Program Analysis for IoT/CPS

Hyungsub Kim

Purdue University

PURDUE UNIVERSITY

CERIAS

PurSecLab

About me

- A PhD student in Purdue CS
  - Joined in 2018
  - Working on how to apply static and dynamic analysis to robotic vehicle security
  - Published papers into security conferences (NDSS, USENIX, ACSAC)

Details of research topics:

1) Find bugs (fuzzing)
2) Automatically patch the bugs
3) Verify the fixed bugs

Outline

- Intro
- Terminology
- Static Analysis
- Dynamic Analysis

Goal (1)

1. Understanding terms in program analysis domain
   - Path-sensitive, flow-sensitive
   - Intra-procedural, Inter-procedural
   - Static single assignment (SSA), pointer analysis

But why should we care about these terms?

Goal (2)

1. why should we care about these terms?
   - To leverage existing program analysis tools
   - To understand security papers

> load and store operations recursively. For pointers, to identify data flow via pointer reference/dereference operators, we perform an inter-procedural, path-insensitive, and flow-sensitive points-to analysis [62]. More precisely, the profiling engine operates in three steps: (1) performs Andersen's pointer analysis [8] to identify aliases of the parameter variables, (2) transforms the code to its single static assignment form [59] and builds the data-flow graph (DFG), and (3) collects the def-use chain of the identified parameter variable from the built DFG.

<A paragraph on a paper in NDSS 2021>

Can you understand this paragraph?

Goal (3)

2. Understanding how each technique is used for improving security in CPS

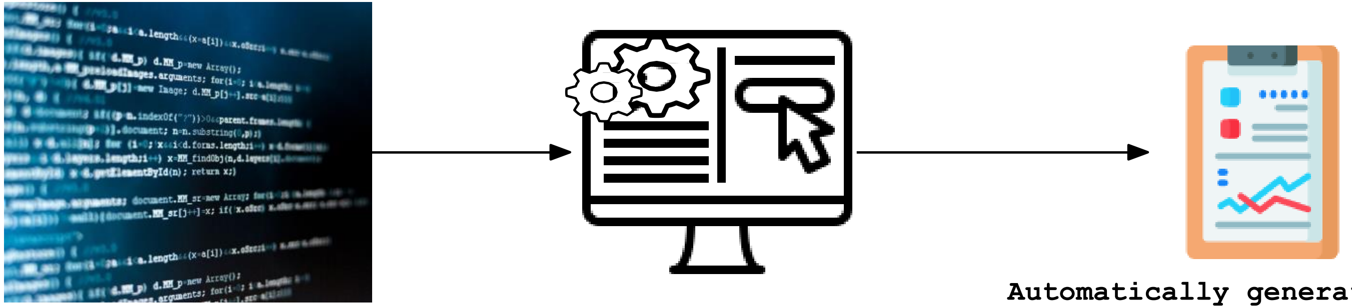
But why?

Unsolved research question → - Solve the research problems
- Publish papers

Apply program analysis to the unsolved problems

What is Program Analysis

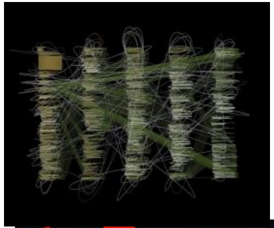
- A process of automatically analyzing behaviors of a program
- Applications:
    - Program understanding
    - Compiler optimizations
    - Bug finding

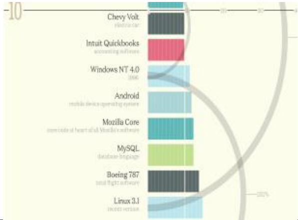


Automatically generated report

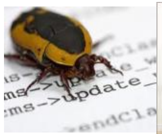
Why should we automate this analysis?

- Modern system software
  - Extremely large and complex but error-prone





More Complex!



Memory Leaks

Buffer Overflows

Null Pointers

Use-After-Frees

Data-races

More Buggy!

Existing Program Analysis Tools

	open-source	commercial
static	deepsource, Cppcheck, sonarqube, SVF-tool, Splint, Flawfinder, FindBugs, Semmle	MICRO FOCUS, SYNOPSYS, VERACODE, DEEPCODE by snyk, CodeScene Powered by Empear, Praxis, kiuwan An Idera, Inc. Company, SOURCEBRELLA, RIPSTECH, COVERITY by Synopsys, CODESONAR
dynamic	KLEE, LDRA, Valgrind, Iroh.js, sanitizers, Dmalloc, Jalangi2	Trustwave, CHECKMARX, SENTINEL Dynamic, OWASP, acunetix, HCL AppScan, detectify

Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it – inspection of its source code by examining all possible program paths*
  - + Pin-point bugs at source code level.
  - + Catch bugs earlier during software development.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

Static Analysis vs. Dynamic Analysis

Static Analysis

- *Analyze a program without actually executing it – inspection of its source code by examining all possible program paths*
  - + Pin-point bugs at source code level.
  - + Catch bugs earlier during software development.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

Dynamic Analysis

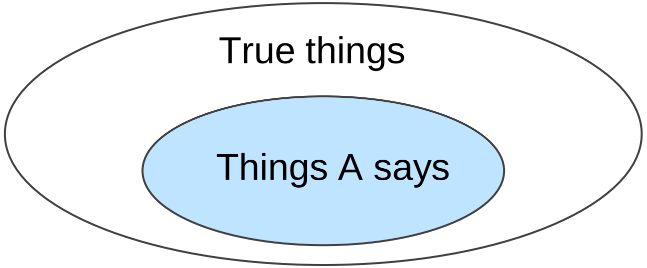
- *Analyze a program at runtime – inspection of its running program by examining some executable paths depending on specific test inputs*
  - + Identify bugs at runtime (catch it when you observe it).
  - + Zero or very low false alarm rates.
  - - Runtime overhead due to code instrumentation.
  - - May miss bugs (false negative) due to under-approximation.

Outline

- Intro
- Terminology
- Static Analysis
- Dynamic Analysis

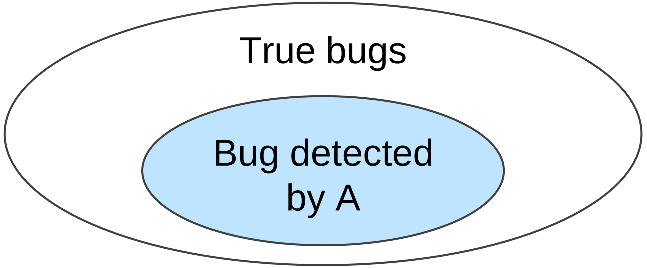
Characterizing Program Analyses (1)

- Soundness
  - If analysis A says that X is true, then X is true.



Trivially sound: Say nothing

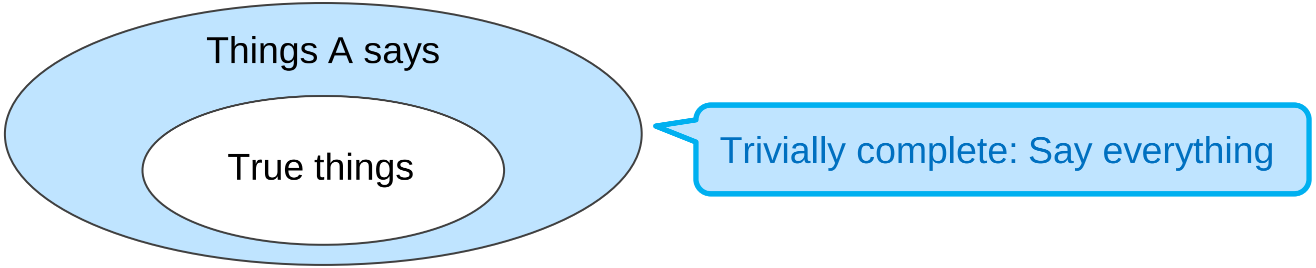
- Example
  - If analysis A says that X is buggy, then X is buggy.



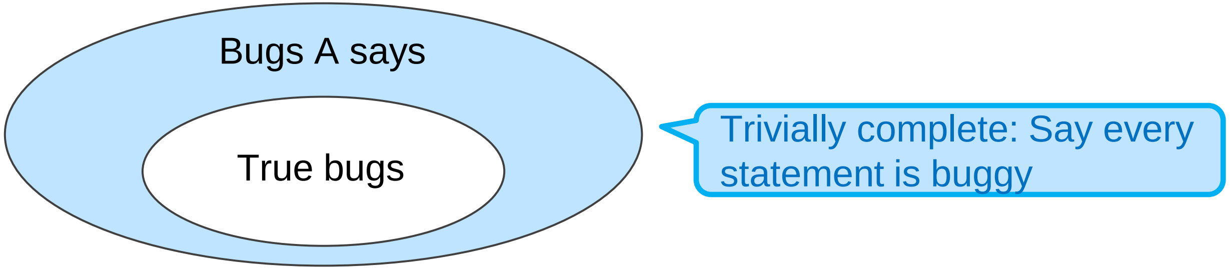
Trivially sound: Say there is no buggy statement

Characterizing Program Analyses (1)

- Completeness
  - If X is true, then analysis A says X is true.



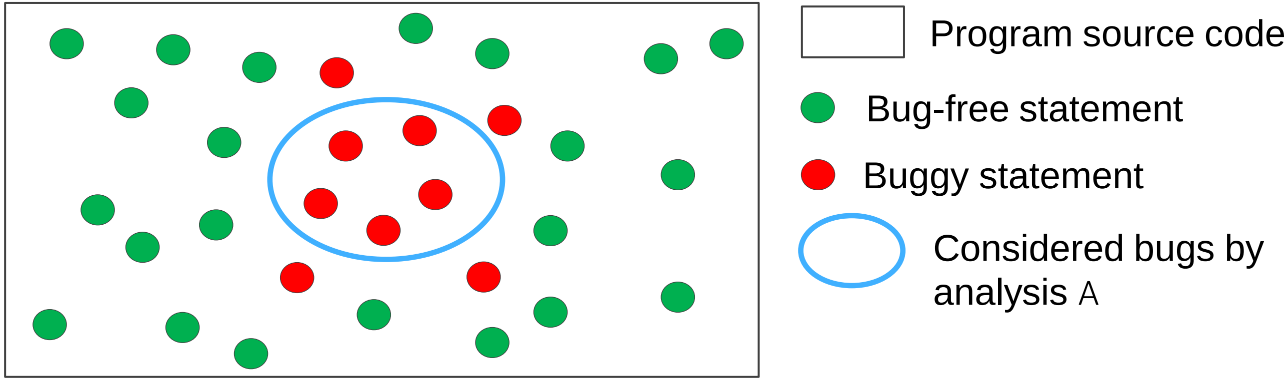
- Example
  - If X is buggy, then analysis A says X is buggy



Sound vs. Complete (1)

- Is analysis A sound? Yes
  - Why? If analysis A says that X is buggy, then X is buggy.

