

# Learning the Language of Failure

CASA Distinguished Lecture, Bochum, 2020-06-18

Andreas Zeller  
with Rahul Gopinath and Zeller's team at CISPA



## Learning the Language of Failure

Andreas Zeller, CISPA Helmholtz Center for Information Security

Joint work with Rahul Gopinath and Zeller's team at CISPA

**Watch:** <https://www.youtube.com/watch?v=3ZW1Dl2Pxvl>

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".

<https://andreas-zeller.info/>  
<https://www.cispa.saarland/>

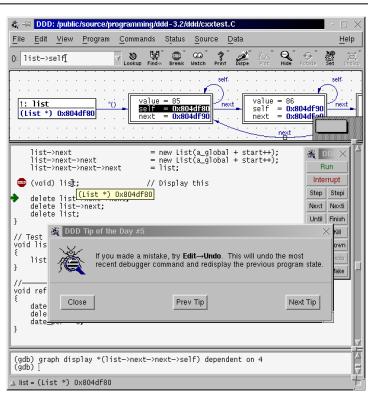
Failure

Welcome everyone to "Learning the Language of Failure". These five words will follow us throughout the talk. To begin, let's talk about failures.



## Failure

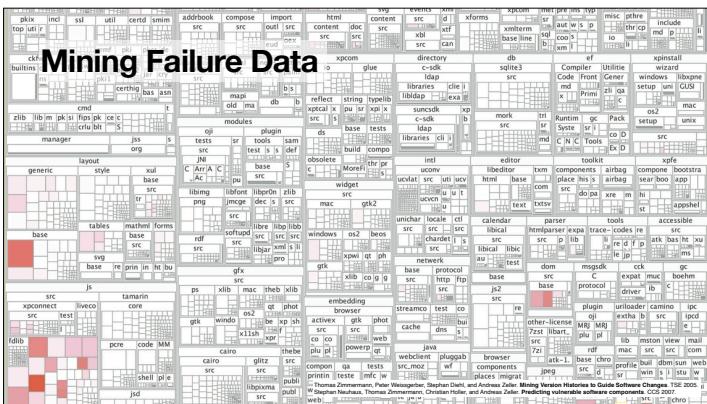
## Debugging Failures



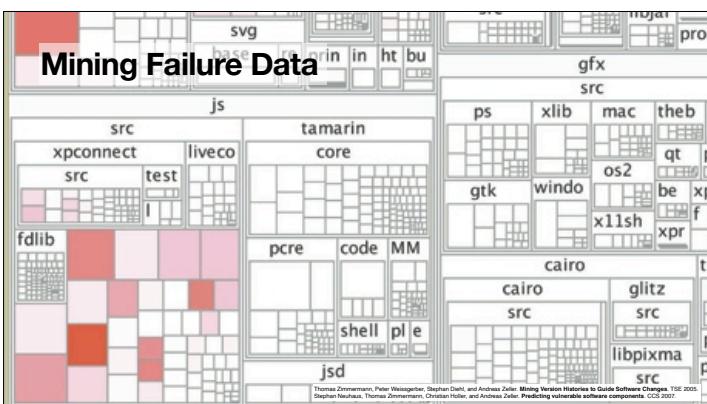
Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debugger  
http://www.cs.tu-berlin.de/~zeller/DDD

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.

## Simplifying Failures

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * ---+090
```



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

## Simplifying Failures

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * ---+090
```



Yet, only a part of this input actually is relevant for the failure.

## Delta Debugging

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * ---+090
```



Delta debugging automatically determines this failure-inducing subset.

## Delta Debugging

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * ---+090
```



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

## Delta Debugging

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * --+090
```

Andreas Zeller and Ralf Hildebrandt: Simplifying and Isolating Failure-Inducing Input. TSE 2002.



## Delta Debugging

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * --+090
```

Andreas Zeller and Ralf Hildebrandt: Simplifying and Isolating Failure-Inducing Input. TSE 2002.



## Delta Debugging

```
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 / (1 / +++6.37) + (1)  
/ 482) / +--+0)))) * -+5 + 7.513))) -  
(+1 / ++((-84))))))) * ++5 / +-(-2 - -  
+-9.0))) / 5 * --+090
```

Andreas Zeller and Ralf Hildebrandt: Simplifying and Isolating Failure-Inducing Input. TSE 2002.



## Delta Debugging

```
1 * (8 - 5)
```

Andreas Zeller and Ralf Hildebrandt: Simplifying and Isolating Failure-Inducing Input. TSE 2002.



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 / (1 / +++6.37) + (1)
/ 482) / +--+0))) * -+5 + 7.513))) - +
(+1 / ++((-84))))))) * ++5 / +-(-2 - --
+-9.0))) / 5 * ---+090
```



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

## Dumb Fuzzing

```
(144 60 )5(5-(05*/( * *)910)25/509505)3)/  
09211762 /(7*+22)76-+/29+/4**2+  
8( )04/844  
4)632/3/7 *0525+ )7*
```



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.)

## Grammars

Specify a **language** (= a set of inputs)

```
(start) ::= (expr)  
(expr) ... (term) + (expr) | (term) - (expr) | (term)  
(term) ::= (term) * (factor) | (term) / (factor) | (factor)  
(factor) ::= + (factor) | - (factor) | ( (expr) ) | (int) | (int) . (int)  
(int) ::= (digit) (int) | (digit)  
(digit) ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Expansion rule

Nonterminal symbol

Terminal symbol

In order to get syntactically valid inputs, you need a specification. A **grammar** specifies the set of inputs as a **language**.

## Grammars as Producers

```
(start) ::= (expr)  
(expr) ... (term) + (expr) | (term) - (expr) | (term)  
(term) ::= (term) * (factor) | (term) / (factor) | (factor)  
(factor) ::= + (factor) | - (factor) | ( (expr) ) | (int) | (int) . (int)  
(int) ::= (digit) (int) | (digit)  
(digit) ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

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

## Grammars as Producers

```
(start) ::= (expr)  
(expr) ... (term) + (expr) | (term) - (expr) | (term)  
(term) ::= (term) * (factor) | (term) / (factor) | (factor)  
(factor) ::= + (factor) | - (factor) | ( (expr) ) | (int) | (int) . (int)  
(int) ::= (digit) (int) | (digit)  
(digit) ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

**<start>**

You start with a start symbol

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<start>

Nikola Hrirkov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<expr>

Nikola Hrirkov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

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

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<term> - <expr>

Nikola Hrirkov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

If there are multiple alternatives, you randomly choose one.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<term> - <expr>

Nikola Hrirkov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<factor> - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<int> . <int> - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<digit> . <int> - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<digit> . <digit> - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> | <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

## 8. <digit> - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> | <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

