

Code Analysis with... Databases (!?)

Adrian Herrera

\$ whoami

Then



Australian Government

Department of Defence

Defence Science and
Technology Group

\$ whoami

Then



Australian Government

Department of Defence

Defence Science and
Technology Group

Now

**INTERRUPT
LABS**

Outline

- Why code analysis using databases?
- Datalog
 - Toy examples
- “Real-world” tools

Caveats

⚠ I am not an expert in any of this ⚠



Automated Code Auditing?



Silvio_Cesare_hacking.jpg

“Automated” Code Auditing



Manual analysis

- Human expertise/intuition 
- Doesn't scale 

Grep

- Widely known/available 
- Doesn't understand syntax, let alone semantics 

Weggli

- Search language = target language 
- Poor composition 

Weggli

Example query to find `memcpy` to stack variables

```
weggli '{  
  _ $buf[_];  
  memcpy($buf,_,_);  
}' ./target/src
```


Weggli

Example query to find `memcpy` to stack variables

```
weggli '{  
  _ $buf[_];  
  memcpy($buf,_,_);  
}' ./target/src
```

What if I want to compose queries?

What about across basic blocks?

What about across functions?

Can we do better?

Treat the program as a database!

Perform queries on “facts” stored in the DB +
infer additional “facts”



[adult swim]

Datalog

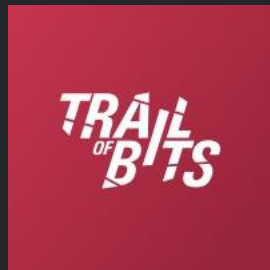
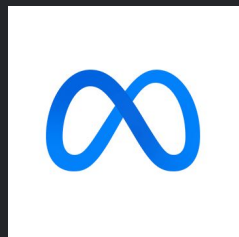
- Declarative logic programming language
- Subset of Prolog
- Bottom up (vs. top-down) evaluation
 - Not Turing complete (guaranteed termination)

High-level Approach

- Stores *facts*
- New facts can be deduced via *rules*
- Facts can be *queried*

Who Uses This?

- CodeQL
- Glean
- Parfait
- DDisasm



Datalog

```
.decl edge(x: number, y: number)
.input edge

.decl path(x: number, y: number)
.output path

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```


Datalog

```
.decl edge(x: number, y: number)
.input edge

.decl path(x: number, y: number)
.output path

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Datalog

```
.decl edge(x: number, y: number)
```

```
.input edge
```

```
.decl path(x: number, y: number)
```

```
.output path
```

```
path(x, y) :- edge(x, y).
```

```
path(x, y) :- path(x, z), edge(z, y).
```

Datalog

```
.decl edge(x: number, y: number)  
.input edge
```

```
.decl path(x: number, y: number)  
.output path
```

```
path(x, y) :- edge(x, y).  
path(x, y) :- path(x, z), edge(z, y).
```

Datalog

```
.decl edge(x: number, y: number)  
.input edge
```

```
.decl path(x: number, y: number)  
.output path
```

```
path(x, y) :- edge(x, y).  
path(x, y) :- path(x, z), edge(z, y).
```

Datalog

```
.decl edge(x: number, y: number)  
.input edge
```

```
.decl path(x: number, y: number)  
.output path
```

```
path(x, y) :- edge(x, y).
```

```
path(x, y) :- path(x, z), edge(z, y).
```

Datalog

```
.decl edge(x: number, y: number)  
.input edge
```

```
.decl path(x: number, y: number)  
.output path
```

```
path(x, y) :- edge(x, y).
```

```
path(x, y) :- path(x, z), edge(z, y).
```

Syntax

Where do input facts come from?

Comma separated values (CSV)

1	2
2	3

Syntax

Outputs are also CSV

1	2
2	3
1	3

Let's build some
code analyses

Steps

1. Extract syntactic facts
2. Derive facts
3. Perform queries

Steps

1. Extract syntactic facts
2. Derive facts
3. Query

1. Extract Syntactic Facts

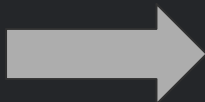
- Decide how to represent code
- *Ingest* the codebase into the database
- Language dependent
 - Cclyzer++ extracts facts from LLVM IR
 - Doop extracts facts from Java Bytecode
 - Ddisasm extracts facts from assembly code

1. Extract Syntactic Facts

```
struct S {  
    int *f;  
}  
  
S *v1 = h1();  
S *v2 = h2();  
v1 = v2;  
int *v3 = h3();  
v1.f = v3;  
int *v4 = v1.f;
```