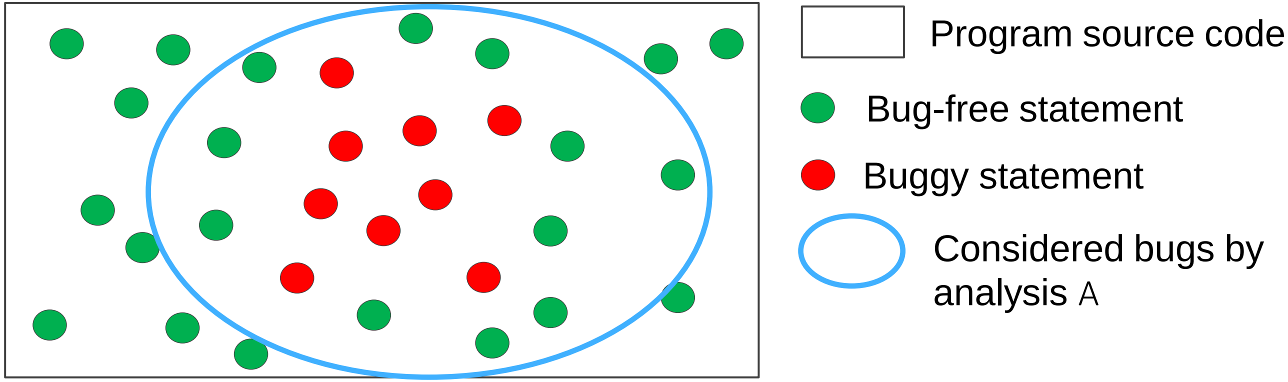
- Is analysis A complete? No
  - Why? If X is buggy, then analysis A says X is buggy.



Sound vs. Complete (2)

- Is analysis A sound? No
  - Why? If analysis A says that X is buggy, then X is buggy.

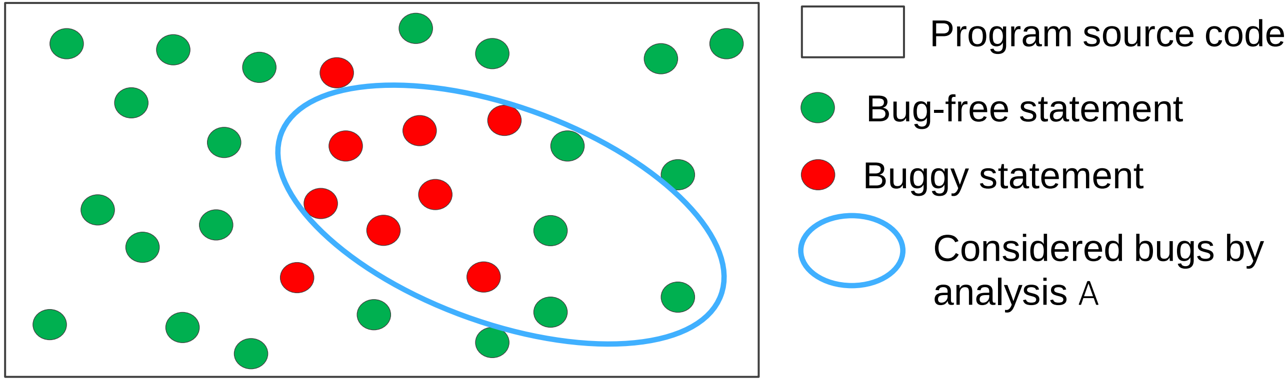
- Is analysis A complete? Yes
  - Why? If X is buggy, then analysis A says X is buggy.



Sound vs. Complete (3)

- Is analysis A sound? No
  - Why? If analysis A says that X is buggy, then X is buggy.

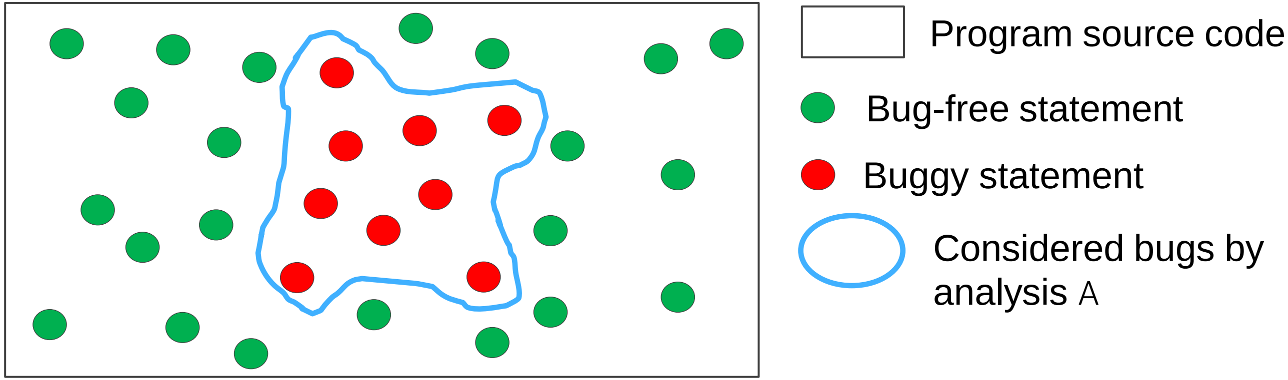
- Is analysis A complete? No
  - Why? If X is buggy, then analysis A says X is buggy.



Sound vs. Complete (4)

- Is analysis A sound? Yes
  - Why? If analysis A says that X is buggy, then X is buggy.

- Is analysis A complete? Yes
  - Why? If X is buggy, then analysis A says X is buggy.



Program Representations

- Original representations of programs
  - Source code
  - Binaries

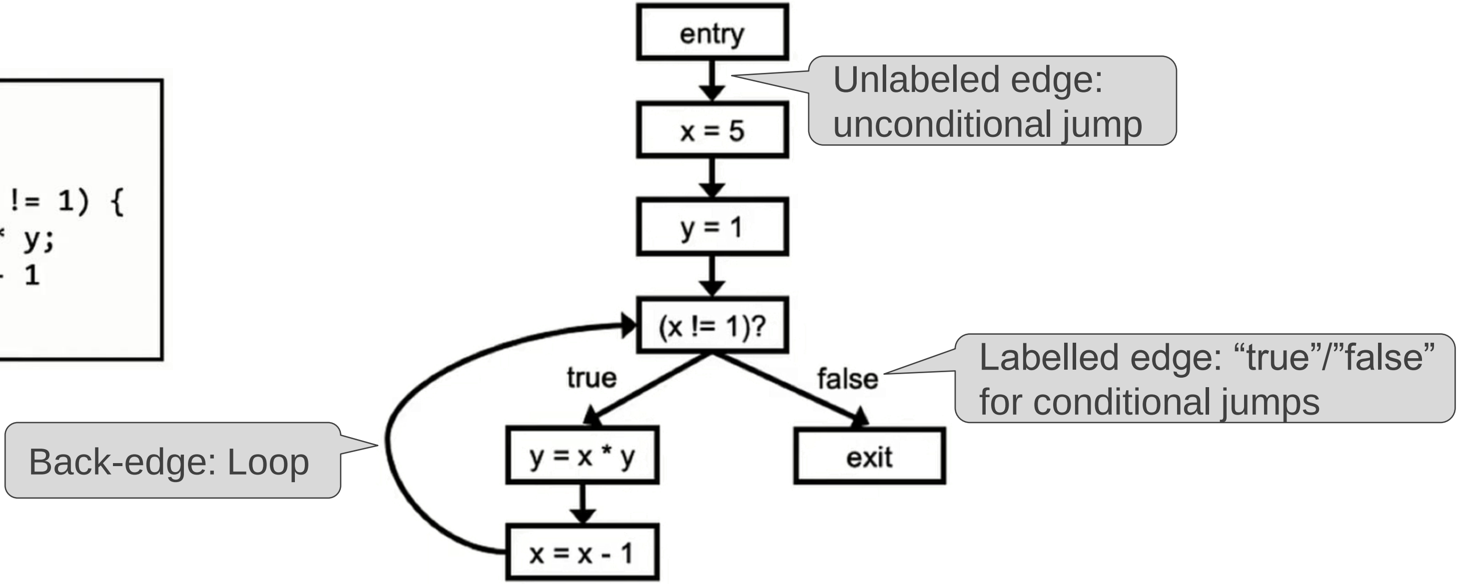
- They are hard for machines to analyze

- Software is translated into certain representations before analyses are applied.

Control-Flow Graph

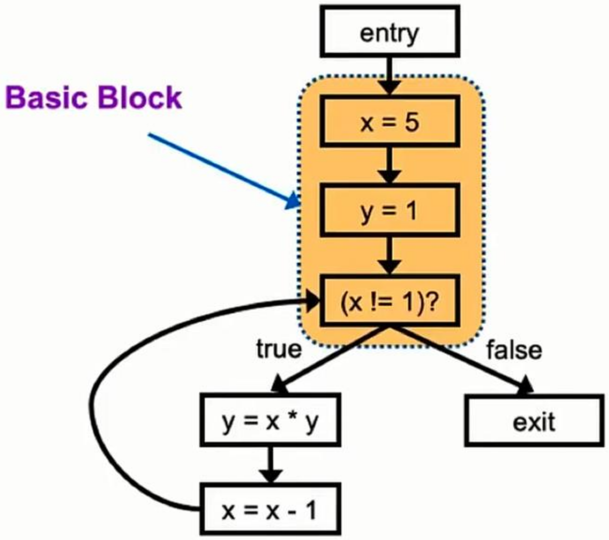
- Directed graph
  - Edge: summarizing flow of graph
  - Node: a statement in a program

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```



Basic Block (1)

