs while maintaining validity; and *repair* inputs to make them valid. The constraint language allows addressing grammar elements with universal and existential quantifiers, relating their positions with structural predicates, and constraining their values using SMT-LIB formulas, making ISLa a robust system for test generation and black-box fuzzing.

2.8 Learning Input Constraints

Along with ISLa [38], the authors also describe *ISLearn*, a pattern-based approach for mining constraints from existing inputs. ISLearn follows the DAIKON approach, using a configurable catalog of common constraint patterns. It instantiates these over all inputs and input elements, retaining those candidates that hold for all given inputs: "In all inputs seen, $\text{len}(\langle payload \rangle) \leq 16357$ holds."

In contrast to DAIKON and like dynamic specification miners, however, ISLearn can make use of *generated inputs* in the first place, leveraging ISLa as a producer. Hence, even given only a partial specification (say, only the grammar), one can first run ISLa to *produce* a myriad of inputs and then have ISLearn infer the constraints only from the *valid* inputs that are accepted by the program under test. This pipeline of test generation and specification mining works at the system level because programs are expected to explicitly *reject* invalid inputs (which specification mining can then ignore)—an assumption that does not hold at the unit level. So far, neither ISLa nor ISLearn has been used for debugging purposes.

3 APPROACH

In this section, we present AVICENNA, our *precise, general, and extensible* approach to determine failure circumstances automatically. The key idea is to leverage both generative and predictive models to satisfy constraints over grammar elements and to detect arbitrary (subject to the catalog patterns) relations of input elements. For this, AVICENNA makes use of four building blocks:

- ISLa's *specification language* allows AVICENNA to express even complex *failure circumstances* as predicates over input elements.
- The ISLearn *tool* allows AVICENNA to learn input properties that are common across all failing inputs.
- The ISLa *tool* allows AVICENNA to *produce* valid inputs—notably inputs that fulfill potential failure-inducing properties, thus allowing for systematic experimentation.

- Finally, AVICENNA follows ALHAZEN in using a *feedback loop* to narrow down failure causes with systematic experiments.

The combination of learning and generating techniques enables AVICENNA to produce a *precise predicate that pinpoints the circumstances under which a program fails.*

AVICENNA starts with a program, a grammar for the input format, and a set of initial inputs (Figure 2 *in*1 & *in*2). To locate and determine the root causes of a program's crash, AVICENNA requires at least one *failing* input. AVICENNA will automatically generate additional failing inputs to strengthen its hypothesized diagnosis. The grammar allows us to associate syntactical features and semantic properties with the observed program behavior.

To detect arbitrary failure-related explanations of the failure circumstances, we proceed in four steps:

(1) We determine the most relevant input elements of the program's failure (Section 3.1).
(2) We instantiate patterns that capture arbitrary relations of the observed failure (Section 3.2).
(3) We produce new inputs to refine and strengthen our hypothesis (Section 3.3), and
(4) We repeat this procedure until a stopping criterion is met (Section 3.4).

3.1 Determining Relevant Input Properties

In contrast to approaches like ALHAZEN, AVICENNA checks passing and failing inputs for *patterns* over input elements and properties. We use these patterns to capture arbitrary relations between input elements and the circumstance of the program's failure. AVICENNA retains those patterns that apply to all or at least the majority of failing inputs, yet not to the passing inputs, and thus makes a *diagnosis candidate.* However, instantiating dozens of patterns with hundreds of input elements and derived values means checking many combinations. Thus, to narrow the search space and the number of possible pattern matches, AVICENNA automatically focuses only on the failure-inducing inputs' essential characteristics. Our tool achieves this by training a machine-learning model and explaining its decisions with *Shapley values.* This mechanism allows us to determine which input elements and derived properties contribute most to a machine learning model's prediction, i.e., the failure of a program. Only those that contribute most are considered during pattern instantiation, allowing for the efficient use even of complex patterns (Figure 2, *Activity 1*).

AVICENNA starts by decomposing each input into its syntactical constituents based on the grammar. The resulting feature vectors and the information if the input is failure-inducing are then used to train a machine learning model. The model eventually associates the program's failure with the occurrence of specific derivation sequences, particularly *non-terminals.* To determine the features that contribute most to the failure in question, AVICENNA employs SHAP [25], a game-theoretic approach explaining the output of a machine learning model based on Shapley values from coalitional game theory. The goal of SHAP is to interpret the model's outcome by computing the contribution of each feature to the final prediction. The final SHAP-value for a feature represents how much the model's prediction changes when we observe that feature. Using these values as an indicator of non-terminals that contribute most to the occurrence of a failure allows us to exclude irrelevant characteristics. This procedure gives us a tremendous advantage over state-of-the-art approaches like ISLearn.

Returning to our example, AVICENNA can quickly determine that $\langle length\rangle$ and $\langle payload\rangle$ contribute most to the failure occurrence. $\langle payload\rangle$, $\langle int\rangle$, and $\langle byte\rangle$ by themselves contribute little (other than for its length), $\langle padding\rangle$ not at all.