## 8.2 - <expr>

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

## Grammars as Producers

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> | <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
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 / (1 / +++6.37) + (1)
/ 482) / +----+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(---2 - --
+-9.0))) / 5 * ---+090
```

Nikola Hrakov and Andreas Zeller: Systematically Covering Input Structures, ASE 2019.

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

## Fuzzing with Grammars

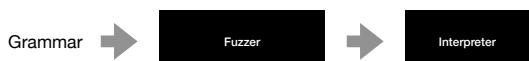
```
8.2 - 27 - -9 / +((+9 * --2 + ---+-((-1 *
+(8 - 5 - 6)) * (-(-+((+(4)))) - ++4) / +
(-+-((5.6 - -(3 * -1.8 * +(6 * +-((-(-6)
* ---+6)) / +-(-++-7 * (-0 * (+(((2)) + 8
- 3 - ++9.0 + ---(--7 / (1 / +++6.37) + (1)
/ 482) / +----+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(---2 - --
+-9.0))) / 5 * ---+090
```



Actually, a pretty **complex** arithmetic expression.

These can now be used as input to your program.

## Fuzzing with Grammars



## Fuzzing with Grammars

JavaScript Grammar

LangFuzz  
Fuzzer



Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. USENIX 2012.

A couple of years ago, we used a JavaScript grammar to fuzz the interpreters of Firefox, Chrome and Edge.

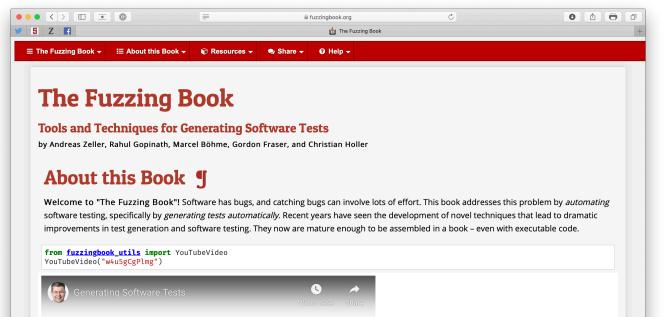
My student Christian Holler found more than 2,600 bugs, and in the first four weeks, he netted more than \$50,000 in bug bounties. If you use a browser to read this, one of the reasons your browser works as it should is because of grammar-based fuzzing.

## The Language of Failure

- A language spec trivially gives you *infinitely many, syntactically valid* inputs
- Generation can be guided by grammar coverage/code coverage/probabilities
- Easily taught and applied

So this was the language of failure.

fuzzingbook.org



We have put all our knowledge on fuzzing and grammars into a book named the *fuzzingbook*, where you can actually try out all the basic algorithms yourself.

## The Language of Failure

- A language spec trivially gives you *infinitely many, syntactically valid* inputs
- Generation can be guided by grammar coverage/code coverage/probabilities
- Easily taught and applied

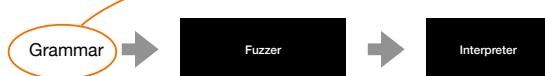
And if you are interested in how to use grammar for fuzzing, the book will give you lots of inspiration.

## Learning the Language

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

## Fuzzing with Grammars

Where do we get the grammar from?



So where did you get this grammar from?

## Mining Grammars

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



## Rules and Locations

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



## Consumption

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

The character is last accessed (consumed) in this method



Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

## Consumption

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

1 \* ( 8 - 5 )

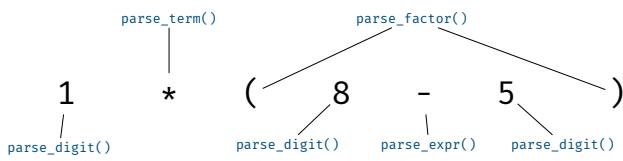
Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

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.

The interesting thing is that there is a correspondence between individual rules in the input grammar and locations in the parsing code.

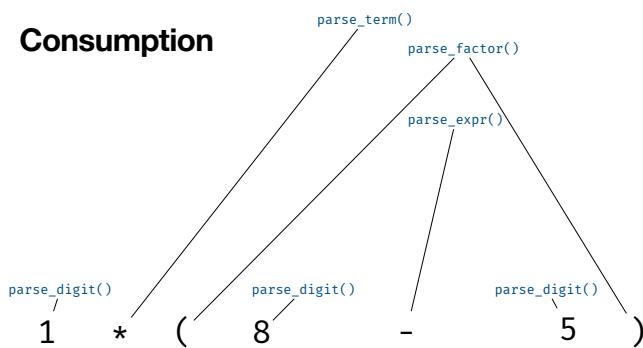
The concept of consumption establishes this correspondence. A character is **consumed** in a method  $m$  if  $m$  is the last to access it.

## Consumption



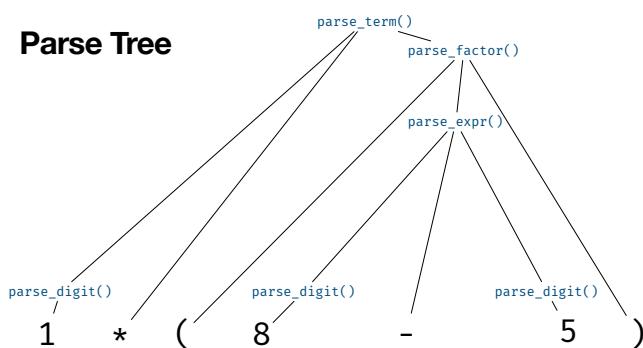
During program execution we can track where characters are consumed using dynamic tainting.

## Consumption



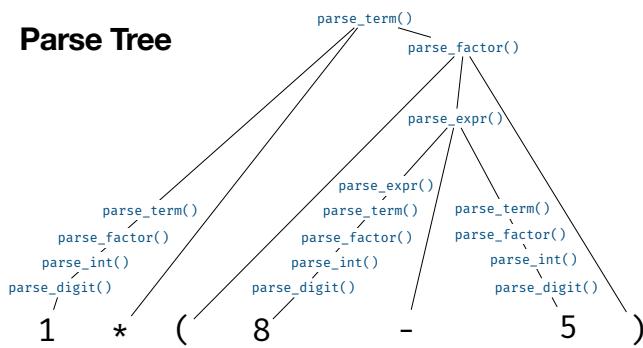
This gives us a tree like structure.

## Parse Tree



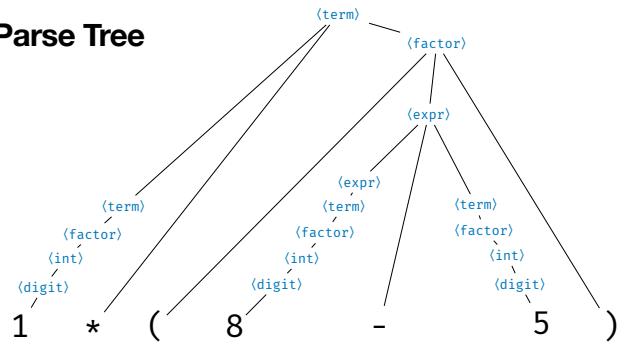
Which we can augment with caller-callee relations.

## Parse Tree



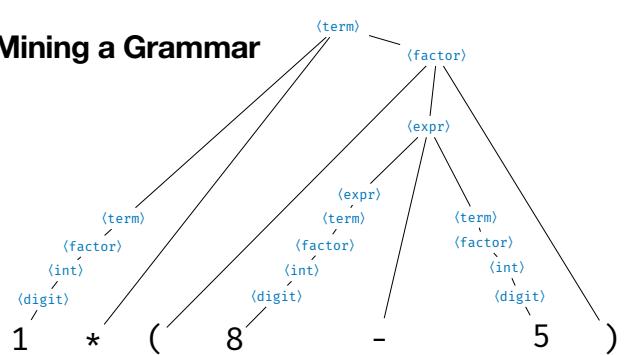
Even for those functions which do not consume anything.

# Parse Tree



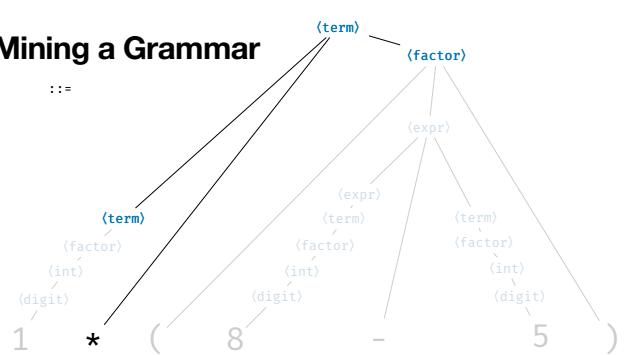
If we take the function names and only use the nouns, we can use those nouns as non-terminal symbols.

## Mining a Grammar



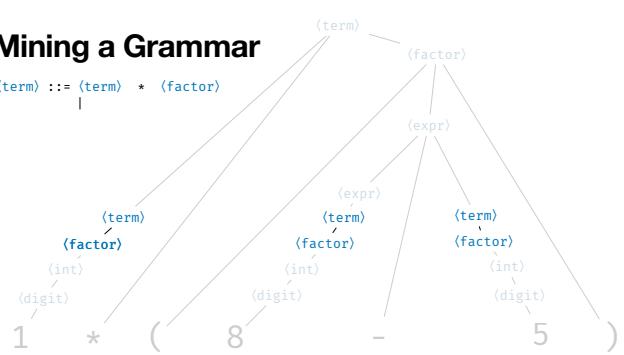
From these parse trees, we can now mine a grammar.

## Mining a Grammar



A term obviously can consist of another term, a multiplication symbol, and a factor.

## Mining a Grammar



So we add this as a rule to our grammar.

## Mining a Grammar

