Datalog for Program Analysis

Meng Xu *(University of Waterloo)*

March 20, 2023

Outline

Why this topic?

A significant portion of software security research is based on the following observation:

> *If the program contains some specific code pattern, that program is more likely to be vulnerable.*
>
> - *e.g., `strcpy` taking a user-supplied `src` argument*

Why this topic?

A significant portion of software security research is based on the following observation:

> *If the program contains some specific code pattern, that program is more likely to be vulnerable.*
>
> - *e.g., `strcpy` taking a user-supplied `src` argument*

Q: How do we even precisely define and express this code pattern?

Why this topic?

A significant portion of software security research is based on the following observation:

> *If the program contains some specific code pattern, that program is more likely to be vulnerable.*
>
> - *e.g., `strcpy` taking a user-supplied `src` argument*

Q: How do we even precisely define and express this code pattern?
And more questions to follow:

- e.g., compare with another code pattern?
- e.g., inter-op / composite with code patterns?
- e.g., scale to more codebases?
- e.g., argue for soundness / completeness?

Programming paradigm: imperative vs declarative

Programming paradigm: imperative vs declarative

Imperative programming is a paradigm describing HOW the program should do something by explicitly specifying each instruction (or state transition) step by step.

Declarative programming is a paradigm describing WHAT the program knows and does, without explicitly specifying its algorithm.

Baking a chocolate cake

The imperative way

1. mix flour, sugar, cocoa powder, baking soda, and salt
2. add milk, vegetable oil, eggs, and vanilla to form the batter
3. preheat the oven at 180°C
4. put the batter in a cake pan and bake for 30 minutes

Baking a chocolate cake

The imperative way

1. mix flour, sugar, cocoa powder, baking soda, and salt
2. add milk, vegetable oil, eggs, and vanilla to form the batter
3. preheat the oven at 180°C
4. put the batter in a cake pan and bake for 30 minutes

The declarative way

- cake = batter + 180°C oven + 30 minutes backing
- batter = solid ingredients + liquid ingredients
- solid ingredients = flour, sugar, cocoa powder, baking soda, and salt
- fluid ingredients = milk, vegetable oil, eggs, and vanilla

Finding a vulnerability

The imperative way

1. for each function in the program, search for a `strcpy` call in the function body
2. trace back how the `src` argument in the `strcpy` call is derived (via def-use analysis)
3. for any ancestor in the trace, if it comes from untrusted user-controlled input, mark the `strcpy` call as vulnerable

The declarative way

- program = [function]
- function = [instruction] (per each function)
- *defines*(var, instruction)
- *uses*(instruction, var)
- *is_user_controlled*(var)
- *is_strcpy_vuln* =
  `strcpy(..., src)`
  + *defines*(src, i_src)
  + *uses*(i_src, x)
  + *defines*(x, i_x)
  + *uses*(i_x, var)
  + *is_user_controlled*(var)

Outline

Datalog overview

Datalog programming is based on rules and facts.

Datalog overview

Datalog programming is based on rules and facts.

For example

- **Fact**: Vancouver is rainy
- **Fact**: Waterloo is rainy
- **Fact**: Waterloo is cold
- **Rule**: If a city is both rainy and cold, then it is snowy

Query: which city is snowy?

Datalog overview

Datalog programming is based on rules and facts.

For example

- **Fact**: Vancouver is rainy
- **Fact**: Waterloo is rainy
- **Fact**: Waterloo is cold
- **Rule**: If a city is both rainy and cold, then it is snowy

Query: which city is snowy?

Encoded as Souffle rules

```
1 .decl rainy(city: symbol)
2 .decl cold(city: symbol)
3 .decl snowy(city: symbol)
4 .output snowy
5
6 rainy("Vancouver").
7 rainy("Waterloo").
8 cold("Waterloo").
9 snowy(city) :- rainy(city), cold(city).
```

Predicates

Predicates are essentially *parameterized propositions*, which are also called atoms. These are building blocks of any Datalog program.