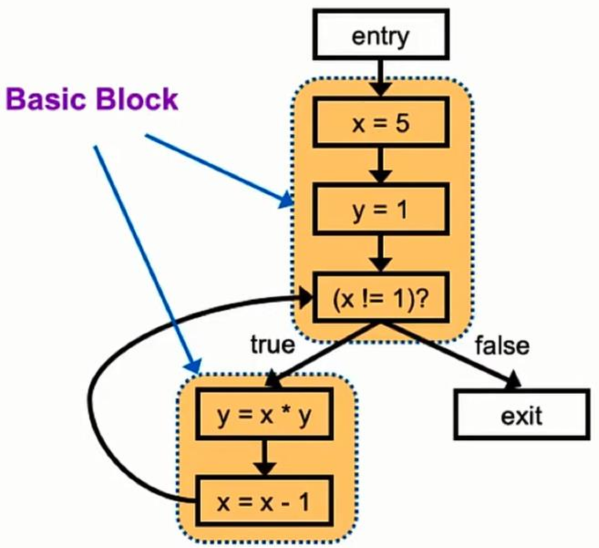
- Definition
  - Group statements without intervening control flow




```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Basic Block (2)

- Definition
  - Group statements without intervening control flow

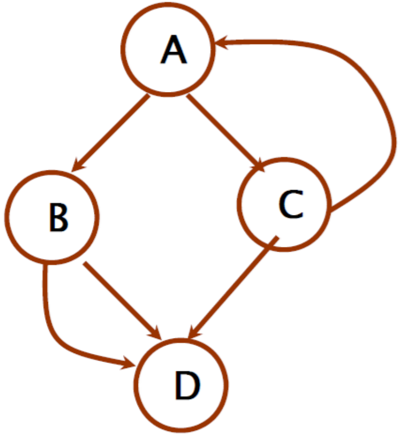



```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Call Graph

- Node
    - Represents a function
- Edge
    - Represents a function invocation

```
void A( ) {
   B( );
   C( );
}

void B( ) {
L1:   D( );
L2:   D( );
}

void C ( ) {
   D( );
   A( );
}

void D ( ) {
}
```



Outline

- Intro
- Terminology
- Static Analysis
- Dynamic Analysis

Def-Use and Use-Def chains

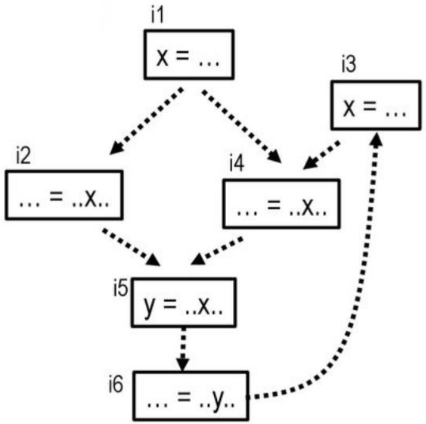
- Dataflow analysis problem
  - Find all sites where a variable X is used
    - (e.g., y = X;)
  - Find all sites where that variable X was last defined
    - (e.g., X = 1;)

Def-Use and Use-Def chains

- Def-Use (DU) chains
  - Link each def (assigns-to) of a variable to all uses
- Use-Def (UD) chains
  - Link each use of a variable to its def

Var	Def	Uses
x	i1	i2, i4, i5
x	i3	i4, i5
y	i5	I6

<Def-Use chain>

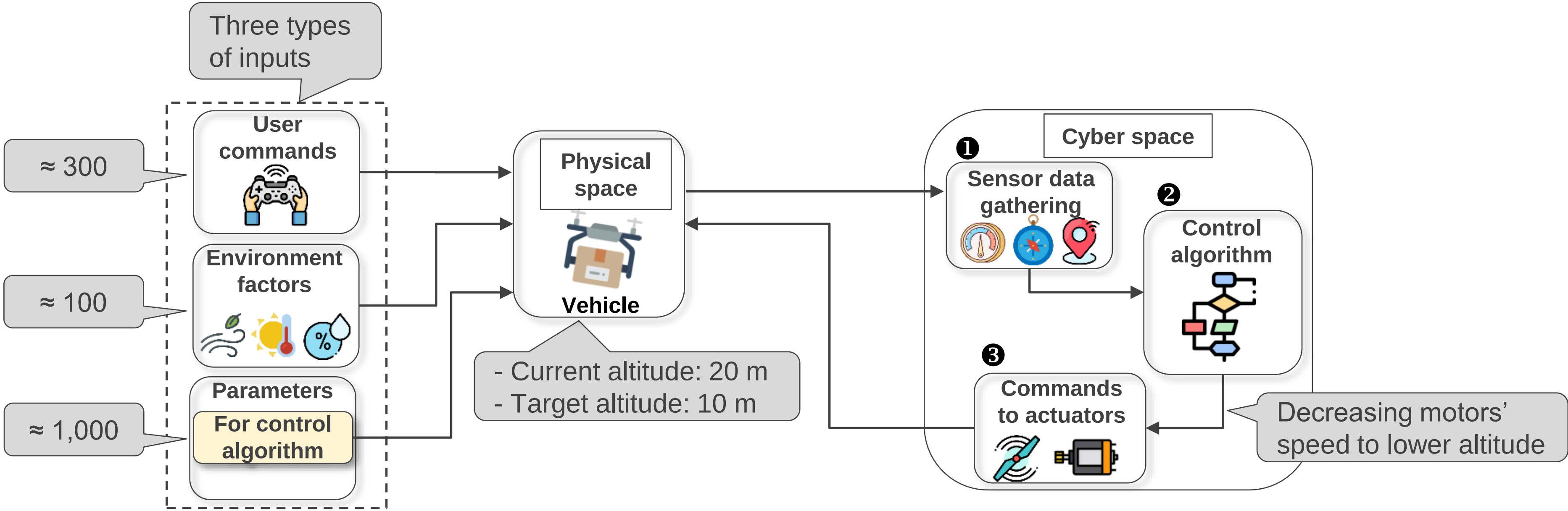


Var	Use	Defs
x	i2	i1
x	i4	i1, i3
x	i5	i1, i3
y	i6	i5

<Use-Def chain>

Applications of Def-Use chains (1)

- Workflow of robotic vehicles (RV)



Challenging issue: Huge input space

Applications of Def-Use chains (2)

- Problem 1:
  - You need to find inputs that change the RV's altitude state.

configuration parameter

```
1    AP_GROUPINFO("TEMP", … , ground_temp);   ①
2    _user_temp = ground_temp + 273.15f;   ②
…
315  temp = _user_temp   ③
…
320  altitude = 153.8462f * temp * …
```

Source code

← Starting point for a def-use chain of *TEMP* parameter

<Def-Use chain>

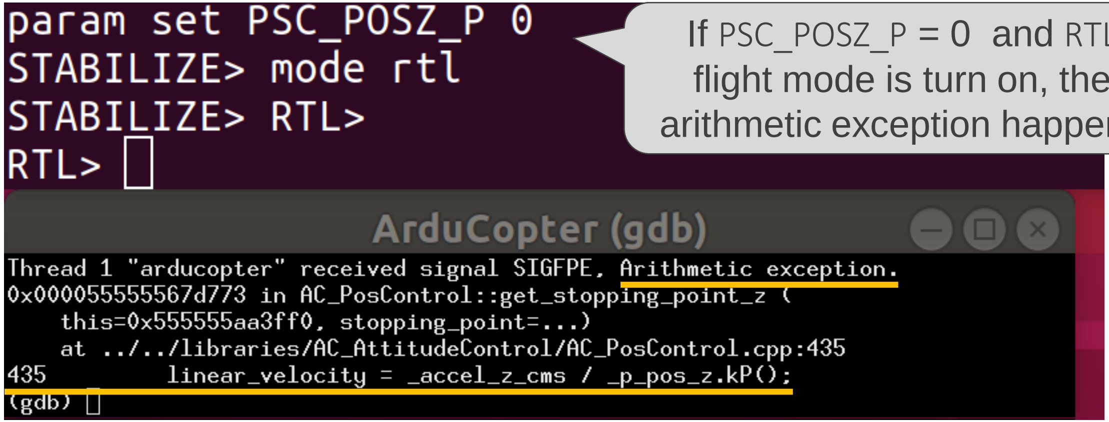
[*definition* | *write access* | *read access*]

(1) [*ground_temp* | *line 1* | *line 2*] ①
(2) [*_user_temp* | *line 2* | *line 315*] ②
(3) [*temp* | *line 315* | *line 320*] ③
(4) [altitude | line 320 | …] ④

Result: "TEMP" configuration parameter will change the RV's altitude

Applications of Def-Use chains (3)

- Problem 2:
  - You need to develop an automatic program repair tool.
  - It automatically fixes divide-by-zero bugs.



If PSC_POSZ_P = 0 and RTL flight mode is turn on, the arithmetic exception happens

Applications of Def-Use chains (3)

- Two cases:
  1) There is no any “if check statement” to prevent the divide-by-zero.
  2) There is an “if statement” to prevent such error. But, the check statement is incorrect.

PSC_POSZ_P = 0.00000001 still leads to the divide-by-zero.

```
…
_accel_z_cms / _p_pos_z.KP();
```

<Case 1>

```
if (_p_pos_z.KP() <= 0.0f) {
        return;
}
_accel_z_cms / _p_pos_z.KP();
```

<Case 2>

Applications of Def-Use chains (3)

- Backtracking def-use chains
  - To find an "if statement" preventing the divide-by-zero.

Def-use of _p_pos_z

Variable		Operand 1	Operand 2
String	Address		
%cmp	0x5d40220	0x5d3f138	0x1501410

<Backtracking the def-use chain of _p_pos_z >

```
[49.def] name:   %_p_pos_z = getelementptr inbounds %class.AC_PosControl, %class.AC_PosC
ontrol* %this1, i32 0, i32 10, address: 0x5d3ff88
                         (# of operands: 3
                                 (operand) name: this1, address: 0x5d3e4b8
                                 (operand) name: , address: 0x1470480
                                 (operand) name: , address: 0x1756440

[50.def] name:   %call10 = call dereferenceable(4) %class.AP_ParamT.1* @_ZNK4AC_P2kPEv(%
class.AC_P* %_p_pos_z), address: 0x5d40060
                         (# of operands: 2
                                 (operand) name: _p_pos_z, address: 0x5d3ff88
                                 (operand) name: _ZNK4AC_P2kPEv, address: 0x1e2c658

[51.def] name:   %call11 = call dereferenceable(4) float* @_ZNK9AP_ParamTIfL11ap_var_typ
e4EEcvRKfEv(%class.AP_ParamT.1* %call10), address: 0x5d40150
                         (# of operands: 2
                                 (operand) name: call10, address: 0x5d40060
                                 (operand) name: _ZNK9AP_ParamTIfL11ap_var_type4EEcvRKfEv
, address: 0x19f2558

[52.def] name:   %11 = load float, float* %call11, align 4, address: 0x5d3f138
                         (# of operands: 1
                                 (operand) name: call11, address: 0x5d40150

[53.def] name:   %cmp = fcmp ole float %11, 0.000000e+00, address: 0x5d40220
                         (# of operands: 2
                                 (operand) name: , address: 0x5d3f138
                                 (operand) name: , address: 0x1501410
```

Instruction

Operands of the instruction

Applications of Def-Use chains (3)

- Backtracking def-use chains
  - To find an “if statement” preventing the divide-by-zero.