```
(term) ::= (term) * (factor)
       | (factor)
```

::=

```
(factor) ::= (int)
```

```
(int) ::= (digit)
```

Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

And likewise for other symbols.

## Mining a Grammar

```
(term) ::= (term) * (factor)
```

```
| (factor)
```

|

```
(factor) ::= (expr)
```

```
|
```

```
(int) ::= (digit)
```

```
(digit) ::= 1 | 5 | 8
```

Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

## Mining a Grammar

```
(start) ::= (expr)
```

```
(expr) ::= (term) - (expr) | (term)
```

```
(term) ::= (term) * (factor) | (factor)
```

```
(factor) ::= ( (expr) ) | (int)
```

```
(int) ::= (digit)
```

```
(digit) ::= 1 | 5 | 8
```

Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

From this single input, we already get the basics of a grammar.

## Completing the Grammar

```
(start) ::= (expr)
(expr) ::= (term) - (expr) | (term)
(term) ::= (term) * (factor) | (factor)
(factor) ::= ( (expr) ) | (int)
(int) ::= (digit)
(digit) ::= 1 | 5 | 8
```

↑  
Parse tree

↑  
0 + 2

Rahul Gopinath, Björn Märtin, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

And if we add more inputs, ...

## Completing the Grammar

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= ( <expr> ) | <int>
<int> ::= <digit>
<digit> ::= 0 | 1 | 2 | 5 | 8
```



Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

... the grammar reflects the structure of these additional inputs.

## Completing the Grammar

```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= ( <expr> ) | <int>
<int> ::= <digit>
<digit> ::= 0 | 1 | 2 | 5 | 8
```



Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

## Completing the Grammar

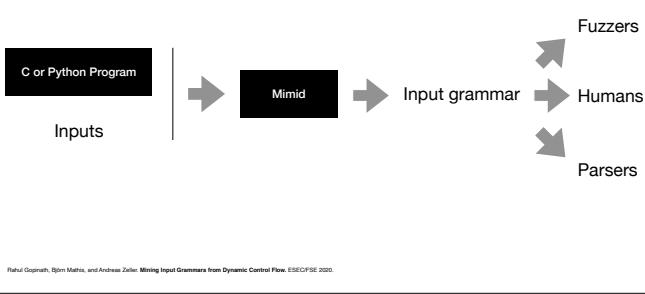
```
(start) ::= <expr>
<expr> ::= <term> + <expr> | <term> - <expr> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | ( <expr> ) | <int> | <int> . <int>
<int> ::= <digit> <int> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

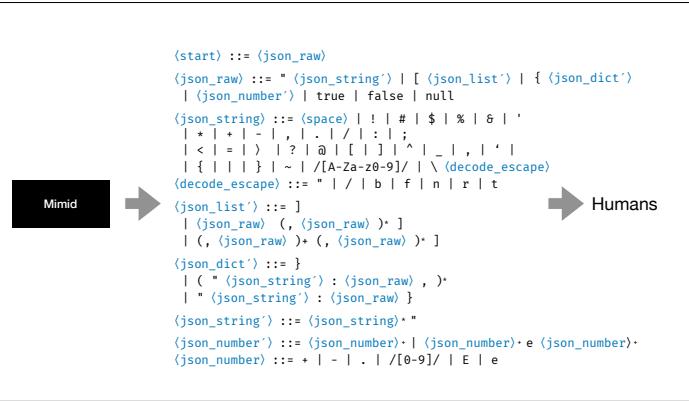
We now have successfully mined our example grammar.

## 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.

Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.



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.