Examples:

- `rainy(x)`, `cold(x)`, `snowy(x)`: city `x` is rainy, cold, and snowy.
- `canadianFood(x)`: `x` is iconic Canadian food (e.g., Tim Hortons).

Predicates

Predicates are essentially *parameterized propositions*, which are also called atoms. These are building blocks of any Datalog program.

Examples:

- `rainy(x)`, `cold(x)`, `snowy(x)`: city `x` is rainy, cold, and snowy.
- `canadianFood(x)`: `x` is iconic Canadian food (e.g., Tim Hortons).

In the above cases, predicates are used to describe attributes of one entity. Predicates can also be used to describe relations between multiple entities, such as.

- `parent(x, y)`: `x` is a parent of `y`
- `square(x, y)`: `y` is the square of `x`
- `xor(x, y, z)`: the `xor` of `x` and `y` is `z`

Recursive rules

The real power of Datalog is on its expressiveness of (mutually) recursively defined relations.

Consider the encoding of a control-flow graph (CFG):

```
1 .decl edge(b1, b2)
2 .input edge
3
4 .decl reachable(b1, b2)
5 reachable(b1, b2) :- edge(b1, b2).
6 reachable(b1, b2) :- edge(b1, b3), reachable(b3, b2).
7
8 .decl more_than_one_hop(b1, b2)
9 more_than_one_hop(b1, b2) :- reachable(b1, b2), !edge(b1, b2).
```

Q: How to interpret these rules (line 5, 6, and 9)?

Outline

Overview

In this section, we will implement several dataflow analysis in Datalog (Souffle to be specific).

We start by modeling the program execution flow in Datalog, based on which we then define the declarative rules for typical dataflow problems such as reaching definition, available expression, etc.

CFG representation

Q: How to encode a sequential program in Datalog?

CFG representation

Q: How to encode a sequential program in Datalog?

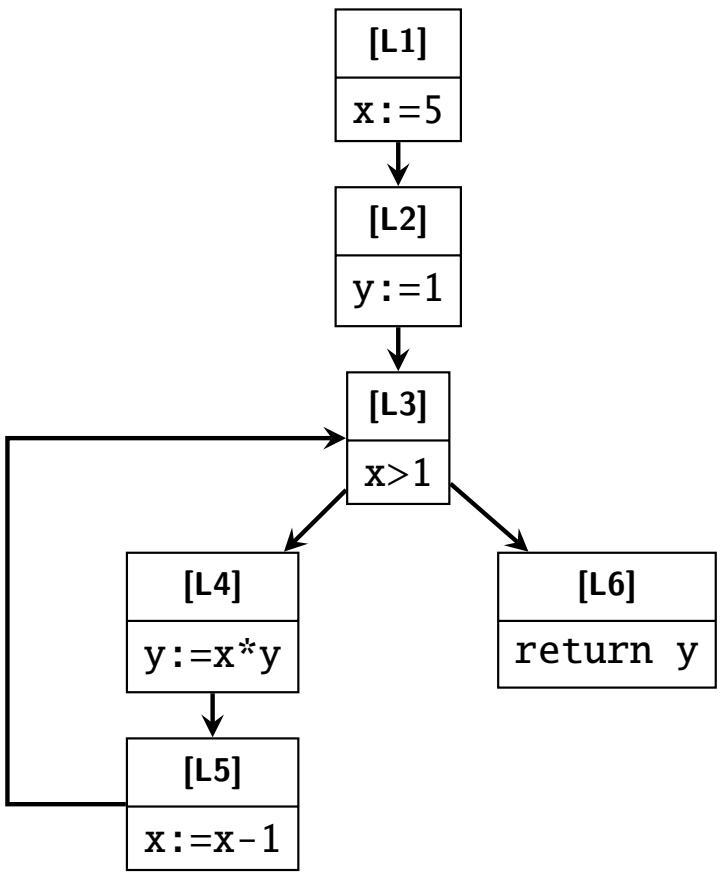
```
1  .type Label <: number
2
3  // control flow from l1 to l2
4  .decl flow(l1: Label, l2: Label)
5
6  // l is the start of the execution
7  .decl init_label(l: Label)
8
9  // l is the end of the execution
10 .decl exit_label(l: Label)
```