Def-use of *_p_pos_z*

Variable		Operand 1	Operand 2
String	**Address**		
%11	0x5d3f138	0x5d40150	-
%cmp	0x5d40220	0x5d3f138	0x1501410

<Backtracking the def-use chain of _p_pos_z >

```
[49.def] name:   %_p_pos_z = getelementptr inbounds %class.AC_PosControl, %class.AC_PosC
ontrol* %this1, i32 0, i32 10, address: 0x5d3ff88
                          (# of operands: 3
                                    (operand) name: this1, address: 0x5d3e4b8
                                    (operand) name: , address: 0x1470480
                                    (operand) name: , address: 0x1756440

[50.def] name:   %call10 = call dereferenceable(4) %class.AP_ParamT.1* @_ZNK4AC_P2kPEv(%
class.AC_P* %_p_pos_z), address: 0x5d40060
                          (# of operands: 2
                                    (operand) name: _p_pos_z, address: 0x5d3ff88
                                    (operand) name: _ZNK4AC_P2kPEv, address: 0x1e2c658

[51.def] name:   %call11 = call dereferenceable(4) float* @_ZNK9AP_ParamTIfL11ap_var_typ
e4EEcvRKfEv(%class.AP_ParamT.1* %call10), address: 0x5d40150
                          (# of operands: 2
                                    (operand) name: call10, address: 0x5d40060
                                    (operand) name: _ZNK9AP_ParamTIfL11ap_var_type4EEcvRKfEv
, address: 0x19f2558

[52.def] name:   %11 = load float, float* %call11, align 4, address: 0x5d3f138
                          (# of operands: 1
                                    (operand) name: call11, address: 0x5d40150

[53.def] name:   %cmp = fcmp ole float %11, 0.000000e+00, address: 0x5d40220
                          (# of operands: 2
                                    (operand) name: , address: 0x5d3f138
                                    (operand) name: , address: 0x1501410
```

Instruction

Operands of the instruction

Applications of Def-Use chains (3)

- Backtracking def-use chains
  - To find an “if statement” preventing the divide-by-zero.

Def-use of _p_pos_z

Variable		Operand 1	Operand 2
String	Address		
%call11	0x5d40150	0x5d40060	0x19f2558
%11	0x5d3f138	0x5d40150	-
%cmp	0x5d40220	0x5d3f138	0x1501410

<Backtracking the def-use chain of _p_pos_z >

```
[49.def] name:   %_p_pos_z = getelementptr inbounds %class.AC_PosControl, %class.AC_PosC
ontrol* %this1, i32 0, i32 10, address: 0x5d3ff88
                         (# of operands: 3
                                 (operand) name: this1, address: 0x5d3e4b8
                                 (operand) name: , address: 0x1470480
                                 (operand) name: , address: 0x1756440

[50.def] name:   %call10 = call dereferenceable(4) %class.AP_ParamT.1* @_ZNK4AC_P2kPEv(%
class.AC_P* %_p_pos_z), address: 0x5d40060
                         (# of operands: 2
                                 (operand) name: _p_pos_z, address: 0x5d3ff88
                                 (operand) name: _ZNK4AC_P2kPEv, address: 0x1e2c658

[51.def] name:   %call11 = call dereferenceable(4) float* @_ZNK9AP_ParamTIfL11ap_var_typ
e4EEcvRKfEv(%class.AP_ParamT.1* %call10), address: 0x5d40150
                         (# of operands: 2
                                 (operand) name: call10, address: 0x5d40060
                                 (operand) name: _ZNK9AP_ParamTIfL11ap_var_type4EEcvRKfEv
, address: 0x19f2558

[52.def] name:   %11 = load float, float* %call11, align 4, address: 0x5d3f138
                         (# of operands: 1
                                 (operand) name: call11, address: 0x5d40150

[53.def] name:   %cmp = fcmp ole float %11, 0.000000e+00, address: 0x5d40220
                         (# of operands: 2
                                 (operand) name: , address: 0x5d3f138
                                 (operand) name: , address: 0x1501410
```

Instruction

Operands of the instruction

Applications of Def-Use chains (3)

- Backtracking def-use chains
  - To find an “if statement” preventing the divide-by-zero.

Def-use of _p_pos_z

Variable		Operand 1	Operand 2
String	Address		
%call10	0x5d40060	0x5d3ff88	0x1e2c658
%call11	0x5d40150	0x5d40060	0x19f2558
%11	0x5d3f138	0x5d40150	-
%cmp	0x5d40220	0x5d3f138	0x1501410

<Backtracking the def-use chain of _p_pos_z >

```
[49.def] name:   %_p_pos_z = getelementptr inbounds %class.AC_PosControl, %class.AC_PosC
ontrol* %this1, i32 0, i32 10, address: 0x5d3ff88
                                   (# of operands: 3
                                        (operand) name: this1, address: 0x5d3e4b8
                                        (operand) name: , address: 0x1470480
                                        (operand) name: , address: 0x1756440

[50.def] name:   %call10 = call dereferenceable(4) %class.AP_ParamT.1* @_ZNK4AC_P2kPEv(%
class.AC_P* %_p_pos_z), address: 0x5d40060
                                   (# of operands: 2
                                        (operand) name: _p_pos_z, address: 0x5d3ff88
                                        (operand) name: _ZNK4AC_P2kPEv, address: 0x1e2c658

[51.def] name:   %call11 = call dereferenceable(4) float* @_ZNK9AP_ParamTIfL11ap_var_typ
e4EEcvRKfEv(%class.AP_ParamT.1* %call10), address: 0x5d40150
                                   (# of operands: 2
                                        (operand) name: call10, address: 0x5d40060
                                        (operand) name: _ZNK9AP_ParamTIfL11ap_var_type4EEcvRKfEv
, address: 0x19f2558

[52.def] name:   %11 = load float, float* %call11, align 4, address: 0x5d3f138
                                   (# of operands: 1
                                        (operand) name: call11, address: 0x5d40150

[53.def] name:   %cmp = fcmp ole float %11, 0.000000e+00, address: 0x5d40220
                                   (# of operands: 2
                                        (operand) name: , address: 0x5d3f138
                                        (operand) name: , address: 0x1501410
```

Instruction

Operands of the instruction

Applications of Def-Use chains (3)

- Backtracking def-use chains
  - To find an “if statement” preventing the divide-by-zero.

Result: This code snippet compares _p_pos_z with 0!

Variable		Operand 1	Operand 2
String	Address		
%_p_pos_z	0x5d3ff88	0x5d3e4b8	0x1470480
%call10	0x5d40060	0x5d3ff88	0x1e2c658
%call11	0x5d40150	0x5d40060	0x19f2558
%11	0x5d3f138	0x5d40150	-
%cmp	0x5d40220	0x5d3f138	0x1501410

<Backtracking the def-use chain of _p_pos_z >

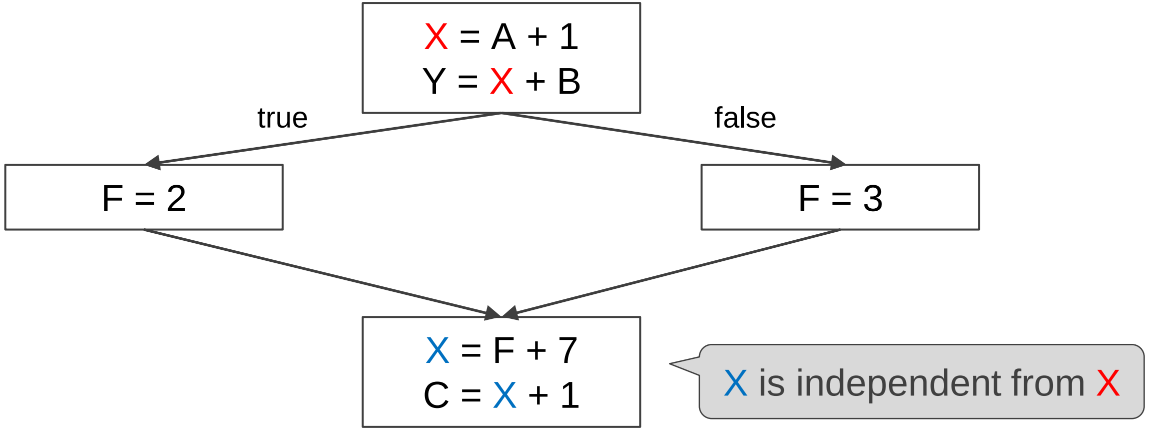
```
[49.def] name:   %_p_pos_z = getelementptr inbounds %class.AC_PosControl, %class.AC_PosC
ontrol* %this1, i32 0, i32 10, address: 0x5d3ff88
                 (# of operands: 3
                        (operand) name: this1, address: 0x5d3e4b8
                        (operand) name: , address: 0x1470480
                        (operand) name: , address: 0x1756440

[50.def] name:   %call10 = call dereferenceable(4) %class.AP_ParamT.1* @_ZNK4AC_P2kPEv(%
class.AC_P* %_p_pos_z), address: 0x5d40060
                 (# of operands: 2
                        (operand) name: _p_pos_z, address: 0x5d3ff88
                        (operand) name: _ZNK4AC_P2kPEv, address: 0x1e2c658

[51.def] name:   %call11 = call dereferenceable(4) float* @_ZNK9AP_ParamTIfL11ap_var_typ
e4EEcvRKfEv(%class.AP_ParamT.1* %call10), address: 0x5d40150
                 (# of operands: 2
                        (operand) name: call10, address: 0x5d40060
                        (operand) name: _ZNK9AP_ParamTIfL11ap_var_type4EEcvRKfEv
, address: 0x19f2558

[52.def] name:   %11 = load float, float* %call11, align 4, address: 0x5d3f138
                 (# of operands: 1
                        (operand) name: call11, address: 0x5d40150

[53.def] name:   %cmp = fcmp ole float %11, 0.000000e+00, address: 0x5d40220
                 (# of operands: 2
                        (operand) name: , address: 0x5d3f138
                        (operand) name: , address: 0x1501410
```

Instruction

Operands of the instruction

Same Variable Name May Be Unrelated

- The values in reused storage locations
  - May be probably independent
- Problem of this situation
  - Unrelated uses of same variable are mixed together
    - This complicates program analysis



Static Single Assignment (SSA)

- Idea
  - Each variable be assigned exactly once, and every variable be defined before it is used

- Why?
  - Explicitly express different definitions of variables

Original

$a \leftarrow x + y$
$b \leftarrow a + x$
$a \leftarrow b + 2$
$c \leftarrow y + 1$
$a \leftarrow c + a$