```

(start) ::= <json_raw>
<json_raw> ::= " <json_string> " | [ <json_list> ] | { <json_dict> }
| <json_number> | true | false | null
<json_string> ::= (space) | ! | # | $ | % | & | '
| * | + | - | , | . | / | : | ;
| < | = | > | ? | @ | [ | ] | ^ | _ | , | ' |
| { | | | } | ~ | /[A-Za-z0-9]/ | \ <decode_escape>
<decode_escape> ::= " | / | b | f | n | r | t
<json_list> ::= ]
| <json_raw> (, <json_raw> )+ ]
| (, <json_raw> )+ (, <json_raw> )+
<json_dict> ::= ]
| ( " <json_string> : <json_raw> , )
| " <json_string> : <json_raw> }
<json_string> ::= <json_string>*
<json_number> ::= <json_number>+ e <json_number>.
<json_number> ::= + | - | . | /[0-9]/ | E | e

```

For instance, by assigning probabilities to individual productions.

```

(start) ::= <json_raw>
<json_raw> ::= " <json_string> " | 10% [ <json_list> ] | 50% { <json_dict> }
| <json_number> | true | false | null
<json_string> ::= (space) | ! | # | $ | % | & | '
| * | + | - | , | . | / | : | ;
| < | = | > | ? | @ | [ | ] | ^ | _ | , | ' |
| { | | | } | ~ | /[A-Za-z0-9]/ | \ <decode_escape>
<decode_escape> ::= " | / | b | f | n | r | t
<json_list> ::= ]
| <json_raw> (, <json_raw> )+ ]
| (, <json_raw> )+ (, <json_raw> )+
<json_dict> ::= ]
| ( " <json_string> : <json_raw> , )
| " <json_string> : <json_raw> }
<json_string> ::= <json_string>*
<json_number> ::= <json_number>+ e <json_number>.
<json_number> ::= + | - | . | /[0-9]/ | E | e

```

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

```

(start) ::= <json_raw>
<json_raw> ::= " <json_string> " | [ <json_list> ] | { <json_dict> }
| <json_number> | true | false | null
<json_string> ::= (space) | ! | # | $ | % | & | '
| * | + | - | , | . | / | : | ;
| < | = | > | ? | @ | [ | ] | ^ | _ | , | ' |
| { | | | } | ~ | /[A-Za-z0-9]/ | \ <decode_escape>
<decode_escape> ::= " | / | b | f | n | r | t
<json_list> ::= ]
| <json_raw> (, <json_raw> )+ ]
| (, <json_raw> )+ (, <json_raw> )+
<json_dict> ::= ]
| ( " <json_string> : <json_raw> , )
| " <json_string> : <json_raw> }
<json_string> ::= <json_string>* " ; DROP TABLE students"
<json_number> ::= <json_number>+ e <json_number>.
<json_number> ::= + | - | . | /[0-9]/ | E | e

```

Fuzzer ➡

```

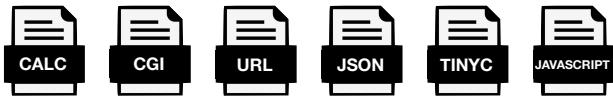
{ ""; " "; DROP TABLE STUDENTS" , "/h?0": [ ] , """: "" , "x": false ,
": null }
{ "": ".qF" , "": "'"; DROP TABLE STUDENTS" , """: 47 }
{ "7": { "y": "" } , "": false , "X": "N7!" , "": [ true ] , "": [ ]
}
{ "": [ ] , "9z6]l": null }
{ "#": false , "D": { "": true } , "t": 90 , "g": [ " "; DROP TABLE
STUDENTS" ] , "": [ false ] , "R5": [ ] , "": "'"; DROP TABLE STUDENTS",
"l": { "": "?l" , "E": null , "": [ 70.3076998940e6 ] , "Ju": true } }
{ "": true , "": "%Y" , "": false , "": true , "": [ "": [ ] , "": -096860E+0 , "U": 0E-5 } }
{ "": "ia": [ true , " "; DROP TABLE STUDENTS" , null , [ false , { } ] ,
true ] }
{ "": "@neB1T": 0.0 , "": null , "": true , "": 208.00E4 , "": true , "": 70e+10 ,
"": " "; "5zJ": [ false , false ] }
{ "": "H" , "": "d" , "": "'"; DROP TABLE STUDENTS" }
{ "": "Y!Z": ".i" , "": "h" , "": "'"; DROP TABLE STUDENTS" }
{ "": 64.00E-06 , "": "p[f": false , "": " "; DROP TABLE STUDENTS",
"m": [ ] , "": true , "": 8D4.0 , "": "QR": true } ]
{ "": " "; " "; DROP TABLE STUDENTS" }
{ "": "T" , "": "'"; DROP TABLE STUDENTS" , "": "zJzjT": 6.59 }
{ "": "oh": false }
{ "": "c": [ false , 304e+008520 , null , false , " "; DROP TABLE STUDENTS",
"m[MD" , [ false ] ] }

```

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.

## 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

Rahul Gopinath, Björn Mahr, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. ESEC/FSE 2020.

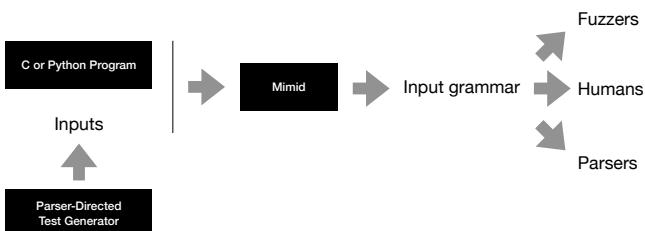
The grammars extracted by Mimid are accurate as producers as well as as parsers.

## 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.

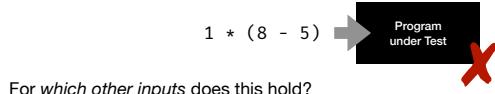
## Learning the Language

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

## Learning the Language of Failure

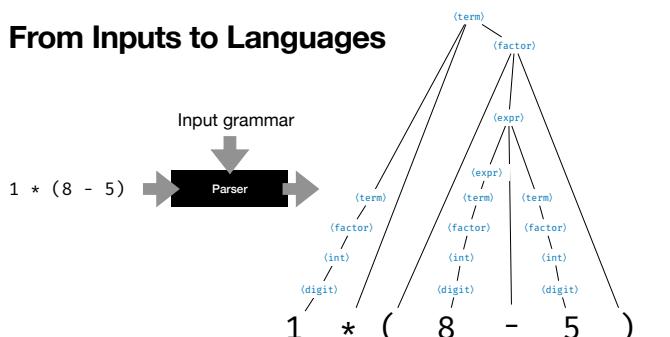
And now for the main point.

### Circumstances of Failure



We have seen how single inputs cause failures. But are these the only inputs?

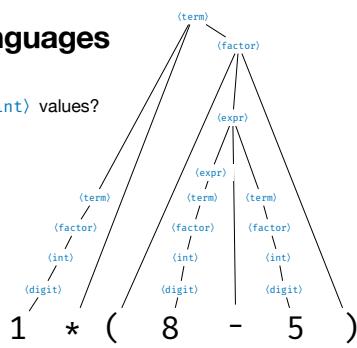
### From Inputs to Languages



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.

## From Inputs to Languages

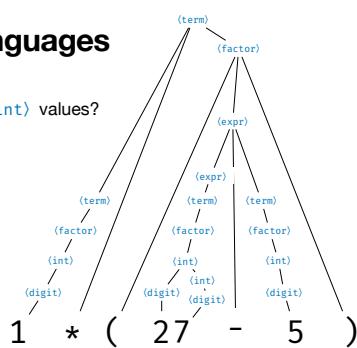
Does the failure occur for other `<int>` values?



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

## From Inputs to Languages

Does the failure occur for other `<int>` values?



... we replace parts of the parse tree (8) by newly generated alternatives (27).

## Patterns of Failure

$1 * (27 - 5)$  → Program under Test

Rahul Gopinath, Alexander Kampmann, Nikolas Henkler, Etsukai Sremelun, and Andreas Zeller.  
Abstracting Failure-Inducing Inputs. ISSTA 2020. ACM SIGSOFT Distinguished Paper Award.

and find that this one fails as well.

## Patterns of Failure

$1 * (\text{int} - 5)$  → Program under Test

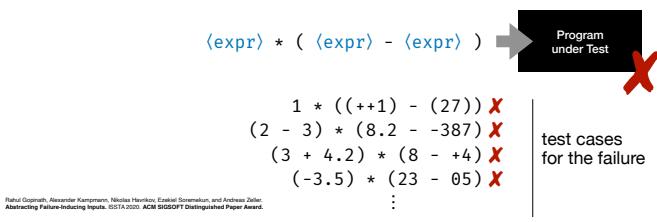
$1 * (8 - 5) \times$   
 $1 * (27 - 5) \times$   
 $1 * (3 - 5) \times$   
 $1 * (205 - 5) \times$

Rahul Gopinath, Alexander Kampmann, Nikolas Henkler, Etsukai Sremelun, and Andreas Zeller.  
Abstracting Failure-Inducing Inputs. ISSTA 2020. ACM SIGSOFT Distinguished Paper Award.

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.

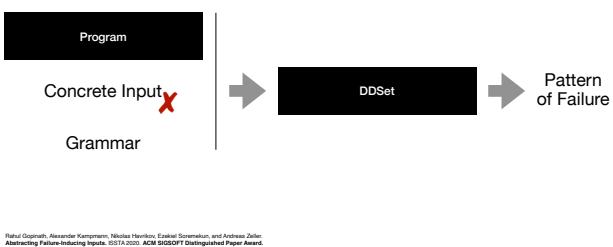
## Patterns of Failure

“The error occurs whenever \* is used in conjunction with –“



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.

## DDSet



Our tool DDSet takes a program, a failing input, and a grammar, and produces such a pattern of failure.

## DDSet: Evaluation



- 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*

In our evaluation, this works really well.

## Input Features

- Failure could also occur for *other inputs* – how about / or + ?
- Failure could depend on *non-structural features* like length, value, etc.

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

## Input Features