3.2 Learning Failure Constraints with Pattern Matching

AVICENNA learns failure-related constraints via *pattern matching*. Steinhöfel et al. [38] showed that input invariants can be mined from existing inputs. Building upon their original pattern-based learner ISLearn, we derive complex semantic constraints that capture the observed failure. We reduce the computational complexity of the pattern matcher by only considering the most relevant input elements for the instantiation of pattern candidates, such as $\langle length\rangle$ and $\langle payload\rangle$.

Figure 2 shows that the learning and candidate generation phase (*Activity 2*) instantiates selected patterns from a provided *pattern catalog* based on the given initial inputs. In the first step, the pattern-matcher instantiates non-terminal placeholders in quantifiers and matches expression placeholder arguments, e.g., $\mathrm{len}(\$1) > \mathrm{num}(\$i)$ with $\langle length\rangle$ or $\langle payload\rangle$. This pattern states that the failure occurs whenever the length of the matched non-terminal is larger than some number $\$i$. The candidates after each instantiation phase are approximately filtered using an ISLearn checker for schematic formulas. Whenever most failure-inducing inputs satisfy an instantiated pattern, AVICENNA retains that pattern.

Let us reconsider our initial *Heartbleed* example. Using AVICENNA, a first failure diagnosis based on the initial inputs may be similar to the following simple constraint:

$$\mathrm{len}(\langle payload\rangle) > 6 \tag{5}$$

This constraint states that whenever the $\langle payload\rangle$ of a request is larger than 6 characters, the program failure occurs. At this early stage, the pattern $\mathrm{int}(\langle length\rangle) > \mathrm{len}(\langle payload\rangle)$ and the above constraint are equivalent regarding their capabilities to partition the *initial* inputs into passing and failing ones. To improve this initial diagnosis and learn the best failure-related constraint, we need to conduct more experiments—with more inputs.

3.3 Validating Hypotheses through Experiments

The failure-related constraints computed initially may be far from perfect. Because of the limited set of initial inputs, AVICENNA can only make basic observations, often resulting in extreme overfitting to the given inputs. To *refine or refute* the initial candidates, we *generate new inputs* to strengthen the learned constraints. AVICENNA generates new inputs according to the most promising extracted failure hypotheses, i.e., constraints that best separate failing from passing inputs. Using the extracted hypotheses allows us to efficiently guide the generation process and focus on the relevant aspects of the failing inputs. AVICENNA is based on the *scientific method:* It tries to *refute* the initial diagnosis hypothesis by *actively* generating new inputs that satisfy the constraints but do not result in the program's failure.

To this end, AVICENNA additionally *negates* the candidate constraints to (*i*) explore the boundaries of the input elements, (*ii*) expand the set of relevant input properties, and (*iii*) refine the surroundings of the input elements. For instance, if both the presence and the absence of the input element ⟨*padding*⟩ result in the failure of the program, then the relevance of ⟨*padding*⟩ is diminished (Figure 2, *Activity 3*). Thus, we generate additional inputs *both* from the original and negated constraints and assign them to the categories failing and passing based on the program's behavior under test. Note that we do not refine the current set of constraints by direct manipulations. Instead, we refine our data set of passing and failing *inputs*, which allows us to (*i*) concretize the set of relevant input elements and (*ii*) infer more precise instantiations in the next inference step.

To efficiently produce new inputs satisfying constraints, we use the recently introduced ISLa fuzzer [38] (Figure 2, *Activity 4*). ISLa not only allows us to *produce* new inputs but also to *validate* complex constraints.

In our running example, we strengthen our initial hypothesis by deriving and adding the following negated constraint:

$$\text{len}(\langle payload \rangle) \leq 6 \tag{6}$$

Hence, with the help of ISLa, AVICENNA will now produce inputs that will have a payload of either *less or euqal* than 6 (for passing inputs) or *more* than 6 (for failing inputs) characters. However, as the length of the ⟨*payload*⟩ is only failure-inducing in combination with the stated ⟨*length*⟩ value, AVICENNA quickly generates inputs that do not satisfy the initial failing diagnosis.

3.4 Refining Hypotheses in a Feedback Loop

To refine the initial diagnosis (failure-related constraints), we repeat the procedure of determining the most relevant input elements, learning constraints, and generating additional inputs. By learning and generating inputs alternately, we can *infer, verify, and generalize* relations between input elements and properties. Most notably, by also considering the negation of the constraints, we generate *adversarial inputs* to falsify our candidate constraints possibly. After only three iterations of the feedback loop, AVICENNA derives the correct failure constraint for the *Heartbleed* example:

$$\text{int}(\langle length \rangle) > \text{len}(\langle payload \rangle) \tag{7}$$

In general, the ideal instantiation will always be among the possible candidate instantiations from the beginning (if it could be derived from AVICENNA's patterns). It will receive a significantly better ranking with the generation of additional test inputs.