SSA

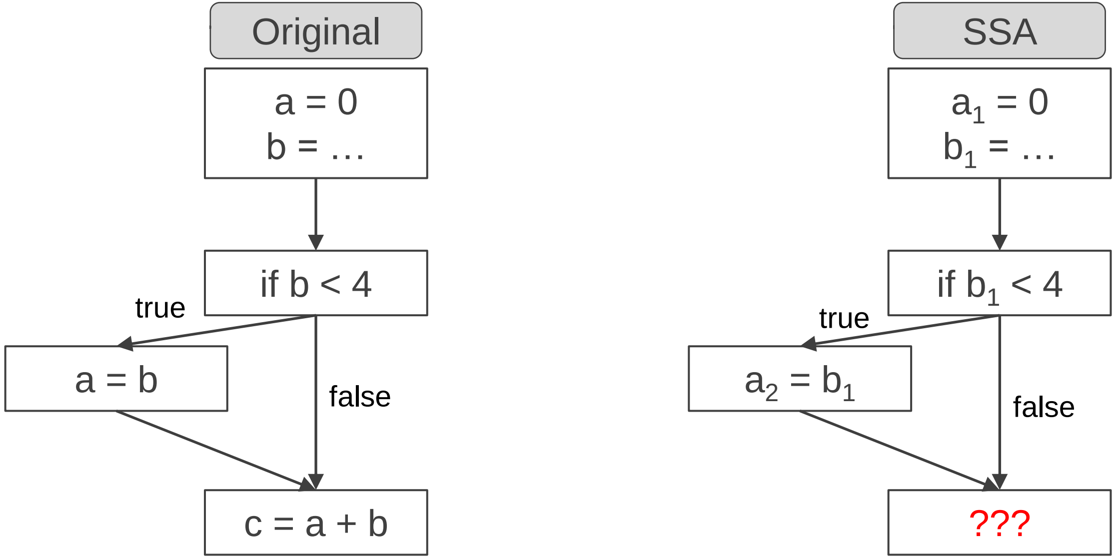
$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_2 \leftarrow b_1 + 2$
$c_1 \leftarrow y + 1$
$a_3 \leftarrow c_1 + a_2$

```
[Original]

if (a < c) {
    ....
}
```

```
[SSA]

if (a2 < c1) {
    ....
}
```

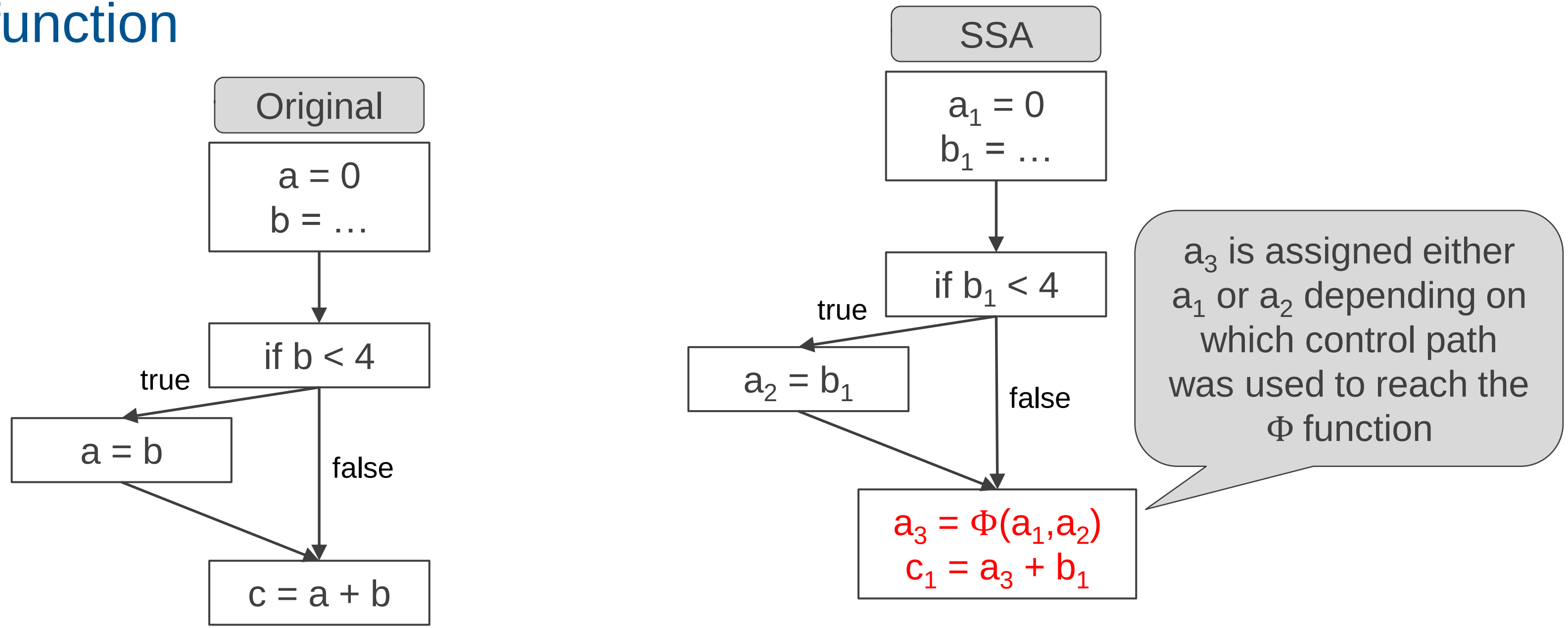
Merge Points (SSA)

- Issue
  - How to handle merge points in the flowgraph?



Merge Points (SSA)

- Issue
  - How to handle merge points in the flowgraph?
- Solution
  - Φ-function



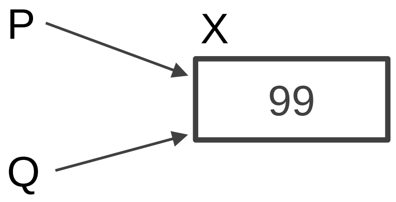
PURDUE UNIVERSITY CERIAS PurSecLab

Pointer Analysis

- What memory locations can a pointer expression refer to?

- Alias analysis
  - When do two pointer expressions refer to the same storage location?

```
int X = 99;
P = &X;
Q = P;

*P and *Q alias
```



Why do we want to know?

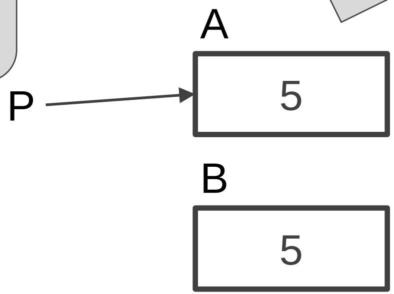
- Pointer analysis tells us what memory locations code uses or modifies

- Useful in many analyses

If *P aliases A or B, then the second computation of A+B is not redundant

Let’s assume that A+B is 10.
Then, can we do that?
*P = 10;
Y = 10;

E.g.,
*P = A + B;
Y = A + B;



Pointer Operations in C

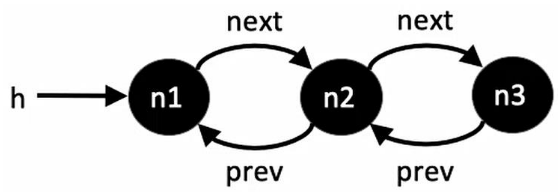
- Recall C pointer semantics
  - &a: Address of a
  - *a: Object pointed to by a
  - *(&a) = a: Converse operators

	Referencing • Create location	Dereferencing read • Access location • Indirect read	Dereferencing write • Access location • Indirect write	Aliasing • Copy pointer
C	a = &b	int *b = &c a = *b	int *a = &c *a = b	a = b
JAVA	a = new A()	a = b.f	a.f = b	a = b

Why Is Pointer Analysis Hard?

- Issue
  - There are infinite many ways to express the same data.

```
class Node {
  int data;
  Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```



h.data
h.next.prev.data
h.next.next.prev.prev.data
h.next.prev.next.prev.data

And many more ...

Approximation to the Rescue

- Pointer analysis problem is undecidable
  - We must sacrifice some combinations of
    - Soundness, completeness, termination

- Many sound approximate algorithms for pointer analysis
  - Differ in two key aspects
    - How to abstract the heap
    - How to abstract control-flow

May-alias Analysis vs. Must-alias Analysis

- May analysis assumes
  - Aliasing that may occur during execution

I will explain only the may-alias analysis

- Must analysis assumes
  - Aliasing that must occur during execution

```
If (user.input = A) {
    P = Q;
}
```

```
P.foo = 1;
Q.foo = 2;
Y = P.foo + 3;
```

***May analysis**
Assumption: Q may alias P
Analysis results:
  Case 1: Y = 4
  Case 2: Y = 5

***Must analysis**
Assumption: Q must alias P
Analysis results: Y = 5

Two Kinds of Pointers

- Heap-directed

```
p = new … or p = malloc(…)
```

- Stack-directed

```
int *p = NULL, v = 0;
p = &v
```

I will explain only the stack-directed pointer analysis

Pointer Analysis Algorithm

- Andersen's Points-To Analysis
  - Asymptotic performance is $O(n^3)$
  - Context-insensitive, flow-insensitive, path-insensitive
  - Four collecting rules
    - Referencing
    - Copy
    - Dereferencing (indirect) read
    - Dereferencing (indirect) write

Context Sensitivity

- Consider calling context

```
int foo (int i) {

   return i;
}
…
y1 = foo (1);
y2 = foo (2);
```

With context sensitivity
- More precise
- We have one i per call site of foo
- y1 is 1
- y2 is 2

Without context sensitivity
- Less precise, but faster
- We have one i total
- y1 is {1, 2}
- y2 is {1, 2}

Flow Sensitivity

- Consider control flow and order of execution

```
x = 2;
y = x;
x = 3;
```

With flow sensitivity

- y is 2

Without flow sensitivity

- y is {2, 3}

Path Sensitivity

- Consider properties inferred from order of execution

Line