```

 $\exists \langle \text{term} \rangle * \langle \text{factor} \rangle$ 
 $\exists \langle \text{term} \rangle - \langle \text{expr} \rangle$ 
 $\exists (\langle \text{expr} \rangle)$ 
 $\exists 1 \exists 5 \exists 8$ 
 $\text{len}(\langle \text{int} \rangle) = 1$ 
 $\text{max}(\langle \text{int} \rangle) = 8$ 
 $\text{min}(\langle \text{int} \rangle) = 1$ 
 $\text{len}(\langle \text{start} \rangle) = 7$ 
:

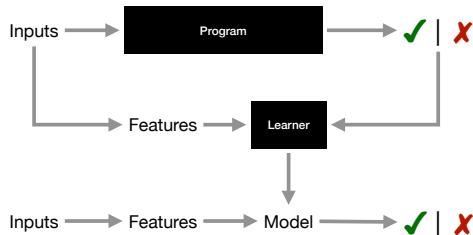
```



Which of these features *correlate* with failure?

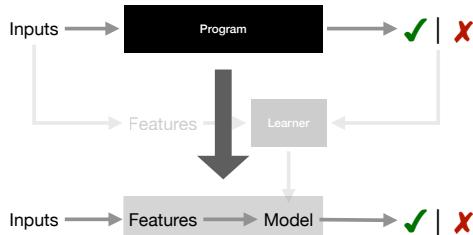
We introduce a number of input features, including existence, length, and maximum and minimum values of specific input elements.

## Learning Failure Models



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

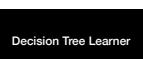
## Learning Failure Models



Actually, the produced model serves as a **model of the program** as it comes to failures or non-failures.

## Training a Classifier

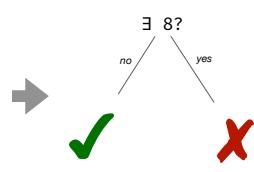
Labeled Inputs	Features
$1 * (8 - 5) \times$	$\exists 1 \exists 5 \exists 8$ $\text{len}(\langle \text{int} \rangle) = 1$ $\dots$
$27 + 3 \checkmark$	$\exists 2 \exists 3 \exists 7$ $\text{max}(\langle \text{int} \rangle) = 27$ $\dots$
$-1 * 23 + 4 \checkmark$	$\exists 1 \exists 2 \exists 3 \exists 4$ $\text{min}(\langle \text{int} \rangle) = -1$ $\dots$
:	



In our experiments, we use decision tree learners as their results are easy to understand.

## Training a Classifier

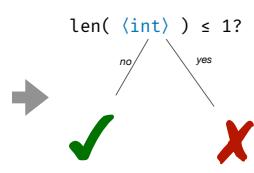
Labeled Inputs	Features
$1 * (8 - 5)$ ✗	$\exists 1 \ \exists 5 \ \exists 8$ <code>len( &lt;int&gt; ) = 1</code> ...
$27 + 3$ ✓	$\exists 2 \ \exists 3 \ \exists 7$ <code>max( &lt;int&gt; ) = 27</code> ...
$-1 * 23 + 4$ ✓	$\exists 1 \ \exists 2 \ \exists 3 \ \exists 4$ <code>min( &lt;int&gt; ) = -1</code>



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.

## Training a Classifier

Labeled Inputs	Features
$1 * (8 - 5)$ ✗	$\exists 1 \ \exists 5 \ \exists 8$ $\text{len}(\langle \text{int} \rangle) = 1$ ...
$27 + 3$ ✓	$\exists 2 \ \exists 3 \ \exists 7$ $\text{max}(\langle \text{int} \rangle) = 27$ ...
$-1 * 23 + 4$ ✓	$\exists 1 \ \exists 2 \ \exists 3 \ \exists 4$ $\text{min}(\langle \text{int} \rangle) = -1$



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

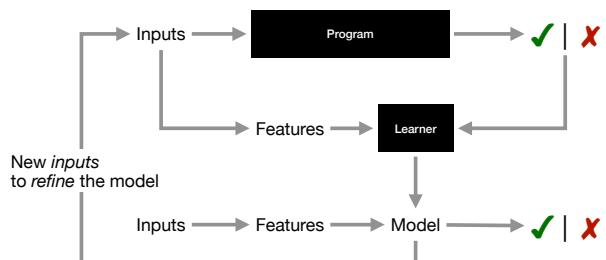
## Training a Classifier

Labeled Inputs	Features
$1 * (8 - 5)$ ✗	$\exists 1 \ \exists 5 \ \exists 8$ <code>len( &lt;int&gt; ) = 1</code>
$27 + 3$ ✓	$\exists 2 \ \exists 3 \ \exists 7$ <code>max( &lt;int&gt; ) = 27</code>
$-1 * 23 + 4$ ✓	$\exists 1 \ \exists 2 \ \exists 3 \ \exists 4$ <code>min( &lt;int&gt; ) = -1</code>

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