4 IMPLEMENTATION

AVICENNA, with the determination of the relevant input elements and the feedback loop, is implemented in Python. We use the latest versions of ISLa and ISLearn to generate and instantiate new failing patterns. In addition, to reduce the computational complexity of the pattern matcher, we pass ISLearn a set of non-terminals that should not be considered during the pattern instantiation. We obtain this exclusion set by determining the most relevant input elements by training a gradient boosting tree based on the XGBoost (Extreme Gradient Boosting) framework [5], an optimized distributed gradient boosting library. Then, we use the SHAP library [25], providing a fast implementation supporting XGBoost to extract the failure-inducing input elements. We consider both the presence and the absence of input elements as necessary. Furthermore, we use the grammar-based fuzzer from the Fuzzing Book [50]. AVICENNA and all experimental data is available as open source (Section 8).

Table 1: Subjects and Grammars

Subject	Grammar	Subject	Grammar
Heartbleed	Figure 1	Pysnooper.1	custom
Calculator	calculator [19]	Pysnooper.2	custom
Genson	JSON [29]	Cookiecutter.1	custom
find.07b941b1	find [19]	Cookiecutter.2	custom
find.091557f6	find [19]	Cookiecutter.3	custom
find.dbcb10e9	find [19]	FastAPI.1	custom
find.ff248a20	find [19]	FastAPI.2	custom
grep.3220317a	grep [19]	FastAPI.3	custom
grep.3c3bdace	grep [19]	FastAPI.4	custom
grep.5fa8c7c9	grep [19]	youtube-dl.1	custom
grep.7aa698d3	grep [19]	youtube-dl.2	custom
grep.c96b0f2c	grep [19]	youtube-dl.3	custom

5 EVALUATION

Let us assess how well AVICENNA fares—both in comparison to human diagnoses, as well as in comparison to the state of the art. We address the following research questions:

RQ1) How does AVICENNA compare against diagnoses provided by *human experts?*

RQ2) How does AVICENNA compare against *ALHAZEN* in terms of diagnosis *complexity* and *accuracy?*

RQ3) How does AVICENNA compare against *ISLearn* in terms of diagnosis *complexity?*

5.1 Evaluation Setup

5.1.1 Evaluation Subjects. To examine the effectiveness of AVICENNA, we evaluate our tool's diagnoses on a set of test subjects similar to those initially covered by Kampmann et al. with ALHAZEN. In total, we evaluate *24* bugs from *nine* different projects of different complexity, namely the *Heartbeat* protocol, a custom *calculator*, the *Genson* JSON parser [4], the command line utils *grep* and *find* from DBGBench [3], and *Pysnooper* [32], *Cookiecutter* [14], *FastAPI* [33] and *youtube-dl* [1] selected from the Tests4Py Benchmark [36]. Tests4Py leverages the bugs present in BugsInPy [46] and extends them with the capability to verify inputs on a system level, which makes it ideal for evaluating AVICENNA. Moreover, each subject of Tests4Py comes with a grammar specifying the input format we can leverage. If a subject of Tests4Py does not provide a CLI for directly accessing the program, Tests4Py already provides a harness as access for execution. DBGBench provides the means to compile and execute old versions of grep and find and document the bugs in those old versions. For the *calculator*, *grep*, and *find*, we used the grammar provided by ALHAZEN. For *Genson*, we adapted the grammar found in the GitHub repository for ANTLR grammars [29]. The handwritten grammars for *grep* and *find* describe complete shell commands consisting of an input, a list of environment variables, and an invocation of the respective command line utility. The subjects, bugs, and used grammar are described in Table 1.

5.1.2 Data Sets. To answer **RQ2** (similar to the evaluation of ALHAZEN), we require sets of test inputs to evaluate the prediction capabilities of AVICENNA's and ALHAZEN's failure diagnosis. To generate these validation inputs, we use the k-path coverage guided, grammar-aware mutation fuzzer provided by ISLearn. With the fuzzer, we automatically generate 100 unique validation inputs for each subject—50 passing and 50 failure-inducing test cases. However, as the failure conditions for the *find* bugs are incredibly narrow, we could not obtain 50 failing inputs within one hour. Thus, we reduced the validation set for the *find* subjects to 20 passing and 20 failing inputs. We measure the respective predictive power of the failure diagnoses based on these validation inputs.

In contrast to the ALHAZEN evaluation, we evaluate AVICENNA's performance with an initial input corpus of two inputs only—one *bug-triggering* and one *passing* input. This decision follows the idea that we want to know if our extended learning process and the feedback loop can generate meaningful additional inputs and thus improve its accuracy and precision. The two initial inputs, one *passing* and one *bug-triggering*, were provided by ALHAZEN, DBGBench, and Tests4Py.

5.1.3 Research Protocol. To answer the research questions, we proceeded as follows: (*i*) First, we started AVICENNA for each subject with the respective grammar and the two provided initial inputs. (*ii*) Then we performed at most 20 iterations of the learning and refinement process. We stopped if we did not generate new inputs in an iteration or AVICENNA could not finish the 20 iterations within one hour. For **RQ1**, we analyzed AVICENNA's failure diagnoses for Heartbleed and the DBGBench subjects. We compare the individual failure conditions of each diagnosis to the bug report provided by experts. We excluded the Calculator, Genson, and Tests4Py subjects as we do not have an expert diagnosis for these bugs. All of AVICENNA's diagnoses, along with details of the respective failures, are available as part of the AVICENNA experimental data (Section 8).

