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, one needs them 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 talk, I present and introduce tools and techniques that automatically learn circumstances of a given failure, expressed over features of input elements. I show how to automatically infer input languages as readable grammars, how to use these grammars for massive fuzzing, and how to systematically and precisely characterize the set of inputs that causes a given failure – the "language of failure".
Failure

Actually, my work always has been about failures. (The work itself has been less of a failure.)

As a PhD student still, Dorothea Lütkehaus and myself built GNU DDD, a GUI front-end for command line debuggers. Great for debugging failures.

Later, my co-workers and I would mine version and bug repositories to see where in a program the most bugs would be fixed. This is a map of Firefox components (boxes) and vulnerabilities (shades of red).
Almost all vulnerabilities are in JavaScript.

Another contribution my name is associated with is simplifying failure-inducing inputs. Here's a long input that causes a program to fail.

Yet, only a part of this input actually is relevant for the failure.
Delta Debugging automatically determines this failure-inducing subset.

Delta Debugging takes away parts of the input and checks whether the failure still occurs.

Such reduced inputs can be invalid, though.
Then, delta debugging takes out smaller parts and repeats.

At the end, it easily determines which characters are necessary for the failure to occur.

Such as these ones, for instance.
These things made me an ACM Fellow "For contributions to automated debugging and mining software archives".

Failure

- You can mine version and bug histories to find out where the failures are
- You can simplify inputs to find out what causes the failure
- You can make a career out of failure

Which tells you that you can make a career out of failures.

The Language of Failure

Okay, that was failures. Now, let's move to languages.
Fuzzing

8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 * 
-(8 - 5 - 6)) * (-((-+((+(4))))) - ++4) / 
(-++++(5.6 - --(3 * -1.8 * +(6 * +-(((-(-6) 
* ---+6)) / + (-+-+-7 * (-0 * (++++(((((2)) + 8 
- 3 = ++9.0 = --(--7 / (2 / +++6.37) + (1) 
/ 482) / +++-+0)))) + ++5 + (1) 
(-+2 - ---2 - - -9.8)))) / 5 = ----998

Fuzzing means to throw random inputs at a program to see if it crashes.

But if you just take sequences of random characters and throw them at an interpreter, all you're going to get is syntax errors. (It's okay to test syntax error handling, but this should not be all.)

In order to get syntactically valid inputs, you need a specification. A grammar specifies the set of inputs as a language.
You may have seen grammars as parsers, but they can also be used as producers of inputs.

You start with a start symbol

You may have seen grammars as parsers, but they can also be used as producers of inputs.

You start with a start symbol
Grammars as Producers

\[
\text{(start)} ::= \langle \text{expr} \rangle
\]

\[
\text{(expr)} ::= \langle \text{term} \rangle \text{+} \langle \text{expr} \rangle | \langle \text{term} \rangle \text{-} \langle \text{expr} \rangle | \langle \text{term} \rangle
\]

\[
\text{(term)} ::= \langle \text{factor} \rangle \text{+} \langle \text{term} \rangle \text{/} \langle \text{factor} \rangle | \langle \text{factor} \rangle \langle \text{int} \rangle \langle \text{int} \rangle \langle \text{int} \rangle
\]

\[
\text{(factor)} ::= + \langle \text{factor} \rangle | - \langle \text{factor} \rangle | \langle \text{expr} \rangle | \langle \text{int} \rangle \langle \text{int} \rangle \langle \text{int} \rangle
\]

\[
\text{(int)} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
\langle \text{expr} \rangle
\]

which then subsequently gets replaced according to the production rules in the grammar.

If there are multiple alternatives, you randomly choose one.

Grammars as Producers

\[
\text{(term)} - \langle \text{expr} \rangle
\]
Grammars as Producers

\[ \texttt{start} ::= \texttt{expr} \]
\[ \texttt{expr} ::= \texttt{(term)} \times \texttt{expr} \mid \texttt{(term)} \div \texttt{(expr)} \mid \texttt{(expr)} \]
\[ \texttt{term} ::= \texttt{(factor)} \times \texttt{term} \mid \texttt{(factor)} \div \texttt{factor} \mid \texttt{(factor)} \]
\[ \texttt{factor} ::= \texttt{(term)} \times \texttt{factor} \mid \texttt{(term)} \div \texttt{factor} \mid \texttt{digit} \]
\[ \texttt{digit} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

\[ \texttt{factor} - \texttt{expr} \]

---

Grammars as Producers

\[ \texttt{start} ::= \texttt{expr} \]
\[ \texttt{expr} ::= \texttt{(term)} \times \texttt{expr} \mid \texttt{(term)} \div \texttt{(expr)} \mid \texttt{(term)} \]
\[ \texttt{factor} ::= \texttt{(factor)} \times \texttt{factor} \mid \texttt{(factor)} \div \texttt{factor} \mid \texttt{(factor)} \]
\[ \texttt{term} ::= \texttt{(digit)} \texttt{factor} \mid \texttt{digit} \]
\[ \texttt{digit} ::= \texttt{digit} \texttt{digit} \]

\[ \texttt{int} \cdot \texttt{int} - \texttt{expr} \]