CFG representation example

$[\texttt{x:=5}]^1;[\texttt{y:=1}]^2;\ \texttt{while}\ [\texttt{x>1}]^3\ \texttt{do}\ ([\texttt{y:=x*y}]^4;[\texttt{x:=x-1}]^5;)\ [\texttt{return y}]^6;$

CFG representation example

`[x:=5]`^{1}`;[y:=1]`^{2}`; while [x>1]`^{3}` do ([y:=x*y]`^{4}`;[x:=x-1]`^{5}`;) [return y]`^{6}`;`



flow.facts

1	2
2	3
3	4
3	6
4	5
5	3

init_label.facts

1

exit_label.facts

6

Instruction encoding

Q: How to encode the semantics of each instruction?

Instruction encoding

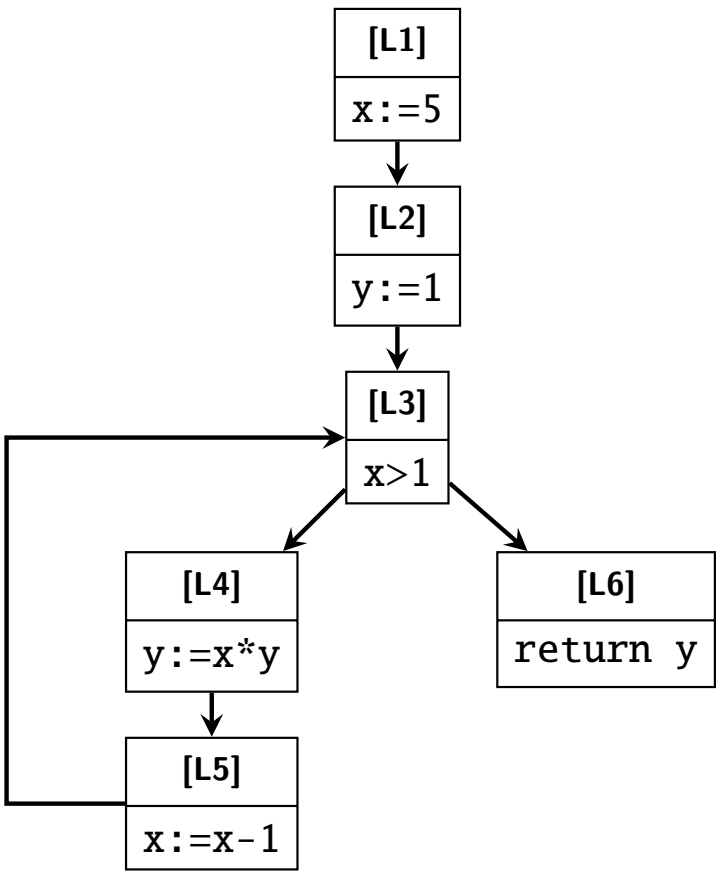
Q: How to encode the semantics of each instruction?

One way to look at instructions is that they (optionally) use variables to (optionally) define variable. It is only a partial semantic view of instructions, but is sufficient for we are about to define next.

```
1 .type Var <: symbol
2
3 // instruction l defines var v
4 .decl def(l: Label, v: Var)
5
6 // instruction l uses var v
7 .decl use(l: Label, v: Var)
```

Def-use example

`[x:=5]`^{1}`;[y:=1]`^{2}`; while [x>1]`^{3} `do ([y:=x*y]`^{4}`;[x:=x-1]`^{5}`;) [return y]`^{6};



def.facts

1	x
2	y
4	y
5	x

use.facts

3	x
4	x
4	y
5	x
6	y

Reaching definition analysis

Recall the semantics of reaching definition analysis: it determines, at each point, what definitions can reach there.

Q: How to encode the reaching definition relation in Datalog?