To answer **RQ2**, we compared the predictive power of AVICENNA's diagnoses to ALHAZEN and proceeded as follows: (*i*) First, we generated the evaluation data sets with the mutation fuzzer. (*ii*) Then, we ran AVICENNA and ALHAZEN with the same starting conditions (i.e., with the same grammar and initial inputs). (*iii*) Finally, we measure the performance of the final diagnosis for each subject and approach. With the same starting conditions, rerunning AVICENNA and ALHAZEN did not change their diagnoses. Finally, to answer **RQ3**, we assessed AVICENNA's feedback loop and its performance effects over ISLearn. We compare the number of returned failure diagnoses to answer this research question.

5.2 RQ1: AVICENNA vs. Human Diagnoses

Let us start with our first research question, pitching AVICENNA against human experts. For a substantial subset of bugs in our evaluation setup, we have *diagnoses by human experts* available collected in the DBGBench study [3], in which practitioners would debug real-world bug reports for the *find* and *grep* utilities. As part of their task, these practitioners were asked to determine the *exact circumstances under which the bug would occur* as part of a simplified bug report—the exact problem that AVICENNA is set to address. In our evaluation, we could thus compare the expert bug reports with the AVICENNA diagnoses and assess whether AVICENNA would produce too much, too little, or even misleading information.

Table 2 relates the AVICENNA diagnoses (using ISLa syntax) against the summaries provided by human experts. Fragments marked in **bold** relate to concepts referred to in both the AVICENNA diagnoses and the expert diagnosis; matching concepts are shown in the same color and linked with a line. Fragments marked with *italics* relate to unmatched concepts and thus indicate failure conditions missed by AVICENNA.

Even without complete knowledge of ISLa, the fact that almost all concepts of expert diagnoses exist in the AVICENNA diagnoses and vice versa is striking. Although independently obtained, the natural language diagnoses read as translations of the formal AVICENNA diagnoses, whereas the AVICENNA diagnoses read as a formalization of the natural language diagnoses. In almost all cases, the semantics are identical.

Failure circumstances as produced by AVICENNA are very close to those determined by human experts.

AVICENNA took about 30 minutes on a regular PC to produce a diagnosis for *find* and *grep*; this is slightly slower than it took the DBGBench participants to debug things [3]. However, in the DBGBench study [3], only 58% of patches were correct in the sense that they addressed all failure circumstances. We conjecture that if developers are aware of the exact bug circumstances (as AVICENNA provides them) and able to test their fixes automatically (for instance, by having ISLa produce test cases from the diagnoses), the quality of fixes may very much increase.

5.3 RQ2: AVICENNA vs. ALHAZEN

Let us now compare AVICENNA against state-of-the-art tools. ALHAZEN [19] pioneered the concept of automatically determining failure circumstances, leveraging and producing *decision trees* that express failure conditions. We first compare the *complexity* of diagnoses produced by AVICENNA and ALHAZEN, respectively. From Table 2, we already have seen that a typical AVICENNA diagnosis contains 2–3 conditions, each referring to one property of an input element; the average number of conditions across our entire set of subjects is *2.25*, which is not the case for ALHAZEN diagnoses, though. As listed in [19, Table 6], the average ALHAZEN decision tree has *19.48 nodes,* each one expressing a condition over input elements. We, therefore, conjecture that the AVICENNA diagnoses are much crisper.

AVICENNA diagnoses are only 1/8 as long as ALHAZEN diagnoses.

However, a shorter (and thus more general) failure condition might also result in less *accuracy,* possibly flagging inputs as failure-inducing that are not. We, therefore, evaluate the accuracy of AVICENNA vs. ALHAZEN. For each subject, test input, and tool, we assess whether the tool flags the input as failure-inducing and how that classification compares against the ground truth given by the respective program and oracle. A high *precision* means that outputs flagged as failure-inducing actually induce failures; a high *recall*

Table 2: AVICENNA diagnoses vs. human diagnoses

Bug	AVICENNA diagnosis (using ISLa syntax [38])	Expert Bug Description
Heartbleed	`str.to.int(<length>) > str.len(<payload>)`	"Attackers can send Heartbeat requests with the **value of the length field greater than** the actual **length of the payload**" [9]

grep.7aa698d3	`exists <utf8> in <lc_all>:`
	`  <utf8> = "UTF-8" and`
	`exists <ignore_case> in <general_options>:`
	`  <ignore_case> = "-i"`
		"If grep **conducts a case-insensitive search (-i)** on an input *that contains multibyte characters* and **the locale is UTF8**, then grep prints a match of incorrect length." [3]

grep.5fa8c7c9	`exists <patterns> in <command>:`
	`  <patterns> = "''" and`
	`exists <utf8> in <lc_all>:`
	`  inside(<utf8>, <lc_all>) and`
	`exists <fixed_string> in <cmd_1>:`
	`  inside(<fixed_string>, <cmd_1>)`
		"Searching with **grep -F** for an **empty string** in a **multibyte locals** [sic] would freeze grep." [3]
		