What we need is more inputs and more observations to come up with a more precise model.

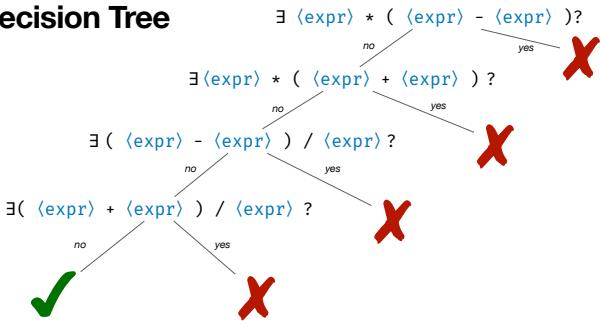
# Refining Models



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



## Decision Tree



Alexander Kampmann, Nikolas Henvickov, Ezeikel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

If we repeat this a number of times, we end up with this decision tree which now accurately characterizes the circumstances of failure.

## The Failure Circumstances

"The program fails when the distributive law can be applied"

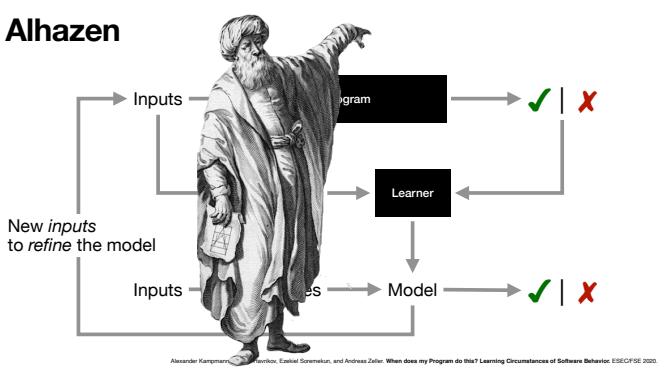


Can be used as explanation, as producer, as predictor

Alexander Kampmann, Nikolas Henvickov, Ezeikel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

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

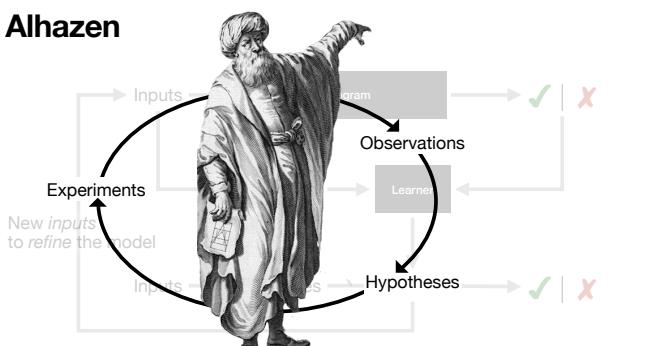
## Alhazen



Alexander Kampmann, Nikolas Henvickov, Ezeikel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

We named our approach **Alhazen**, after Ḥasan ibn al-Haytham (Latinized as Alhazen /æl'hæzən/; full name Abū 'Alī al-Ḥasan ibn al-Ḥasan ibn al-Haytham; أبو علي، الحسن بن الحسن بن الهيثم; c.965 – c.1040) – an Arab mathematician, astronomer, and physicist of the Islamic Golden Age.

## Alhazen



Alexander Kampmann, Nikolas Henvickov, Ezeikel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

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

Program

Inputs

Inputs

Grammar



Alhazen

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

## Alhazen

Experiments

Observations

Alhazen

Inputs

Model

|

Hypotheses

By abstracting over observations, and gradually refining hypothesis through experiments, Alhazen produces a predictive (and generative) model on whether failures occur or not.

## Alhazen

Program

Fuzzer

Grammar Miner

Alhazen

Model

Since the passing and failing inputs can come from a fuzzer, and since the grammar can come from a miner, ...

## Alhazen

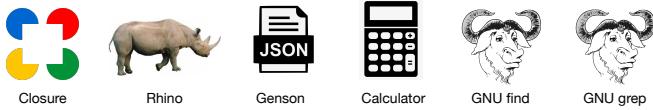
Program

Alhazen

Model

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

## Alhazen: Evaluation



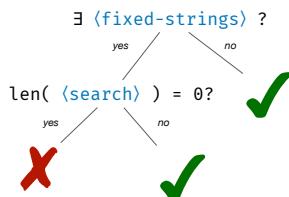
- As a *predictor*, Alhazen models classify 92% of all inputs correctly
- As a *producer*, ~68.5% of produced inputs correctly cause failures
- On average, decision trees refer to less than 5% of all input elements

Alexander Kampmann, Nikolas Henkler, Ezekiel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

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.

## Grep Crash

"grep crashes when --fixed-strings is used together with an empty search string"

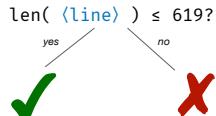


Alexander Kampmann, Nikolas Henkler, Ezekiel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