```
1: x = 0;
2: if (P) {
3:     x = 1;
4: }
5: y = 2;
6:
7: If (P) {
8:     y = x;
9: }
```

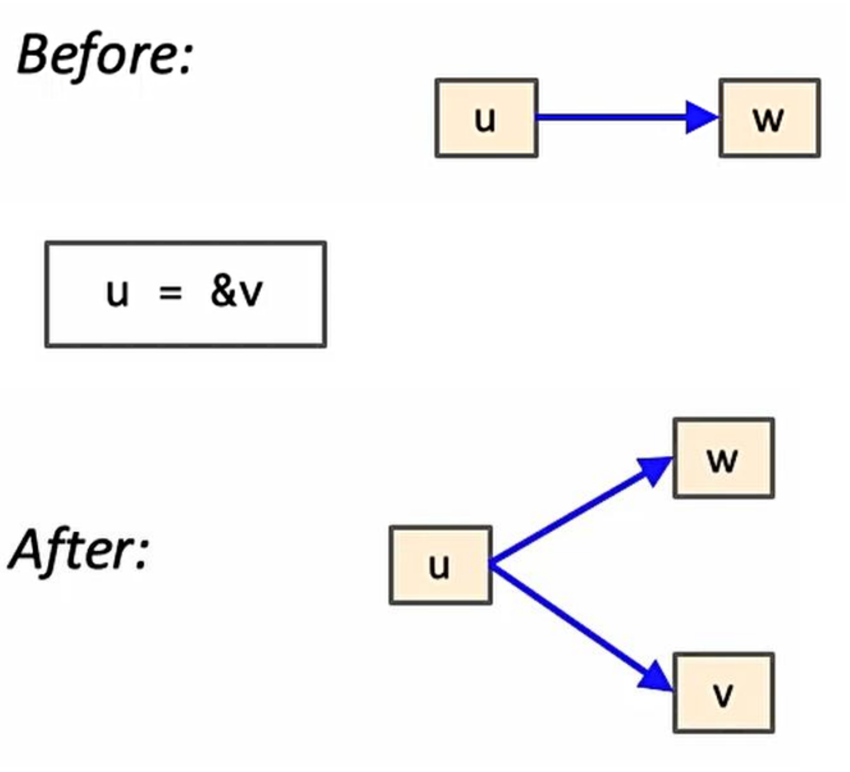
With path sensitivity

- y is {1, 2} at line 8
- Records that x = 0 when P = false
- Knows that line 8 is executed only if P = true (i.e., x ≠ 0 at line 8)

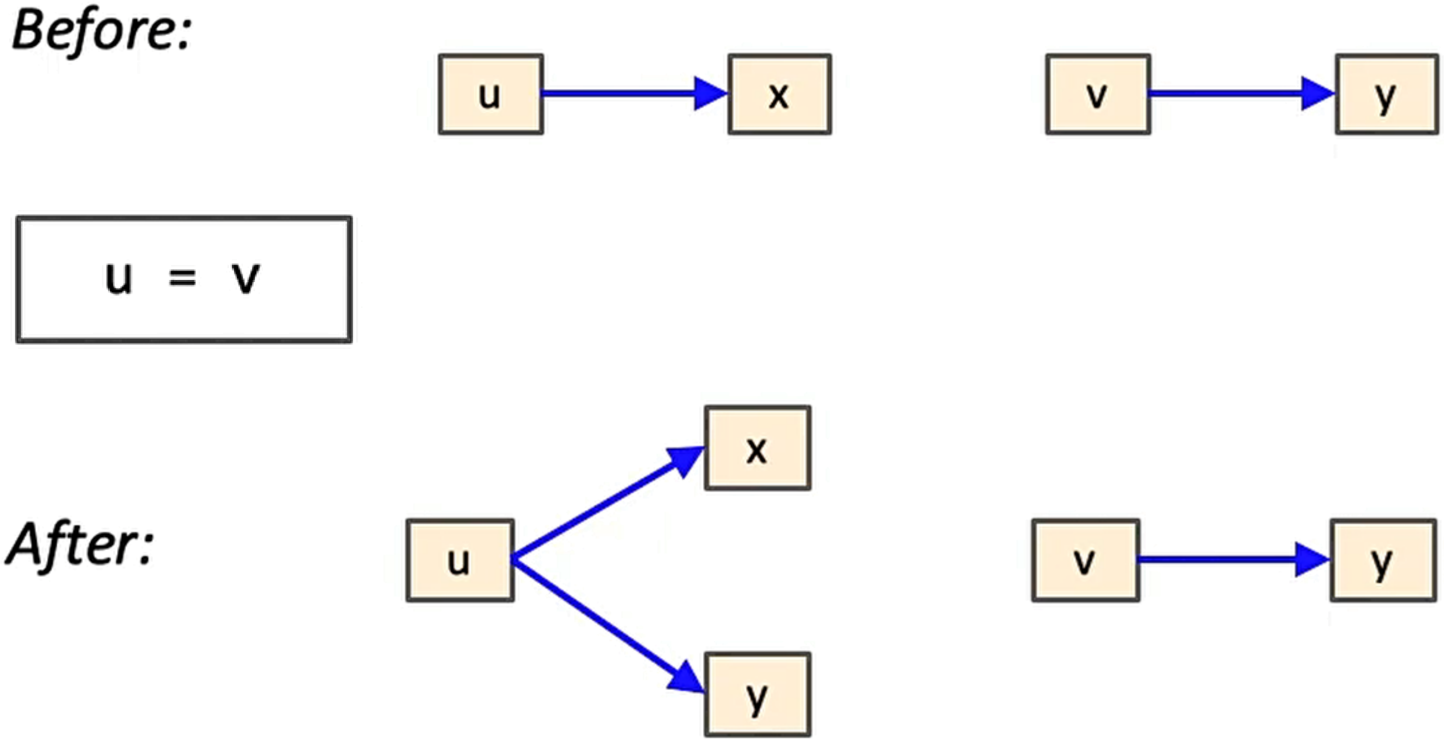
Without path sensitivity

- y is {0, 1, 2} at line 8
- Less precise

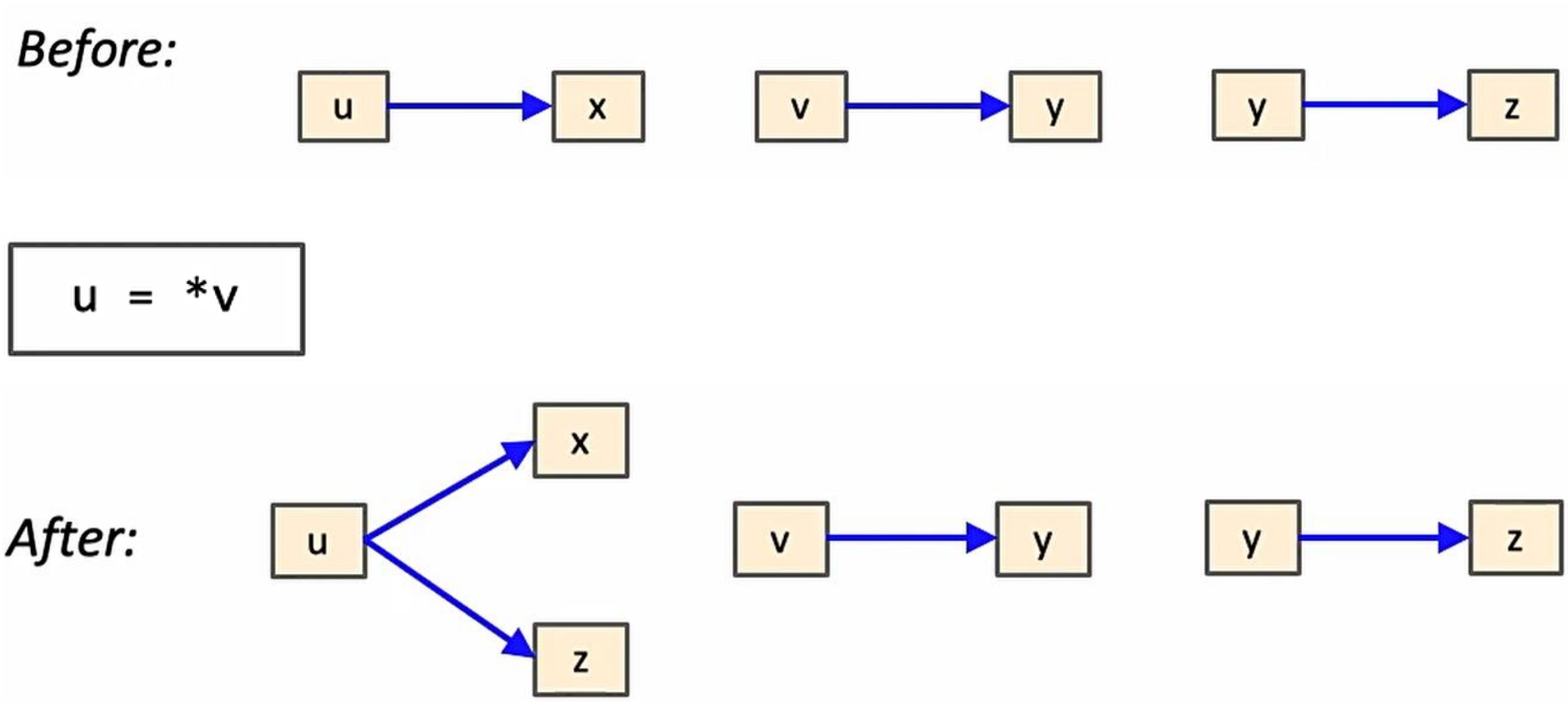
Rule for Referencing



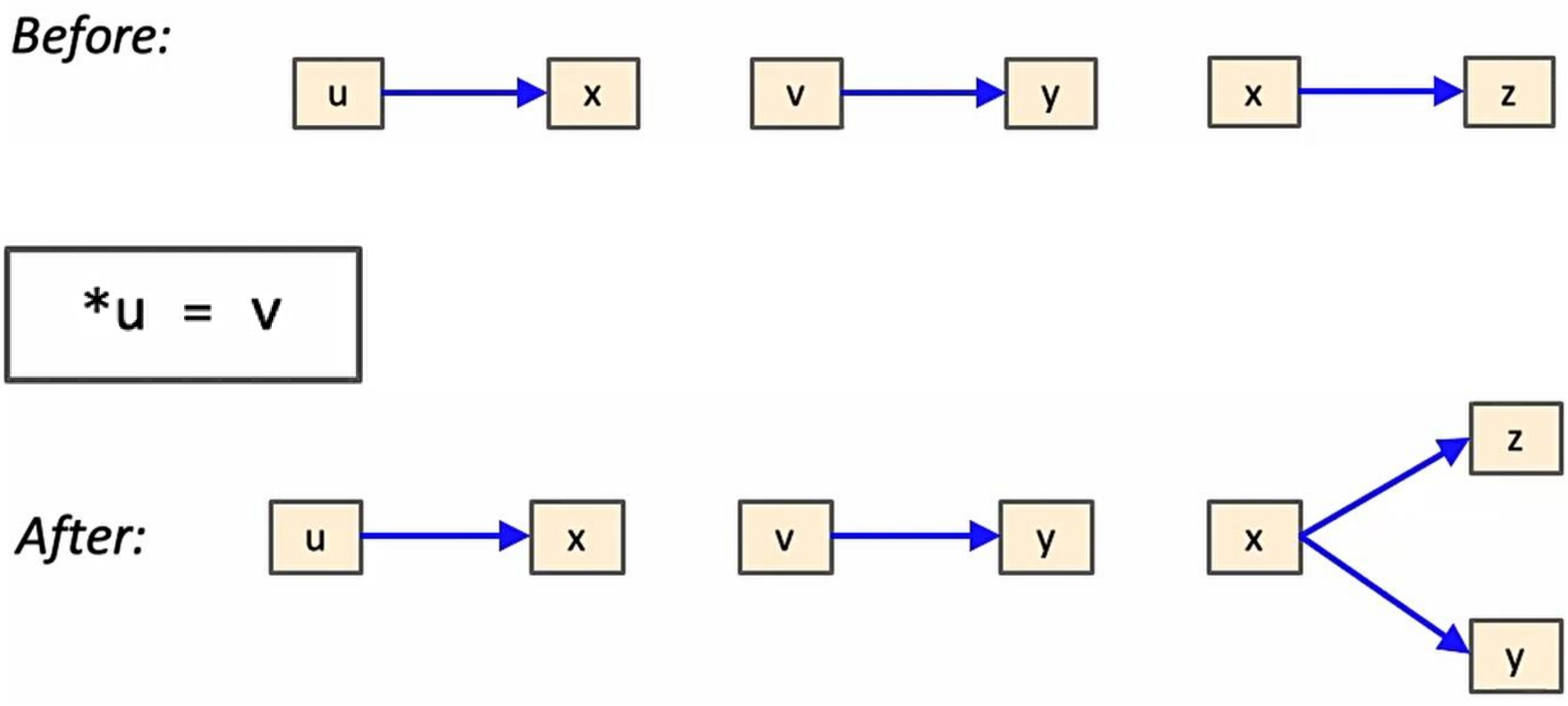
Rule for Copy



Rule for Indirect Read



Rule for Indirect Write



Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```

Recall: Andersen's Algorithm

graph = empty

repeat:

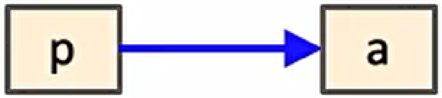
    for (each statement s in program)

        apply rule corresponding to s on graph

until graph stops changing

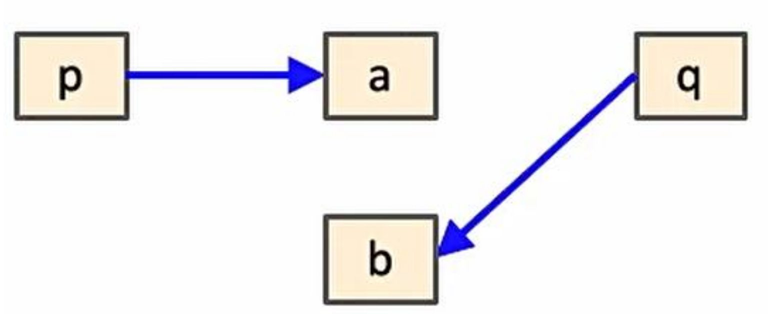
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



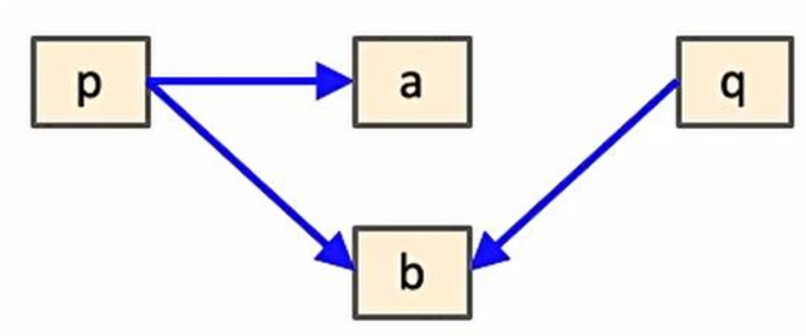
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



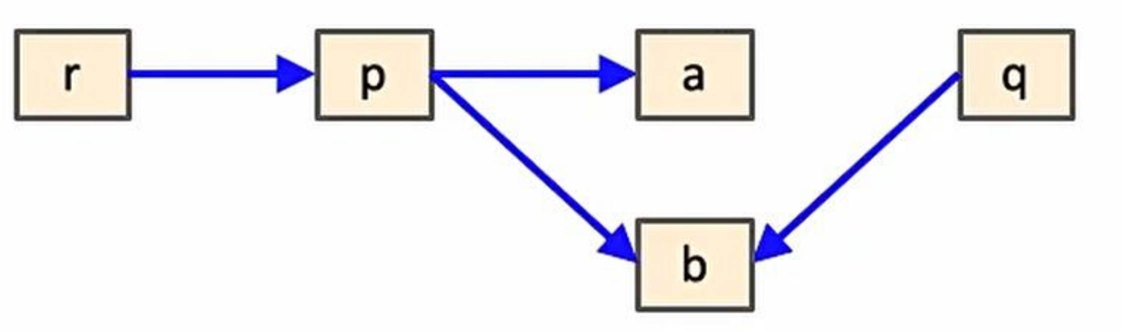
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



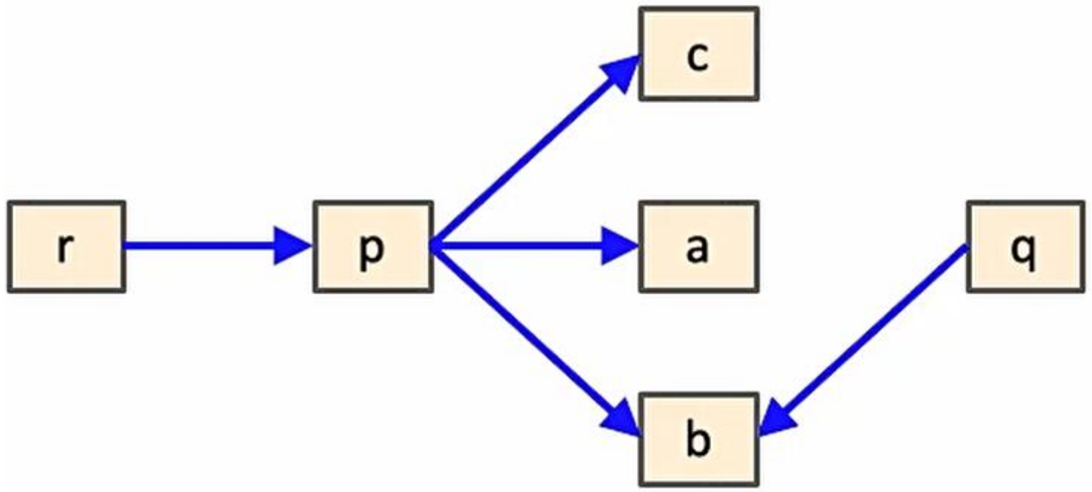
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



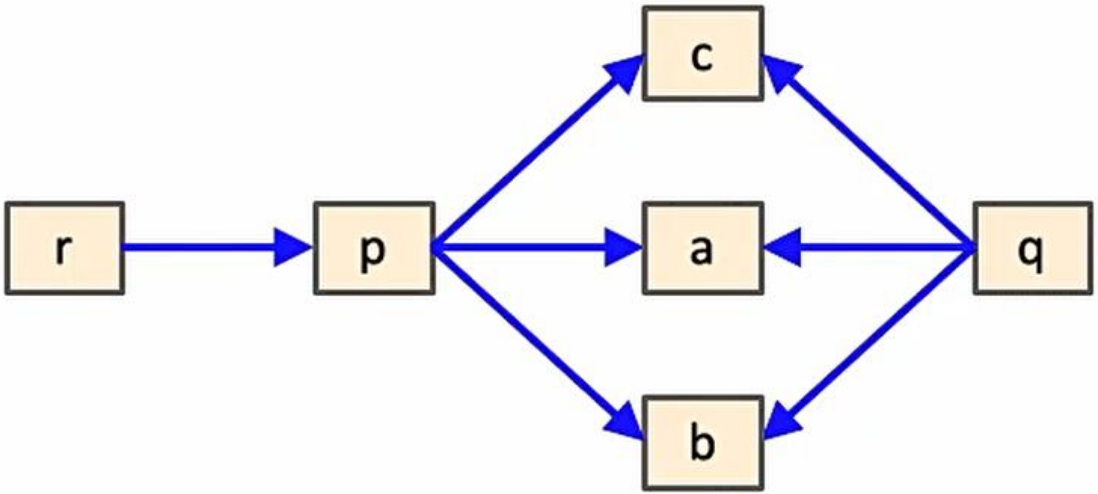
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



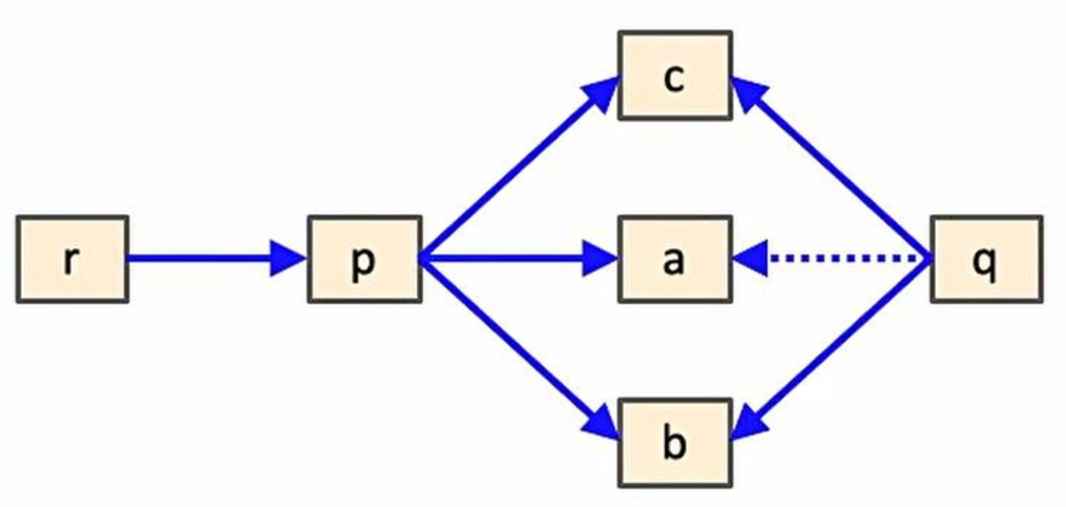
Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



Stack-Based Pointer Analysis Example