		(Note: ⟨*fixed_string*⟩ expands to `-F` and `-fixed-strings`)

grep.c96b0f2c	`exists <regex_> in <patterns>:`
	`  <regex_> = "^$" and`
	`(exists <ignore_case> in <matching_control>:`
	`   inside(<ignore_case>, <matching_control>) or`
	` exists <line_no> in <output_line_prefix_control>:`
	`  inside(<line_no>, <output_line_prefix_control>))`
		"Options **-i** and **-n** will not work when applied to an **empty line**" [3] [*in a UTF-8 locale*]
		
		(Note: ⟨*ignore_case*⟩ expands to `-i` and `-ignore-case`; ⟨*line_no*⟩ expands to `-n` and `-line-number`)

grep.3c3bdace	`exists <extended_regex> in <matcher_selection>:`
	` inside(<extended_regex>, <matcher_selection>) and`
	`exists <repetition> in <patterns>:`
	`  <repetition> = "*"`
		"Core dump with pattern `'(^|␣)*( |$)'`" [3] [and **-E option**]
		
		(Note: ⟨*extended_regex*⟩ expands to `-E` and `-extended-regexp`)

grep.3220317a	`exists <bracket_expr> in <first_expression>:`
	`  inside(<bracket_expr>, <first_expression>) and`
	`exists <utf_characters> in <bracket_char>:`
	`  inside(<utf_characters>, <bracket_char>)`
		"Segmentation fault on **multibyte character classes**" [3]
		
		(Note: ⟨*bracket_expr*⟩ expands to `[...]` in a regular expression; ⟨*utf_characters*⟩ occur within ⟨*bracket_char*⟩, i.e. the characters within ⟨*bracket_expr*⟩)

find.07b941b1	`exists <match_opts> in <find_expression>:`
	`  <match_opts> = "-regex " and`
	`exists <character_expr_no_minus> in <first_expression>:`
	`  <character_expr_no_minus> = "."`
		"find segfaults when using **-regex**, for instance `./find -regex '.*'`" [3]

find.091557f6	`exists <file_properties> in <find_expression>:`
	`  <file_properties> = "-type f" and`
	`exists <directory_name> in <starting_dir_list>:`
	`  inside(<directory_name>, <starting_dir_list>)`
		"assertion failure on *symbolic link loop*: Let's say we accidentally create a symlink loop `$ mkdir tmp; cd tmp` and `$ ln -s a b; ln -s b a` and use find to find files and follow symlinks inside the tmp-folder: `../find -L -type f`" [3]

find.dbcb10e9	`exists <digit> in <last_modified>:`
	`  inside(<digit>, <last_modified>) and`
	`exists <numeric_arg> in <find_command>:`
	`  inside(<numeric_arg>, <find_command>)`
		"**-mtime** produces segmentation fault, e.g., `./find -mtime 2`" [3]
		
		(Note: ⟨*last_modified*⟩ expands to `-mtime`)

find.ff248a20	`exists <ln_file> in <ln>:`
	`  <ln_file> = "al_ln -s . link" and`
	`exists <find_expression_or_empty> in <command>:`
	`  <find_expression_or_empty> = " -follow"`
		"infinite loop with **-follow**; e.g., $ `mkdir testingfindagain; ln -s . testingfindagain/symlink; ./find testingfindagain -follow`" [3]

For details on all DBGBench bugs (including expert bug descriptions), visit https://dbgbench.github.io/.
For the Heartbleed description, see https://www.synopsys.com/blogs/software-security/heartbleed-bug/.

Table 3: Precision and recall of the produced failure conditions: AVICENNA vs. ALHAZEN

Subject	AVICENNA		ALHAZEN	
	Precision	Recall	Precision	Recall
Heartbleed	**100%**	**100%**	11%	10%
Calculator	100%	100%	100%	100%
Genson	100%	**100%**	100%	74%
find.07b941b1	80%	**100%**	**100%**	85%
find.091557f6	**47%**	**40%**	31%	25%
find.dbcb10e9	93%	70%	**95%**	**90%**
find.ff248a20	83%	75%	83%	**100%**
grep.3220317a	**77%**	**100%**	72%	56%
grep.3c3bdace	**77%**	**100%**	74%	56%
grep.5fa8c7c9	**96%**	**92%**	94%	58%
grep.7aa698d3	**64%**	**28%**	18%	12%
grep.c96b0f2c	73%	**96%**	**91%**	86%
Pysnooper.1	100%	100%	100%	100%
Pysnooper.2	100%	100%	100%	100%
Cookiecutter.1	**78%**	**94%**	13%	15%
Cookiecutter.2	88%	**90%**	88%	83%
Cookiecutter.3	**100%**	**100%**	11%	24%
FastAPI.1	100%	100%	100%	100%
FastAPI.2	100%	100%	100%	100%
FastAPI.3	69%	**90%**	69%	78%
FastAPI.4	**100%**	**100%**	40%	31%
youtube-dl.1	**88%**	**98%**	14%	65%
youtube-dl.2	**98%**	**100%**	11%	66%
youtube-dl.3	**94%**	**100%**	66%	92%
Average	**88%**	**91%**	66%	67%

means that inputs that induce failures are identified as such. Our results are detailed in Table 3. We see that despite having shorter (and more general) failure conditions, the precision of AVICENNA is at least on par with the state-of-the-art ALHAZEN.