Reaching definition analysis

```
1  // var v defined at label def can reach *before* instruction l
2  .decl rd_entry(l: Label, v: Var, def: Label)
3
4  // var v defined at label def can reach *after* instruction l
5  .decl rd_exit(l: Label, v: Var, def: Label)
6
7  // rule 1: def of v can reach the end of def
8  rd_exit(l, v, l) :- def(l, v).
9
10 // rule 2: def of v can reach the end of l if l does not define v
11 rd_exit(l, v, def) :- rd_entry(l, v, def), !def(l, v).
12
13 // rule 3: def of v can reach next instruction
14 rd_entry(l, v, def) :- rd_exit(prev_l, v, def), flow(prev_l, l).
```

Reaching definition analysis

```
--------------
rd_entry
l       v       def
==============
2       x       1
3       x       1
3       x       5
3       y       2
3       y       4
4       x       1
4       x       5
4       y       2
4       y       4
5       x       1
5       x       5
5       y       4
6       x       1
6       x       5
6       y       2
6       y       4
==============
```

```
--------------
rd_exit
l       v       def
==============
1       x       1
2       x       1
2       y       2
3       x       1
3       x       5
3       y       2
3       y       4
4       x       1
4       x       5
4       y       4
5       x       5
5       y       4
6       x       1
6       x       5
6       y       2
6       y       4
==============
```

Liveness analysis

Recall the semantics of liveness analysis: given a variable v and a code location l, it determines whether v will be used (and before being re-defined by other instructions) in any program path starting from l.

Q: How to encode the liveness relation in Datalog?

Live variable analysis

```
1  // var v defined at label def is alive *before* instruction l
2  .decl lv_entry(l: Label, v: Var, def: Label)
3
4  // var v defined at label def is alive *after* instruction l
5  .decl lv_exit(l: Label, v: Var, def: Label)
6
7  // rule 1: use of v make v alive before the use
8  lv_entry(l, v, l) :- use(l, v).
9
10 // rule 2: use of v can reach the entry of l if l does not define v
11 lv_entry(l, v, def) :- lv_exit(l, v, def), !def(l, v).
12
13 // rule 3: def of v can reach next instruction
14 lv_exit(l, v, def) :- lv_entry(next_l, v, def), flow(l, next_l).
```

Outline

A new trend: Datalog in smart contract auditing

Recent years have observed a new trend in applying Datalog-style tooling in finding security vulnerabilities in smart contracts.

Sample projects include:

- Gigahorse
- Vandle
- Securify 2.0

Basis of analysis

```
1  .type Statement
2  .type Variable
3  .type Opcode
4  .type Value
5
6  .decl entry(s:Statement)
7  .decl edge(h:Statement, t:Statement)
8  .decl def(var:Variable, stmt:Statement)
9  .decl use(var:Variable, stmt:Statement)
10 .decl op(stmt:Statement, op:Opcode)
11 .decl value(var:Variable, val:Value)
12
13 .input entry, edge, def, use, op, value
```

Sample checkers

```
1 .decl uncheckedCall(u:Statement)
2
3 uncheckedCall(u) :-
4     callResult(_, u),
5     !checkedCallThrows(u),
6     !checkedCallStateUpdate(u).
```

```
1 .decl reentrantCall(stmt:Statement)
2
3 reentrantCall(stmt) :-
4     op(stmt, "CALL"),
5     !protectedByLoc(stmt, _),
6     gassy(stmt, gasVar),
7     op_CALL(stmt, gasVar, _, _, _, _, _, _).
```

Sample checkers

```
.decl unsecuredValueSend(stmt:Statement)

unsecuredValueSend(stmt) :-
    op_CALL(stmt, _, target, val, _, _, _, _),
    nonConstManipulable(target),
    def(val, _),
    !value(val, "0x0")
    !fromCallValue(val),
    !inaccessible(stmt).
```

```
.decl originUsed(stmt:Statement)

originUsed(stmt) :-
    op(stmt, "ORIGIN"),
    def(originVar, stmt),
    depends(useVar, originVar),
    usedInStateOrCond(useVar, _).
```

Other deployments

Datalog has also been widely used in other program analysis areas, including

- DOOP points-to analysis (for Java)
- cclyzer++ points-to analysis (for LLVM)
- DDisasm disassembler

Other deployments

Datalog has also been widely used in other program analysis areas, including

- DOOP points-to analysis (for Java)
- cclyzer++ points-to analysis (for LLVM)
- DDisasm disassembler

It is not yet heavily adopted in software security / bug finding yet, but seems to be a very promising candidate.