```
p = &a;
q = &b;
p = q;
r = &p;
*r = &c;
q = *r;
```



Imprecision in Andersen's analysis: **q** never points to **a** in a concrete execution.

Static Analysis Tools

- LLVM
  - To convert a program into a language-independent intermediate representation (IR)
  - Def-Use & Use-Def[1)]

- SVF[2)]
  - Analysis tool for LLVM-based languages
  - Pointer alias analysis
  - Memory SSA form construction
  - Data value-flow tracking

1) https://labs.engineering.asu.edu/mps-lab/resources/llvm-resources/llvm-def-use-use-def-chains/
2) https://github.com/SVF-tools/SVF
   https://github.com/SVF-tools/SVF-Teaching

Outline

- Intro
- Terminology
- Static Analysis
- Dynamic Analysis

Dynamic Analysis

- Gcov
  - Measure code coverage in a program

- Dynamic symbolic execution
  - Automatically generating tests to achieve higher levels of coverage in a program

Gcov

```c
int foo (int a) {
  return a+10;
}

int main (void) {
  int i=2, j=1, k=0;

  if (i < j) {
    k = foo (i);
    printf("Result:%d\n", k);
  }

  printf("Result:%d\n", k);
  return 1;
}
```

<test.c>

a.out: An instrumented executable file

-ftest-coverage: Adds instructions for counting the number of times individual lines are executed

-fprofile-arcs: Branch instrumentation records how frequently different paths are taken through 'if' statements and other conditionals.