The average precision of AVICENNA diagnoses is on par with ALHAZEN diagnoses.

In a debugging context, however, precision is not that important, as false positives can easily be identified by them not causing a failure. Of much larger interest are the *false negatives*, as they indicate that the diagnosis may *miss* some failure conditions, which in turn may lead to incomplete fixes. This is actually where AVICENNA surpasses ALHAZEN in *22* of the *24* subjects, leveraging its capability to identify more general failure circumstances.

The average recall of AVICENNA diagnoses surpasses the recall of ALHAZEN diagnoses.

Generally speaking, the ALHAZEN diagnoses follow the original failure-inducing input much closer than the more general AVICENNA diagnoses, which is a reasonable approach. Diagnoses of previous techniques like syntactic input reduction also stay close to the original failing input, still providing benefits for programmers. However, AVICENNA does a better job in exploring the *surroundings* of the original failing test case, thus inferring generalizations that state-of-the-art approaches like ALHAZEN miss. With this in mind, we see that ALHAZEN archives slightly better results for *find.dbcb10e9* and *find.ff248a20*. We argue that this is due to their extremely narrow failing conditions. Consequently, the validation set is similar to the original failing input; thus, overspecializing to the failure-inducing

```
exists ⟨utf8⟩in <lc_all>:                          AVICENNA
  <utf8> = "UTF-8" and                          grep.7aa698d3
exists <ignore_case> in <general_options>:
  <ignore_case> = "-i"
```

```
exists ⟨unicode_no_minus⟩ in start:                 ALHAZEN
  ⟨unicode_no_minus⟩ == "U+0130" and           grep.7aa698d3
exists ⟨start⟩ in start:
  str.len(⟨start⟩) >= "66" and
exists ⟨first_inputchar⟩ in start:
  ⟨first_inputchar⟩ == "\"⟨digit⟩⟨digit⟩⟨digit⟩ and
(exists ⟨first_inputchar⟩ in start:
    str.len(⟨first_inputchar⟩) >= "3" or
 exists ⟨inputstring⟩ in start:
    str.len(⟨inputstring⟩) >= "35")
```

Figure 3: AVICENNA vs. ALHAZEN: Diagnoses for *grep.7aa698d3*.

inputs is beneficial for ALHAZEN's prediction — even though the diagnoses are not as general as those produced by AVICENNA. This property is even further highlighted if we take a closer look at, for instance, the diagnoses for *grep.7aa698d3*. We show the differences by translating the decision tree [19, Figure 9] produced by ALHAZEN to an equivalent ISLa formula (Figure 3).

By comparing this diagnosis to AVICENNA and the expert diagnosis (Table 2), we quickly realize that the produced formula, and thus the equivalent decision tree, not only overspecializes to the failure-inducing inputs but is also more complex than the explanation produced by AVICENNA. Even with the shorter diagnosis, AVICENNA captures the circumstances of the failure better by not overspecializing to the single Unicode character "`U+0130`".

5.4 RQ3: AVICENNA vs. ISLearn

In the last part of our evaluation, we pitch AVICENNA against ISLearn, the input invariant learner [38]. As discussed earlier, AVICENNA makes extensive use of ISLearn, gradually refining the detected input invariants in a feedback loop. Is this feedback loop necessary, and does AVICENNA improve over ISLearn? To answer this question, we compared the number of invariants produced by ISLearn to the number of invariants obtained at the end of an AVICENNA run. For ISLearn, we only considered invariants with recall and specificity estimates of 100%: From ISLearn's point of view, all reported invariants have an equivalent quality. Table 4 summarizes our results. We see that running ISLearn on its own is not a viable alternative, producing *hundreds to tens of thousands* of invariants. Additionally, ISLearn was not able to produce any invariants for *Cookiecutter* within one hour (*n/a*). AVICENNA can reduce this number—to a single invariant in all cases—solely by refining these invariants through additional experiments.

The AVICENNA feedback loop is crucial for providing crisp failure circumstances.