---

Grammars as Producers

\[ \texttt{start} ::= \texttt{expr} \]
\[ \texttt{expr} ::= \texttt{(term)} \times \texttt{expr} \mid \texttt{(term)} \div \texttt{(expr)} \mid \texttt{(term)} \]
\[ \texttt{factor} ::= \texttt{(factor)} \times \texttt{factor} \mid \texttt{(factor)} \div \texttt{factor} \mid \texttt{(factor)} \]
\[ \texttt{term} ::= \texttt{(digit)} \texttt{factor} \mid \texttt{digit} \]
\[ \texttt{digit} ::= \texttt{digit} \texttt{digit} \]

\[ \texttt{digit} \cdot \texttt{int} - \texttt{expr} \]
Grammars as Producers

\[
\text{(start)} ::= \text{<expr>}
\]
\[
\text{(expr)} ::= \text{(term)} \text{ + (expr)} | \text{(term)} \text{ - (expr)} | \text{<term>}
\]
\[
\text{(term)} ::= \text{<factor> \text{ * (term)} / (factor)} \text{ | (factor)} \text{ | (int)} \text{ | (int)} \text{ . (int)}
\]
\[
\text{(factor)} ::= \text{+ (factor)} | \text{- (factor)} | \text{(expr)} \text{ | (int)} \text{ | (int)} \text{ . (int)}
\]
\[
\text{(int)} ::= \text{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
\]

\[
\text{(digit)} \cdot \text{(digit)} - \text{<expr>}
\]

Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.

8. \text{<digit>} - \text{<expr>}

Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.

8.2 - \text{<expr>}

Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.
Grammars as Producers

Actually, a pretty complex arithmetic expression.

Fuzzing with Grammars

These can now be used as input to your program.
A couple of years ago, we used a JavaScript grammar to fuzz the interpreters of Firefox, Chrome and Edge.

My student Christian Holler found more than 2,600 bugs, and in the first four weeks, he netted more than $50,000 in bug bounties. If you use a browser to read this, one of the reasons your browser works as it should is because of grammar-based fuzzing.

And if you are interested in how to use grammar for fuzzing, the book will give you lots of inspiration.
Learning the Language

But all of this still requires a grammar in the first place.

Fuzzing with Grammars

So where did you get this grammar from?

Mining Grammars

So let me tell you a bit about how to mind such grammars. The idea is to take a program that parses such inputs and extract the input grammar from it.

```c
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}

void parse_term() {
    ...}

void parse_factor() {
    ...}

void parse_int() {
    ...}

void parse_digit() {
    ...}
```
The interesting thing is that there is a correspondence between individual rules in the input grammar and locations in the parsing code.


The concept of consumption establishes this correspondence. A character is **consumed** in a method \( m \) if \( m \) is the last to access it.

For each input character, we dynamically track where it is consumed.

\[
1 \times (8 - 5)
\]
During program execution we can track where characters are consumed using dynamic tainting.

This gives us a tree like structure.

Which we can augment with caller-callee relations.
Even for those functions which do not consume anything.

If we take the function names and only use the nouns, we can use those nouns as non-terminal symbols.

From these parse trees, we can now mine a grammar.
A term obviously can consist of another term, a multiplication symbol, and a factor.

So we add this as a rule to our grammar.

And likewise for other symbols.
From this single input, we already get the basics of a grammar.

And if we add more inputs, ...
 Completing the Grammar

\[
\text{(start)} ::= \text{(expr)} \\
\text{(expr)} ::= \text{(term)} \cdot \text{(expr)} \mid \text{(term)} \cdot \text{-(expr)} \mid \text{(term)} \\
\text{(term)} ::= \text{(term)} \cdot \text{(factor)} \mid \text{(factor)} \\
\text{(factor)} ::= \text{(expr)} \mid \text{(int)} \\
\text{(int)} ::= \text{(digit)} \\
\text{(digit)} ::= 0 \mid 1 \mid 2 \mid 5 \mid 8
\]

We now have successfully mined our example grammar.
Mimid: A Grammar Miner

Our Mimid grammar miner takes a program and its inputs and extracts a grammar out of it. This grammar can directly be used by fuzzers, parsers, and humans.

The extracted grammars are well structured and human readable as you can see in this grammar extracted from a JSON parser.

Humans can edit these grammars.
Performance by highly trained professional. Do not try this at home, your university or anywhere else.

This change to the grammar injects SQL statements everywhere. Do not do this at home, folks – thank you.

For instance, by assigning probabilities to individual productions.

Or by inserting magic strings that program analysis would have a hard time finding out.
Mimid: Evaluation

- Mined grammars can generate ~98% of the actual language
- Mined grammars can parse ~92% of the actual language
- Works on modern combinatory parsers, too

The grammars extracted by Mimid are accurate as producers as well as as parsers.