Here is an example. Alhazen correctly determines the circumstances of a grep crash.

## Nethack Crash

"NetHack crashes when a line in the config file has more than 619 characters"



Alexander Kampmann, Nikolas Henkler, Ezekiel Sonnenburg, and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. ESEC/FSE 2020.

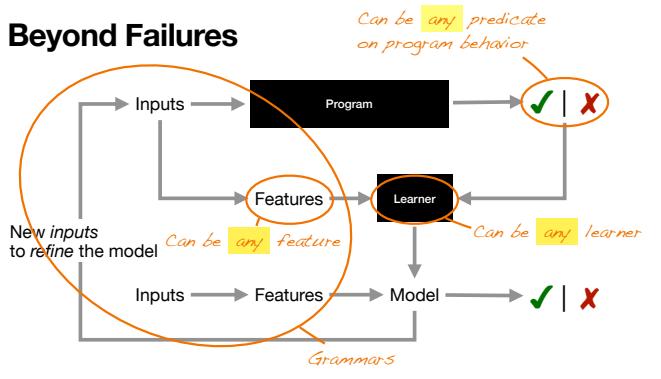
Since my time as a PhD student, I always wanted to have a slide with NetHack on it. This is how Alhazen explains the circumstances of a NetHack crash.

## 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

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

## Beyond Failures



One exciting thing about our approach is that it can generalize in many ways. For instance, one can use other learners besides decision tree learners.

## Learning the Language of Failure

But one can use other predicates too.

## Learning the Language of Acceptance

We can learn the set of inputs accepted by a program.

## Learning the Language of Coverage

Or the inputs that cover a particular location.

## **Learning the Language of Data Leaks**

The set of inputs that cause a particular unwanted behavior.

## **Learning the Language of Exploits**

Possibly even the exact language under which an exploit takes place.

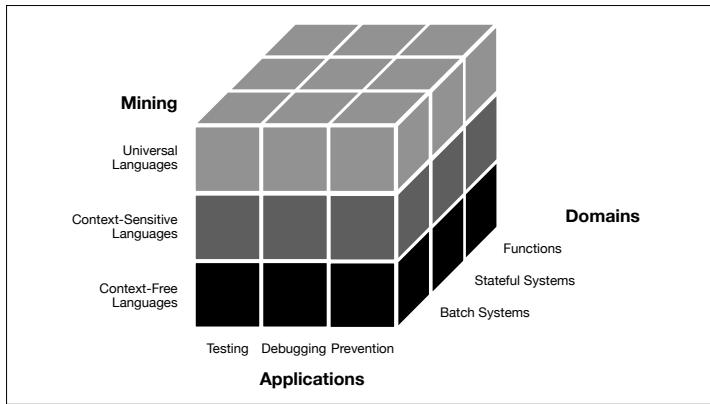
## **Learning the Language of Security**

Or, the complement: the language for which nothing bad happens. (There is a lots of future work in that one.)

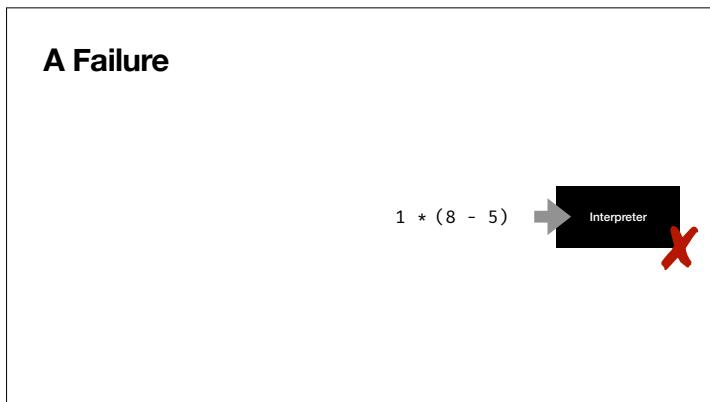


## **Perspectives**

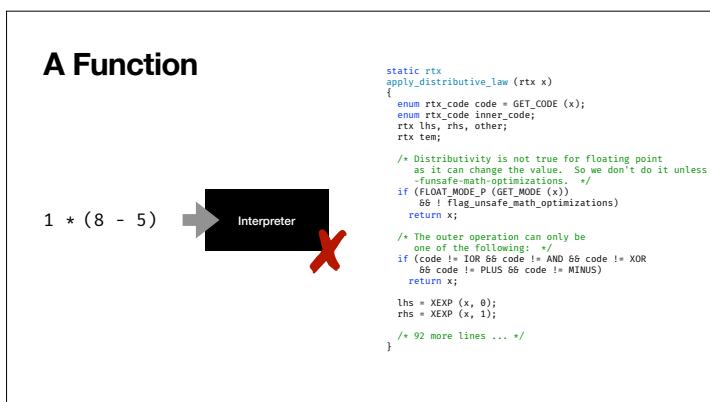
So, where are we going from here?



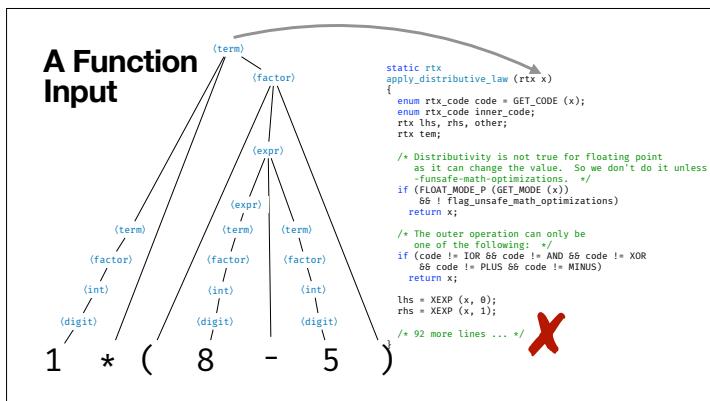
There are many ways languages, applications, and domains, can be combined.



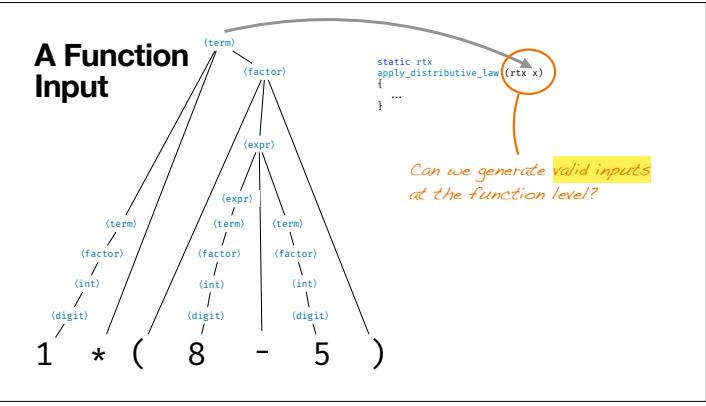
A domain I have not talked about yet is actual **code**.



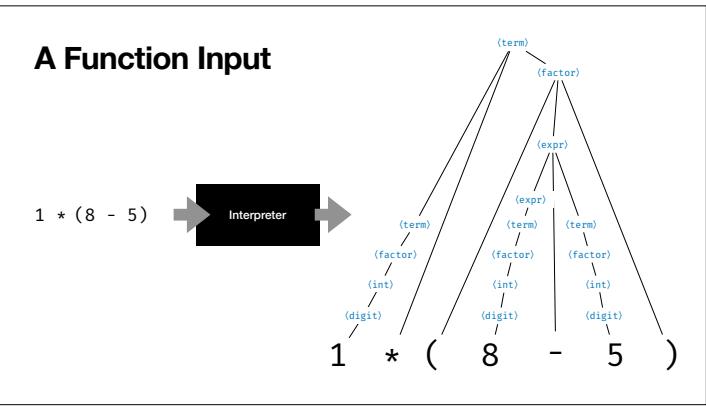
Because somewhere in your program is a piece of code that is faulty and fails. Here we have a function from GCC that once was faulty.



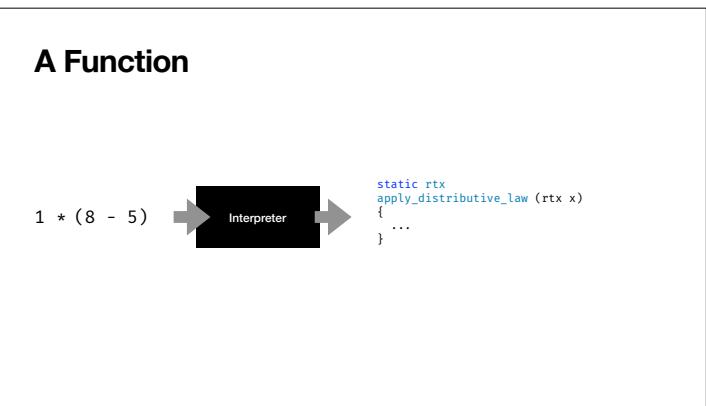
In our case, the input to this function actually is a parse tree.



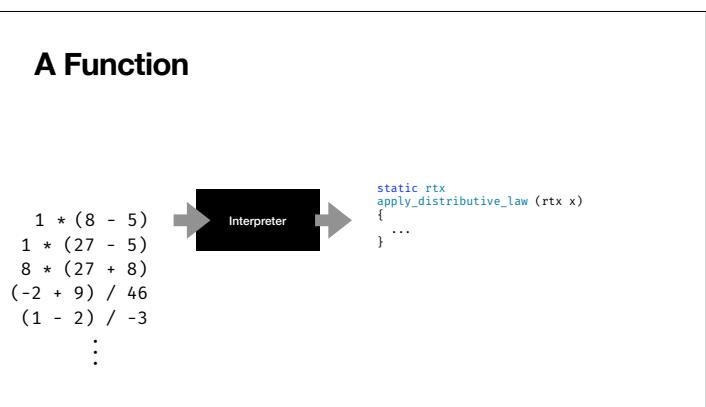
The question is: Can we test `apply_distributive_law()` by supplying it with valid values for `x`? If you choose a function-level test generator, you will feed plenty of invalid parse trees into this function. This will not test well.



At the system level, however, this is easy, as we can feed valid **system inputs** into our interpreter/compiler.



We can do so for one input...



... and for as many as we like, each one producing a valid value for `x`.

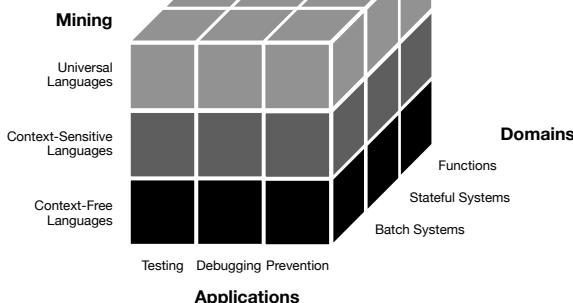
## A Function