5.5 Threats to Validity

Our evaluation has the following threats to validity:

Table 4: # Invariants produced by AVICENNA vs. ISLearn

Subject	AVICENNA	ISLearn	Subject	AVICENNA	ISLearn
HeartBleed	1	451	Pysnooper.1	1	194
Calculator	1	39	Pysnooper.2	1	8844
Genson	1	777	Cookiecutter.1	1	n/a
find.07b941b1	1	1006	Cookiecutter.2	1	n/a
find.091557f6	1	361	Cookiecutter.3	1	n/a
find.dbcb10e9	1	69	FastAPI.1	1	247
find.ff248a20	1	620	FastAPI.2	1	346
grep.3220317a	1	475	FastAPI.3	1	607
grep.3c3bdace	1	4	FastAPI.4	1	711
grep.5fa8c7c9	1	54	youtube-dl.1	1	245
grep.7aa698d3	1	26,711	youtube-dl.2	1	3
grep.c96b0f2c	1	456	youtube-dl.3	1	4427

Internal Validity. The threats to internal validity relate to the correctness of AVICENNA's implementation and the correctness of our experiments. AVICENNA has 4,000 lines of code in total, and formally proving its correctness would be cumbersome at least. However, the results in Table 2 strongly suggest, based on face validity, that AVICENNA operates as intended. Concerning our experiments, we can rule out major mistakes due to the tuning of parameters since AVICENNA does not rely on tuning parameters. Additionally, we performed the experiments with realistic settings; more precisely, we set the upper bound for the number of AVICENNA iterations to 20 and used a one-hour time limit for the experiments to show the practical value of AVICENNA.

External Validity. The threats to external validity are mainly related to the selection of the programs and bugs in our study. We have used DBGBench [3], a common debugging benchmark, and Tests4Py [36], a benchmark comprised of many different projects, as sources of programs and real bugs. The *Heartbleed*, *Calculator*, and *Genson* subjects show that AVICENNA can produce a diagnosis for authentic bugs. Consequently, we would argue that the results of our study are transferable and generalizable to other programs and bugs.

6 LIMITATIONS

Despite its advances, AVICENNA is not a perfect diagnosis tool, as there are fundamental limitations. To illustrate the challenge, let us have a look at the *grep.7aa698d3* bug, for which AVICENNA misses the fact that the input "contains multibyte characters" (Table 2).

If *grep* conducts a case-insensitive search (`-i`) on an input containing multibyte characters and the locale is UTF8, then grep prints a match of incorrect length. When conducting the case-insensitive search, `EXECUTE_FCT` first converts the input to lowercase (search.c:388). The length of the match is computed for the match in the lower-case input (search.c:555). However, a multibyte character can take 1 byte less in lower-case: The lengths of the normal-case and lower-case inputs differ. The computed match_size value could be half the expected value (grep.c:1081–1085); the match in the normal-case input is printed with incorrect length (grep.c:1091).

A perfect failure condition would thus read

$$\text{len}\Big(\text{utf-8}\big(\text{upper}(\langle arg\rangle)\big)\Big) > \text{len}\Big(\text{utf-8}\big(\text{lower}(\langle arg\rangle)\big)\Big) \quad (8)$$

which, in contrast to the actual AVICENNA diagnosis (Table 2) would be 100% accurate.

Why can AVICENNA not synthesize such a diagnosis? The problem is twofold. First, AVICENNA needs an appropriate *vocabulary* even to express the failure conditions —in our case, functions like "utf-8" and "upper". Second, we are facing a *combinatorial complexity* problem, as there can be an arbitrary number of combinations of predicates, functions, operators, and nonterminals to consider when deriving a diagnosis. Eventually, the problem can be framed as a *program synthesis* problem—we want a formula (or program) that exactly predicts when an input causes a failure, which, of course, resembles the *halting problem* and thus is undecidable in general.

However, an undecidable problem may still be solvable under specific (often common) conditions; our results illustrate this. Possible ways to obtain even more comprehensive diagnoses include:

Domain-specific vocabularies. Adding catalogs with domain- or program-specific patterns, predicates, and functions would allow AVICENNA to detect these failure conditions in the first place. If a significant portion of *grep* bugs, for instance, were related to UTF-8 handling, then adding an appropriate vocabulary would allow AVICENNA to detect and express related failure causes.

Program analysis. Static and dynamic program analysis could reveal important functions and properties directing the search toward a meaningful diagnosis. In *grep.7aa698d3*, for instance, a data flow analysis could reveal that the input is subject to UTF-8 and case transformations, thus guiding the search to vocabularies related to these properties.

Documentation. The `grep` documentation relates the `-i` option to case sensitivity, which in conjunction with the current AVICENNA diagnosis (Table 2) could again guide the search.

Expert interaction. Guidance such as above could also be provided by a human expert, starting with the initial AVICENNA diagnosis and giving hints on where to search further.

Program synthesis. Finally, *program synthesis* could provide further inspiration to synthesize *higher-order combinators* [8], *recursive functions* [31], or even *relational queries* [40, 42]. In the extreme, a tool like AVICENNA would thus not only produce a single predicate but synthesize a program that would precisely characterize and predict failure conditions.

7 CONCLUSION AND FUTURE WORK

With AVICENNA, we introduce a technique to *fully automatically determine failure circumstances,* expressed as *logical properties over input elements.* The approach is general and uses an extensible vocabulary, a powerful constraint solver, and a refinement loop to conduct additional experiments, resulting in crisp and to-the-point diagnoses, matching the precision of human experts. Furthermore, by formulating the failure diagnosis problem as finding a predicate over input elements, AVICENNA opens up a wealth of future research opportunities. Notably, each of the diagnoses can be instantiated into a myriad of test cases for exploring the surroundings of a bug, validating fixes, and preventing regressions. All this is good news for the future of automated debugging.