```
hskim@ubuntu:~/gcov$ gcc -Wall -fprofile-arcs -ftest-coverage test.c
hskim@ubuntu:~/gcov$ ls
a.out  test.c  test.gcno
hskim@ubuntu:~/gcov$
```

Gcov

```c
int foo (int a) {
  return a+10;
}

int main (void) {
  int i=2, j=1, k=0;

  if (i < j) {
    k = foo (i);
    printf("Result:%d\n", k);
  }

  printf("Result:%d\n", k);
  return 1;
}
```

<test.c>

```
hskim@ubuntu:~/gcov$ ./a.out
Result:0
hskim@ubuntu:~/gcov$ gcov test.c
File 'test.c'
Lines executed:55.56% of 9
Creating 'test.c.gcov'

hskim@ubuntu:~/gcov$ ls
a.out  test.c  test.c.gcov  test.gcda  test.gcno
```

The **gcov** command produces an annotated version of the original source file, with the file extension '.gcov', containing counts of the number of times each line was executed.

Gcov

```c
int foo (int a) {
  return a+10;
}

int main (void) {
  int i=2, j=1, k=0;

  if (i < j) {
    k = foo (i);
    printf("Result:%d\n", k);
  }

  printf("Result:%d\n", k);
  return 1;
}
```

<test.c>

```
        -:    0:Source:test.c
        -:    0:Graph:test.gcno
        -:    0:Data:test.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
    #####:    3:int foo (int a) {
    #####:    4:  return a+10;
        -:    5:}
        -:    6:
        1:    7:int main (void) {
        1:    8:  int i=2, j=1, k=0;
        -:    9:
        1:   10:  if (i < j) {
    #####:   11:    k = foo (i);
    #####:   12:    printf("Result:%d\n", k);
        -:   13:  }
        -:   14:
        1:   15:  printf("Result:%d\n", k);
        1:   16:  return 1;
        -:   17:}
        -:   18:
```

Examples of Gcov Usages

- Why is Gcov Useful?
  - Identify which code test cases cover
  - Identify inputs to trigger a specific code snippet

- ArduPilot
  - https://firmware.ardupilot.org/coverage/

- PX4
  - https://coveralls.io/github/PX4/Firmware

Existing Approach

- Random Testing
  - Generate random inputs
  - Execute the program on those (concrete) inputs

```
void test_me (int x) {

    if (x == 94389) {
        // Buggy code
    }
}
```

Probability of finding the buggy code: $1/2^{32}$ = 0.0000000023%

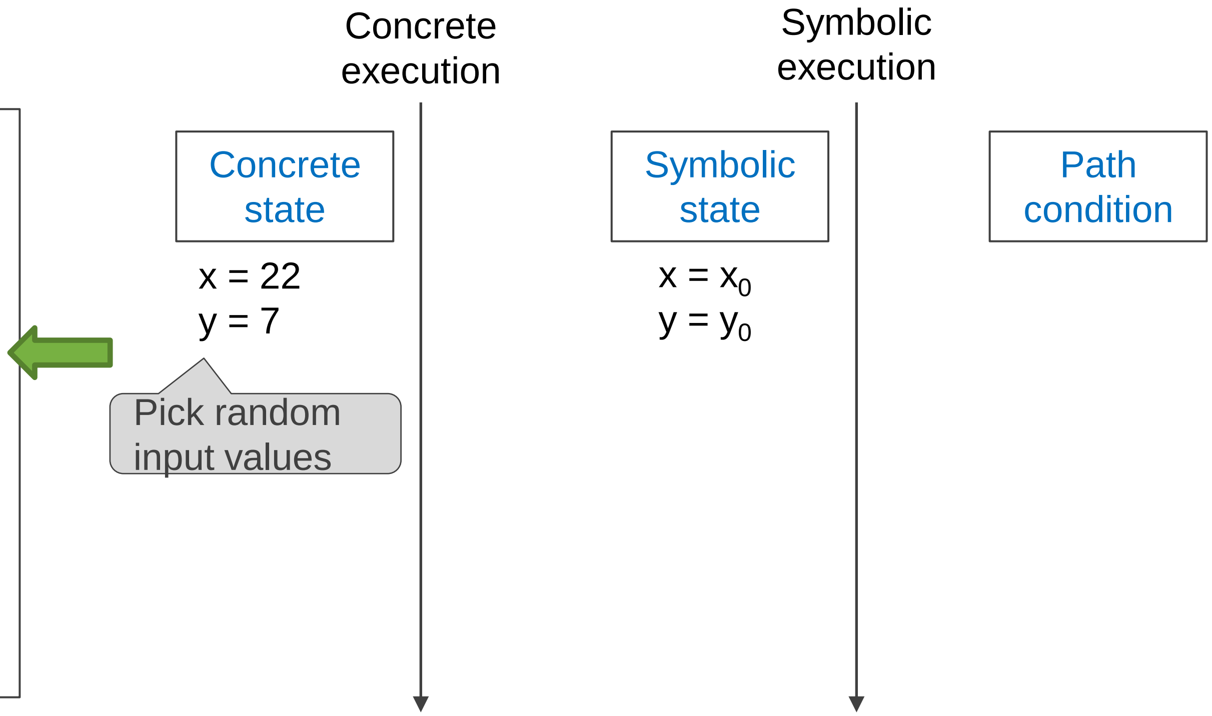
PURDUE UNIVERSITY CERIAS PurSecLab

Dynamic Symbolic Execution (DSE)

- DSE
  - Pick random input values
  - Keep track of both concrete values and symbolic constraints
  - Use concrete values to simplify symbolic constraints

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```



DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state	Symbolic state	Path condition
x = 22 y = 7 z = 14	$x = x_0$ $y = y_0$ $z = 2*y_0$	

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state

x = 22
y = 7
z = 14

Symbolic state

$x = x_0$
$y = y_0$
$z = 2*y_0$

Path condition

$2*y_0 != x_0$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state	Symbolic state	Path condition
x = 22 y = 7 z = 14	$x = x_0$ $y = y_0$ $z = 2*y_0$	$2*y_0 != x_0$

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2$, $y_0 = 1$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
       if (x > y+10)
           // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state

$x = 2$
$y = 1$

Values obtained from a constraint solver

Symbolic state

$x = x_0$
$y = y_0$

Path condition

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
       if (x > y+10)
           // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state	Symbolic state	Path condition
x = 2 y = 1 z = 2	$x = x_0$ $y = y_0$ $z = 2*y_0$	

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state	Symbolic state	Path condition
x = 2 y = 1 z = 2	$x = x_0$ $y = y_0$ $z = 2*y_0$	$2*y_0 == x_0$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution | Symbolic execution

Concrete state

$x = 2$
$y = 1$
$z = 2$

Symbolic state

$x = x_0$
$y = y_0$
$z = 2*y_0$

Path condition

$2*y_0 == x_0$
$x_0 <= y_0+10$

Solve: $(2*y_0 == x_0)$ and $(x_0 > y_0+10)$

Solution: $x_0 = 30$, $y_0 = 15$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution		Symbolic execution	
Concrete state		Symbolic state	Path condition
x = 30 y = 15		$x = x_0$ $y = y_0$	

Values obtained from a constraint solver

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);
   
   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state

x = 30
y = 15
z = 30

Symbolic state

$x = x_0$
$y = y_0$
$z = 2*y_0$

Path condition

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state

x = 30
y = 15
z = 30

Symbolic state

$x = x_0$
$y = y_0$
$z = 2*y_0$

Path condition

$2*y_0 == x_0$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Concrete execution

Symbolic execution

Concrete state

x = 30
y = 15
z = 30

Symbolic state

$x = x_0$
$y = y_0$
$z = 2*y_0$

Path condition

$2*y_0 == x_0$
$x_0 > y_0+10$

DSE example

```
int foo (int v) {
   return 2*v;
}

void test_me (int x, int y) {
   int z = foo (y);

   if (z == x)
      if (x > y+10)
         // Buggy code
}
```

Finally trigger the buggy code!

Concrete execution		Symbolic execution	
Concrete state		**Symbolic state**	**Path condition**
x = 30 y = 15 z = 30		$x = x_0$ $y = y_0$ $z = 2*y_0$	$2*y_0 == x_0$ $x_0 > y_0+10$

- Concrete execution guided symbolic execution
- Symbolic execution guided generation of concrete inputs (increases program code coverage)

Why is DSE Useful?

- Problem
  - You want to develop a tool that automatically tests patched code lines.
  - 'Test' means that you need to trigger the code lines.
  - How?

You can get inputs (e.g., x, y, and i), which trigger the patched code, from a dynamic symbolic execution.

```
void test_me (int x, int y, int i, int j, int k, int l) {
  …
  if (k == l) {
    if (x == y && i == j) {
      // Patched code
    } } }
```

Symbolic Execution Tools

- KLEE[1)]
  - Built on top of the LLVM compiler infrastructure
- angr[2)]
  - Static and dynamic symbolic analysis for binaries

1) https://klee.github.io/tutorials/
2) https://angr.io/

Summary

- Program analysis
  - Is useful to understand behaviors of programs
    - Find inputs that change the RV's altitude state (Def-use)
    - Find if statements that prevent divide-by-zero bugs (Def-use)
    - Identify which code test cases cover (Gcov)
    - Identify inputs to trigger a specific code snippet (DSE)

Summary

- What are the next steps?
  - More understanding about program analysis
    - https://www.youtube.com/watch?v=v0dKdfmziHs&t=1578s
  - Dive into static analysis
    - https://github.com/SVF-tools/SVF-Teaching
  - Dive into symbolic execution
    - https://klee.github.io/tutorials/

Thank you! Questions?

kim2956@purdue.edu

PURDUE UNIVERSITY

CERIAS

PurSecLab