```
static rtx  
apply_distributive_law (rtx x)  
{  
    ...  
}
```

What is the language of x?

But if we want to test the function in isolation, or reason about how it works, we need to know the set of valid values for x. This is a type and a language. How can we learn this? This is an open question.

## Perspectives



## Teamwork

Andreas Zeller and Dorothea Lütkehaus. *DDD—a free graphical front-end for UNIX debuggers*. SIGPLAN Notices 1996.  
Andreas Zeller and Ralf Hildebrandt. *Simplifying and Isolating Failure-Inducing Input*. TSE 2002.  
Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. *Mining Version Histories to Guide Software Changes*. TSE 2005.  
Stephan Neuhaus, Thomas Zimmermann, Christian Holer, and Andreas Zeller. *Predicting vulnerable software components*. CCS 2007.  
Christian Holer, Kim Herzig, and Andreas Zeller. *Fuzzing with Code Fragments*. USENIX 2012.  
Nikolas Havrikov and Andreas Zeller. *Systematically Covering Input Structure*. ASE 2019.  
Björn Mathis, Rahul Gopinath, Michael Mera, Alexander Kampmann, and Andreas Zeller. *Parser-Directed Fuzzing*. PLDI 2019.  
Rahul Gopinath, Björn Mathis, and Andreas Zeller. *Mining Input Grammars from Dynamic Control Flow*. ESEC/FSE 2020.  
Björn Mathis, Rahul Gopinath, and Andreas Zeller. *Learning Input Tokens for Effective Fuzzing*. ISSTA 2020.  
Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. *Abstracting Failure-Inducing Inputs*. ISSTA 2020.  
Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. *When does my Program do this? Learning Circumstances of Software Behavior*. ESEC/FSE 2020.

Here is a list of all the papers that went into this talk.



## Teamwork

Thomas Zimmermann Dorothea Lütkehaus Peter Weissgerber Stephan Diehl Stephan Neuhaus Christian Holer Ralf Hildebrandt  
Nikolas Havrikov Kim Herzig Rahul Gopinath Björn Mathis Michaël Mera Alexander Kampmann Ezekiel Soremekun  
Nataniel Pereira Borges Junior Rafael Dutra Konstantin Kuznetsov Jenny Rau Sascha Just Matthias Höschel Andreas Rau  
Clemens Hammacher Kevin Streit Konrad Jamrozik Alessio Gambi Vitalii Avdiienko María Gómez Lacruz Alessandra Gorla  
Sudipta Chattopadhyay Andrey Tarasevich Juan Pablo Galeotti Gordon Fraser Ilaria Tavecchia Florian Gross Eva May Marcel Böhme  
Valentín Dallmeier Bernd Pohl Michael Mirold Christian Lindig Silvia Breu Stephan Neuhaus Martin Burger Frank Padberg  
Rahul Premraj Yana Mileva Matthias Schur David Schuler Jeremias Rößler Andrzej Wasylkowski Irina Brudaru Holger Cleve

And these are my all of my students and post-docs over the years. My work became possible only through them. Thank you!

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!