CS 858: Software Security

Offensive and Defensive Approaches

Detection: declarative rules

Meng Xu *(University of Waterloo)*

Fall 2022

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 you 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 you even precisely define and express this code pattern?

- 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

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

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

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

```
.decl rainy(city: symbol)
.decl cold(city: symbol)
.decl snowy(city: symbol)
.output snowy

rainy("Vancouver").
rainy("Waterloo").
cold("Waterloo").
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`

Relations

When encoding relations among different entities, parameters in the predicates are not directional, i.e., relations are not functions that bear input-output semantics.

Relations

When encoding relations among different entities, parameters in the predicates are not directional, i.e., relations are not functions that bear input-output semantics.

For example, given:

- parent(Sam, Mike)
- parent(Sussan, Mike)
- parent(Don, Sam)
- parent(Rosy, Sam)

we can further define

- parentOfMike(x) :- parent(x, Mike)
  - who are the parents of Mike
- childrenOfSussan(c) :- parent(Sussan, c)
  - who are the children of Sussan

Horn clauses

A Horn clause has a head h, which is a predicate, and a body, which is a list of literals l_1, l_2, . . . , l_n, written as $h \leftarrow l_1, l_2, \ldots, l_n$.

- Each literal l_i is either a predicate or the negation of a predicate.
- This means "h is true when $l_1, l_2, \ldots, l_n$ are simultaneously true"
  - e.g., `snowy(city) :- rainy(city), cold(city).`

Horn clauses

A Horn clause has a head h, which is a predicate, and a body, which is a list of literals l_1, l_2, ..., l_n, written as $h \leftarrow l_1, l_2, \ldots, l_n$.

- Each literal l_i is either a predicate or the negation of a predicate.
- This means "h is true when $l_1, l_2, \ldots, l_n$ are simultaneously true"
  - e.g., `snowy(city) :- rainy(city), cold(city).`
- **Q**: How to specify disjunction (i.e., OR)?

Horn clauses

A Horn clause has a head h, which is a predicate, and a body, which is a list of literals l_1, l_2, . . . , l_n, written as $h \leftarrow l_1, l_2, \ldots, l_n$.

- Each literal l_i is either a predicate or the negation of a predicate.
- This means "h is true when $l_1, l_2, \ldots, l_n$ are simultaneously true"
  - e.g., `snowy(city) :- rainy(city), cold(city).`

- **Q**: How to specify disjunction (i.e., OR)?
  - `parent(x, y) :- father(x, y).`
  - `parent(x, y) :- mother(x, y).`

Horn clauses

A Horn clause has a head h, which is a predicate, and a body, which is a list of literals l_1, l_2, ..., l_n, written as $h \leftarrow l_1, l_2, \ldots, l_n$.

- Each literal l_i is either a predicate or the negation of a predicate.
- This means "h is true when $l_1, l_2, \ldots, l_n$ are simultaneously true"
  - e.g., `snowy(city) :- rainy(city), cold(city).`
- **Q**: How to specify disjunction (i.e., OR)?
  - `parent(x, y) :- father(x, y).`
  - `parent(x, y) :- mother(x, y).`
- When a Horn clause has no body and just a head, it is a fact.
  - e.g., `cold("Waterloo")`

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?

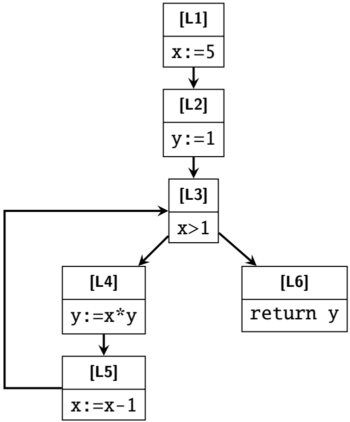
```
.type Label <: number

// control flow from l1 to l2
.decl flow(l1: Label, l2: Label)

// l is the start of the execution
.decl init_label(l: Label)

// l is the end of the execution
.decl exit_label(l: Label)
```

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}`;`

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

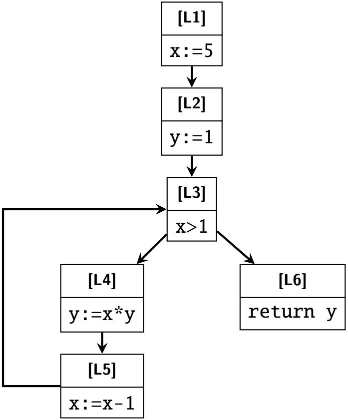
Q: How to encode the semantics of each instruction?

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.

```
.type Var <: symbol

// instruction l defines var v
.decl def(l: Label, v: Var)

// instruction l uses var v
.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

```
// var v defined at label def can reach *before* instruction l
.decl rd_entry(l: Label, v: Var, def: Label)

// var v defined at label def can reach *after* instruction l
.decl rd_exit(l: Label, v: Var, def: Label)

// rule 1: def of v can reach the end of def
rd_exit(l, v, l) :- def(l, v).

// rule 2: def of v can reach the end of l if l does not define v
rd_exit(l, v, def) :- rd_entry(l, v, def), !def(l, v).

// rule 3: def of v can reach next instruction
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

```
// var v defined at label def is alive *before* instruction l
.decl lv_entry(l: Label, v: Var, def: Label)

// var v defined at label def is alive *after* instruction l
.decl lv_exit(l: Label, v: Var, def: Label)

// rule 1: use of v make v alive before the use
lv_entry(l, v, l) :- use(l, v).

// rule 2: use of v can reach the entry of l if l does not define v
lv_entry(l, v, def) :- lv_exit(l, v, def), !def(l, v).

// rule 3: def of v can reach next instruction
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

```
.type Statement
.type Variable
.type Opcode
.type Value

.decl entry(s:Statement)
.decl edge(h:Statement, t:Statement)
.decl def(var:Variable, stmt:Statement)
.decl use(var:Variable, stmt:Statement)
.decl op(stmt:Statement, op:Opcode)
.decl value(var:Variable, val:Value)

.input entry, edge, def, use, op, value
```

Sample checkers

```
.decl uncheckedCall(u:Statement)

uncheckedCall(u) :-
    callResult(_, u),
    !checkedCallThrows(u),
    !checkedCallStateUpdate(u).
```

```
.decl reentrantCall(stmt:Statement)

reentrantCall(stmt) :-
    op(stmt, "CALL"),
    !protectedByLoc(stmt, _),
    gassy(stmt, gasVar),
    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.

$\langle$ **End** $\rangle$