1. Extract Syntactic Facts

```
struct S {  
    int *f;  
}  
  
S *v1 = h1();  
S *v2 = h2();  
v1 = v2;  
int *v3 = h3();  
v1.f = v3;  
int *v4 = v1.f;
```



```
.type var <: symbol  
.type obj <: symbol  
.type field <: symbol  
  
// -- inputs --  
.decl assign( a:var, b:var )  
.decl new( v:var, o:obj )  
.decl load( a:var, b:var, f:field )  
.decl store( a:var, f:field, b:var )  
  
// -- facts --  
assign("v1","v2").  
  
new("v1","h1").  
new("v2","h2").  
new("v3","h3").  
  
store("v1","f","v3").  
load("v4","v1","f").
```

Steps

1. Extract syntactic facts
2. Derive facts
3. Query

2. Derive new facts

```
// -- analysis --  
.decl alias( a:var, b:var ) output  
  
alias(X,X) :- assign(X,_).  
alias(X,X) :- assign(_,X).  
alias(X,Y) :- assign(X,Y).  
alias(X,Y) :- load(X,A,F), alias(A,B), store(B,F,Y).  
  
.decl pointsTo( a:var, o:obj )  
.output pointsTo  
pointsTo(X,Y) :- new(X,Y).  
pointsTo(X,Y) :- alias(X,Z), pointsTo(Z,Y).
```


Steps

1. Extract syntactic facts
2. Derive facts
3. Query

3. Query

Run the Datalog engine

```
-----  
pointsTo  
=====
```

v1	h1
v1	h2
v2	h2
v4	h3
v3	h3

```
struct S {  
    int *f;  
}  
  
S *v1 = h1();  
S *v2 = h2();  
v1 = v2;  
int *v3 = h3();  
v1.f = v3;  
int *v4 = v1.f;
```

“Real World” Analyses

Lots of cool tools available

- Ddisasm (Assembly)
- Cclyzer++ (LLVM)
- Doop (Java)
- Treedb (TreeSitter)

Treedb

- TreeSitter is an incremental parser
 - Works on broken code!
- Transforms code into Abstract Syntax Tree (AST)
- Treedb puts AST into a Datalog database to query



Treedb Example

Find all constant-value binary expressions:

```
.decl const_binop(expr: JavaBinaryExpression)

const_binop(expr) :-
  java_binary_expression(expr),
  java_binary_expression_left_f(expr, l),
  java_binary_expression_right_f(expr, r),
  java_decimal_integer_literal(l),
  java_decimal_integer_literal(r).

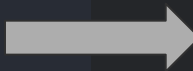
.decl show_const_binop(text: JavaNodeText)
show_const_binop(text) :-
  const_binop(expr),
  java_node_text(expr, text).
```

Treedb Example

```
.decl const_binop(expr: JavaBinaryExpression)
```

```
const_binop(expr) :-  
  java_binary_expression(expr),  
  java_binary_expression_left_f(expr, l),  
  java_binary_expression_right_f(expr, r),  
  java_decimal_integer_literal(l),  
  java_decimal_integer_literal(r).
```

```
.decl show_const_binop(text: JavaNodeText)  
show_const_binop(text) :-  
  const_binop(expr),  
  java_node_text(expr, text).
```



```
class Main {  
  public static void main(String[] args) {  
    int x = 2 + 2;  
  }  
}
```

CodeQL

CodeQL is a “Frankenstein Datalog” 

Query to find all redundant “if” blocks

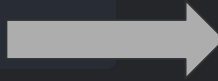
```
from IfStmt ifstmt, BlockStmt block

where ifstmt.getThen() = block and block.getNumStmt() = 0

select ifstmt, "This 'if' statement is redundant."
```

CodeQL

```
from IfStmt ifstmt, BlockStmt block
where ifstmt.getThen() = block and block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```



```
if (x == 10) {
} else {
    // Do stuff...
}
```


CodeQL

- Very powerful
- ... But very complex
- Free to use on open-source code

Summary

- Treating a codebase as a database is powerful
- Starting to become more prevalent (e.g., CodeQL)
- Not just static analysis!

```
// The "fuzzing frontier"  
.decl frontier(block: BasicBlock)  
frontier(pred_bb) :-  
    block_predecessors(bb, pred_bb),  
    block_cov(pred_bb, _),  
    !block_cov(bb, _).
```