Besides general improvements regarding performance and generality, our future work will focus on the following topics:

Location conditions. Besides *failure* vs. *success,* AVICENNA can determine the circumstances of *arbitrary execution predicates*—for instance, the (input) circumstances under which a particular location is reached, which not only helps with understanding code. Notably, solving the circumstances of a code location L yields inputs that specifically target L—a great feature for test generators.

Resource consumption. Another interesting class of execution predicates to apply AVICENNA on are *non-functional* properties such as *resource consumption*: Under which circumstances does this program require more than one GB of memory? Or more than one second for a request? Extending AVICENNA with numerical approximation algorithms can precisely narrow these circumstances.

Fault localization. AVICENNA diagnoses allow generating an unlimited number of passing and failing runs; this should enable a much more precise fault localization than with only a few failing runs. We want to evaluate this.

Automated repair. The AVICENNA diagnoses provide important hints on how to repair code automatically. We want to *map* input elements to variables and *relate their properties* to synthesize fixes that exactly capture failure conditions.

ACKNOWLEDGMENTS

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GR 3634/4-2 Emperor (261444241). M. Eberlein conducted this work during a research visit at CISPA Helmholtz Center for Information Security.

8 DATA AVAILABILITY

The current version of our AVICENNA prototype can be downloaded from

https://github.com/martineberlein/avicenna

REFERENCES

[1] Remita Amine. 2021. youtube-dl. https://github.com/ytdl-org/youtube-dl

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[3] 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. https://dbgbench.github.io/

[4] Eugen Cepoi. 2017. Genson. https://github.com/owlike/genson.

[5] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[6] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Proceedings of the 12th Symposium on Search-Based Software Engineering (SSBSE 2020)*.

[7] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

[8] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[9] Anil Gajawada. 2016. Heartbleed bug: How it works and how to avoid similar bugs. https://www.synopsys.com/blogs/software-security/heartbleed-bug/

[10] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[11] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org/citation.cfm?id=3155562.3155573

[12] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Abstracting Failure-Inducing Inputs. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (Virtual Event). ACM, 237–248. https://doi.org/10.1145/3395363.3397349

[13] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[14] Audrey Roy Greenfeld. 2022. Cookiecutter. https://github.com/cookiecutter/cookiecutter

[15] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[16] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 861–871. https://doi.org/10.1109/ASE.2017.8115697

[17] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[18] 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* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[19] Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. 2020. When does my Program do this? Learning Circumstances of Software Behavior. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3368089.3409687

[20] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering* 22, 2 (2017), 928–961. https://doi.org/10.1007/s10664-015-9422-4

[21] Lukas Kirschner, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 75–86. https://doi.org/10.1145/3377811.3380329

[22] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical Delta Debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 16–22. https://doi.org/10.1145/3278186.3278189

[23] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1065010.1065014

[24] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[25] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 4765–4774. https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html

[26] Eric Markowitz. 2014. Behind the Scenes: The Crazy 72 Hours Leading Up to the Heartbleed Discovery. https://www.vocativ.com/tech/hacking/behind-scenes-crazy-72-hours-leading-heartbleed-discovery/

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the*

38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[28] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32. https://doi.org/10.1145/2000791.2000795

[29] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[30] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 609–620. https://doi.org/10.1109/ICSE.2017.62

[31] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[32] Ram Rachum. 2019. PySnooper - Never use print for debugging again. https://github.com/cool-RR/pysnooper

[33] Sebastián Ramírez. 2018. FastAPI. https://github.com/tiangolo/fastapi

[34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[35] Ridwan Salihin Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 390–405. https://doi.org/10.1145/3453483.3454051

[36] Marius Smytzek, Martin Eberlein, Batuhan Serce, Lars Grunske, and Andreas Zeller. 2023. Tests4Py: A Benchmark for System Testing. arXiv:2307.05147 [cs.SE]

[37] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).

[38] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://publications.cispa.saarland/3596/

[39] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

[40] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

[41] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. 2022. The effect of hoisting on variants of Hierarchical Delta Debugging. *Journal of Software: Evolution and Process* online first (2022). https://doi.org/10.1002/smr.2483

[42] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[44] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[45] Mark David Weiser. 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* Ph.D. Dissertation. USA. AAI8007856.

[46] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943

[47] Wikipedia. Accessed 2022-07-26. Avicenna. https://en.wikipedia.org/wiki/Avicenna.

[48] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[49] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[50] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/

[51] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[52] Alice Zheng, Michael Jordan, Ben Liblit, and Alex Aiken. 2003. Statistical Debugging of Sampled Programs. In *Advances in Neural Information Processing Systems*, S. Thrun, L. Saul, and B. Schölkopf (Eds.), Vol. 16. MIT Press. https://proceedings.neurips.cc/paper/2003/file/0a65e195cb51418279b6fa8d96847a60-Paper.pdf

Received 2023-02-02; accepted 2023-07-27