Learning the Language

- Learn readable language specs (grammars) automatically
- Mined input grammars are accurate: ~98% generating, ~92% parsing
- Learn from given program only; no input samples required

So this was about learning (input) languages.

Mining Grammars without Samples

Our grammar miner needs inputs in the first place. But we also have specific test generators that systematically cover all alternatives in a parser. So technically, all you need is the program to test.
Learning the Language

- Learn readable language specs (grammars) automatically
- Mined input grammars are accurate: ~98% generating, ~92% parsing
- Learn from given program only; no input samples required

And now for the main point.

Learning the Language of Failure

Circumstances of Failure

1 * (6 - 5)

Program under test

For which other inputs does this hold?

We have seen how single inputs cause failures. But are these the only inputs?
We want to know the set of inputs that causes the failure – in other words, the language. To this end, we parse the input into a tree.

To find out whether the failure occurs for other integer values too, …

… we replace parts of the parse tree (8) by newly generated alternatives (27).
and find that this one fails as well.

Actually, the program fails for any integer in this position. So we can come up with an abstract pattern that represents the set of failing inputs.

By repeating this, we can come up with a general pattern of which all instantiations cause the failure. These instantiations also serve as test cases for validating a fix.
Our tool DDSet takes a program, a failing input, and a grammar, and produces such a pattern of failure.

In our evaluation, this works really well.

But we can go even further. What other features in the input cause a failure?

- For 19 of 22 bugs, concrete inputs could be abstracted into patterns
- 91.8% of inputs from patterns were semantically valid; 98.2% reproduced the failure
- Patterns serve as diagnostics as well as producers

- Failure could also occur for other inputs – how about / or + ?
- Failure could depend on non-structural features like length, value, etc.
We introduce a number of input features, including existence, length, and maximum and minimum values of specific input elements.

These features together with a pass and fail label then go into a machine learner which produces a predictive model.

Actually, the produced model serves as a model of the program as it comes to failures or non-failures.
In our experiments, we use decision tree learners as their results are easy to understand.

Here is a decision tree that classifies the three inputs on the left. We see that the existence of the digit 8 serves as classifying feature. The model is consistent with all the observations made so far.

The learner also could come up with another model over the presence or non-presence of multi digit integers. Is any of these correct?
Training a Classifier

Labeled Inputs

Features

Failures are scarce – so how can we get sufficiently many inputs?

Refining Models

We create **new inputs** right from the model learned so far.

Training a Decision Tree

Specifically, for every path in the tree, we generate more inputs.
Training a Decision Tree

Labeled Inputs

1 \times (8 - 5) \times

27 + 3 \checkmark

-1 \times 23 + 4 \checkmark

\exists 8?

\text{yes} \quad \text{no}

New inputs

without \exists 8

\checkmark \quad \times

New inputs

with \exists 8

\checkmark \quad \times

So, here are more inputs with and without the digit 8. For every input, we test whether the failure occurs.

For these inputs, the old hypothesis no longer holds. The decision tree now comes up with a more detailed model.
If we repeat this a number of times, we end up with this decision tree which now accurately characterizes the circumstances of failure.

And this now tells us under which circumstance the failure occurs – namely, whenever the distributive law can be applied.

Alhazen was an early proponent of the concept that a hypothesis must be supported by experiments based on confirmable procedures or mathematical evidence—an early pioneer in the scientific method five centuries before Renaissance scientists.

Alhazen takes a program, failing and passing inputs, and a grammar.

By abstracting over observations, and gradually refining hypothesis through experiments, Alhazen produces a predictive (and generative) model on whether failures occur or not.
Since the passing and failing inputs can come from a fuzzer, and since the grammar can come from a miner, ...

... Alhazen actually only need the program to be debugged to produce a model.

Alhazen works great as a predictor and as a producer. Also, the decision trees refer to a small subset of the input grammar, allowing developers to focus on these.
Here is an example. Alhazen correctly determines the circumstances of a grep crash.

Since my time as a PhD student, I always wanted to have a slide with NetHack on it. This is how Alhazen explains the circumstances of a NetHack crash.

So this is learning the language of failure – the set of inputs that causes a program to fail.

Learning the Language of Failure

- Learned behavior models explain, produce, predict (failing) behavior
- Models refer to terms from problem domain rather than internals
- Generalizes to arbitrary predicates on program behavior
The Fuzzing Book

Tools and Techniques for Generating Software Tests

By Andreas Deller, Rahul Gopinath, Marius Wilkins, Gordon Fraser, and Christian Haller

About this Book

Welcome to "The Fuzzing Book"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by automating software testing, specifically by generating input automatically. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

From fuzzingbookاكتساب YouTubede "wmcgraff"
That's all! If you like this work, and want to know more, follow me on Twitter or visit my homepage at https://andreas-zeller.info/. See you!