Outline

Background

Rego is the policy language behind Open Policy Agent (OPA), which is a policy engine supported by all major cloud providers (Azure, AWS, Google Cloud).

Background

Rego is the policy language behind Open Policy Agent (OPA), which is a policy engine supported by all major cloud providers (Azure, AWS, Google Cloud).

```
1  package application.authz
2
3  import future.keywords
4
5  # Everyone can see pets up for adoption
6  allowed_pets contains pet if {
7      some pet in input.pet_list
8      pet.up_for_adoption
9  }
10
11 # Employees can see all pets.
12 allowed_pets contains pet if {
13     [_, payload, _] := io.jwt.decode(input.token)
14     payload.employee
15     some pet in input.pet_list
16 }
```

Semantics? what semantics?

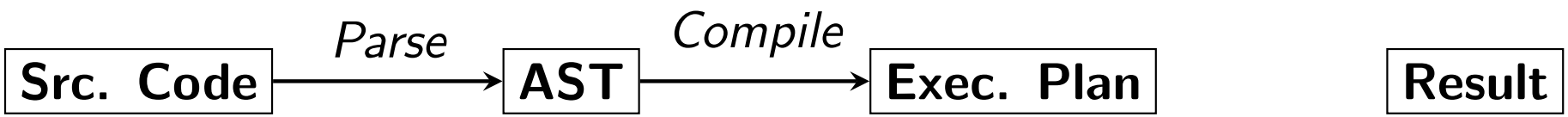
Semantics? what semantics?

Src. Code

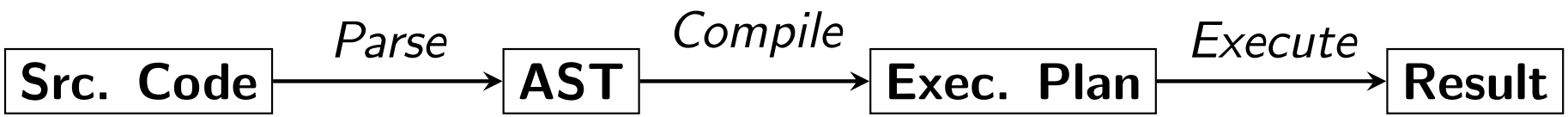
Result

Semantics? what semantics?

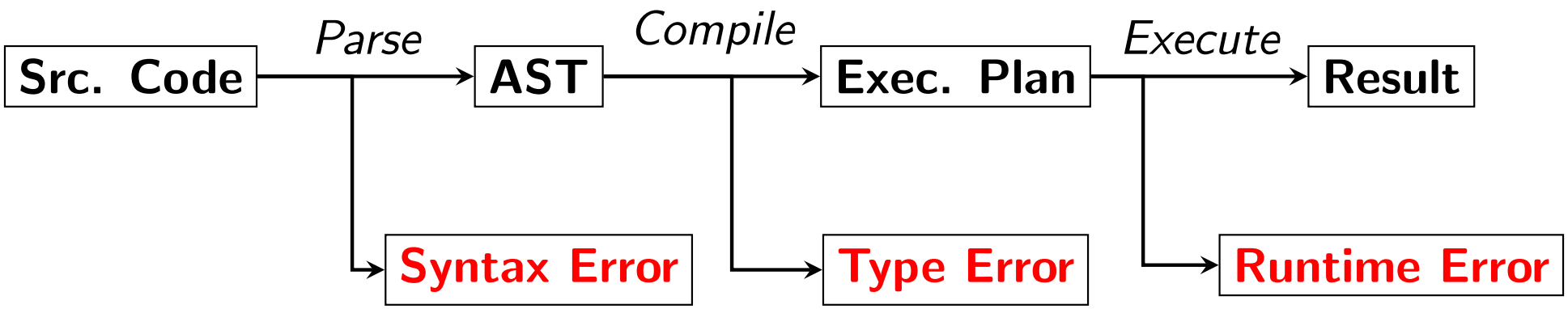
Semantics? what semantics?



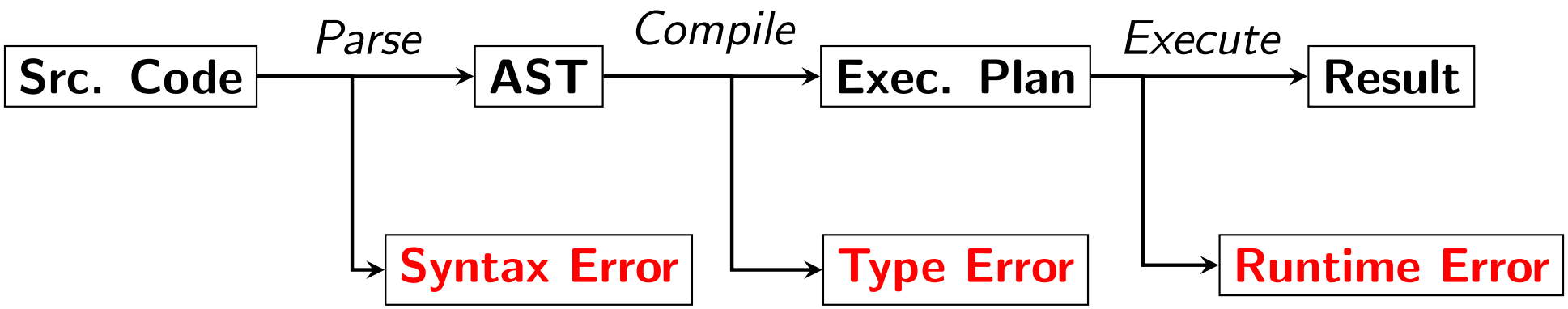
Semantics? what semantics?



Semantics? what semantics?



Semantics? what semantics?



A completely defined language semantics should, given a specific text of source code, unambiguously decide:

- A unique AST *OR* a specific syntax error
- A unique execution plan *OR* a specific type error
- An exhaustive list of runtime errors that can be triggered

Approach

Step 1: decompose AST into relations

```
1  .type term_id <: symbol
2
3  // term: literal
4  .decl def_term_null(id: term_id)
5  .decl def_term_bool(id: term_id, value: bool)
6  .decl def_term_real(id: term_id, value: number)
7  .decl def_term_text(id: term_id, value: string)
8
9  // term: composite
10 .decl def_term_array(id: term_id)
11 .decl set_term_array_element(id: term_id, index: number, element: term_id)
12 ......
13
14 // term: expression
15 .decl def_term_add(id: term_id, lhs: term_id, rhs: term_id)
16 .decl def_term_sub(id: term_id, lhs: term_id, rhs: term_id)
17 ......
```

Approach

Step 2: encode typing rules as Datalog rules

```
1  // unique def rule: each term is defined once and only once
2  .decl has_term_def_single(id: term_id)
3  has_term_def_single(id) :- def_term_null(id).
4  has_term_def_single(id) :- def_term_bool(id).
5  ......
6
7  .decl has_term_def_double(id: term_id)
8  has_term_def_double(id) :- def_term_null(id), def_term_bool(id).
9  // ... many more equality testing ...
10
11 .decl has_term(id: term_id)
12 has_term(id) :- has_term_def_single(id), !has_term_def_double(id).
13
14 // unique use rule: each term is used once and only once
15 use_term(id) :- has_term_use_single(id), !has_term_use_double(id).
16
17 // term typing rule
18 .decl term_checked(id: term_id)
19 term_checked(id) :- has_term(id), use_term(id), // other typing rules
```

Approach

Step 3: Differential testing (fuzz) the existing OPA interpreter

- Use SMT solver to construct test cases that
  - fail on every possible violation of each typing rule.
  - satisfies all typing rules.
- Send these test cases to the real-world OPA interpreter(s) and cross test their behaviors.

⟨ End ⟩