

Wanted: Students to participate in a user study

➤ Requirements:

- Know how to use the Eclipse IDE
- Knowledge in Java development
- Knowledge of static analysis is not required, but it is a plus

➤ Time: 2-3 hours

➤ Interested? Contact Lisa Nguyen: lisa.nguyen@cased.de

Projects

Goal: Perform a security analysis of an open source project

Organization

- Group: 2 to 3 students
- Choose an open source software
- Choose whether to do a **code analysis** or **penetration testing** and the tools to use
- Send ½ page project proposal by Dec 4 and wait for confirmation
- Repost submission deadline March 1, 2016
- Work on your own—no advising
- No recommended size for the report

Projects

Report structure

- Description of the architecture
- Threat analysis
- Performed testing/ code analysis
- Review of the results

Grading criteria (20% of total score)

- Importance of the software
- Thorough threat analysis
- Discovered vulnerabilities
- Innovation
- Quality of the report

Research Activity

Goal: Address a research question related to one of the topics of the course

Paper size

- 5 pages for group of 2 students
- 7 pages for group of 3 students
- Paper Format: IEEE Conference Style (2 columns)

Organization

- Send ½ page project proposal by Dec 11 and wait for confirmation
 - Group members
 - Research problem and approach
- Submission deadline is March 20
- Work on your own—no advising

Lecture 6

Static Code Analysis

Lisa Nguyen Quang Do



TECHNISCHE
UNIVERSITÄT
DARMSTADT

20 November 2015



Summary

- Introduction to static analysis
 - Static and dynamic analysis
 - Basics of static analysis
- Data-flow analysis
 - Intra-procedural analysis
 - Inter-procedural analysis
- Pointer analysis
 - Simple points-to analysis
 - Field-sensitive points-to analysis
- Taint analysis
 - Simple taint analysis
 - Taint analysis and alias information
- Static analysis in practice
 - Examples
 - In industry

Testing programs

➤ Testing

- Functional testing: make sure everything behaves as it should
- Security testing: make sure nothing behaves as it shouldn't

➤ Methods

- Static analysis (white box approach)
- Dynamic analysis (black box approach)

Static analysis (white box approach)

- Inspect the source code and try to predict malicious behaviour
 - Requires extensive knowledge
 - Software skills: software engineering, language specifics
 - operating systems, databases, networking, etc.
 - Awareness: known vulnerabilities, latest attacks
 - Requires creativity
 - Imagining possible threats, crafting attacks
- Will report every possible scenario, even unlikely
- Best done with automated tools
 - HP Fortify, IBM AppScan, Coverity, etc.



Dynamic analysis (black box approach)

- Inspect the running program, without access to the source code
 - Passive: observing traffic
 - Active: Interacting with the program to find and confirm issues
 - Achieved with different levels of coverage (authenticated or not, different privileges, right to use social engineering, etc.)
 - Ex: Stress tests, penetration testing
- Can miss some issues (code coverage is not total)
- Can be automated
 - Burp, HP WebInspect, etc.
 - But automated tools cannot reason on very complex exploits yet.



Static and dynamic analysis

➤ Static analysis

- Scans the whole code base
- Reports lots of warnings
- Finds coding issues (Dead code, null pointer dereference, hardcoded credentials, unchecked errors, API mishandling, etc.)

➤ Dynamic analysis

- Does not cover the whole attack surface
- Can miss vulnerabilities
- Finds runtime and platform-specific issues (Environment configuration, unpatched versions, runtime privileges, issues in 3rd party programs, etc.)

➤ Both find common vulnerabilities

- SQL injection, XSS, Buffer overflows, etc.

Basics of static analysis

- Does a program P satisfy the property Φ ?
 - Is this variable a constant? -> Constant propagation
 - Is this code reachable? -> Dead code analysis
 - Can this pointer ever be null? -> Nullness analysis
 - Is this file closed at that statement? -> Typestate analysis

- Unfortunately, we cannot write a perfect analysis
 - Rice's theorem: For any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property.

Basics of static analysis

➤ Soundness:

- The analysis over-approximates how the program behaves
- It will report all violations of the property
- But it will also report many false positives

➤ Completeness:

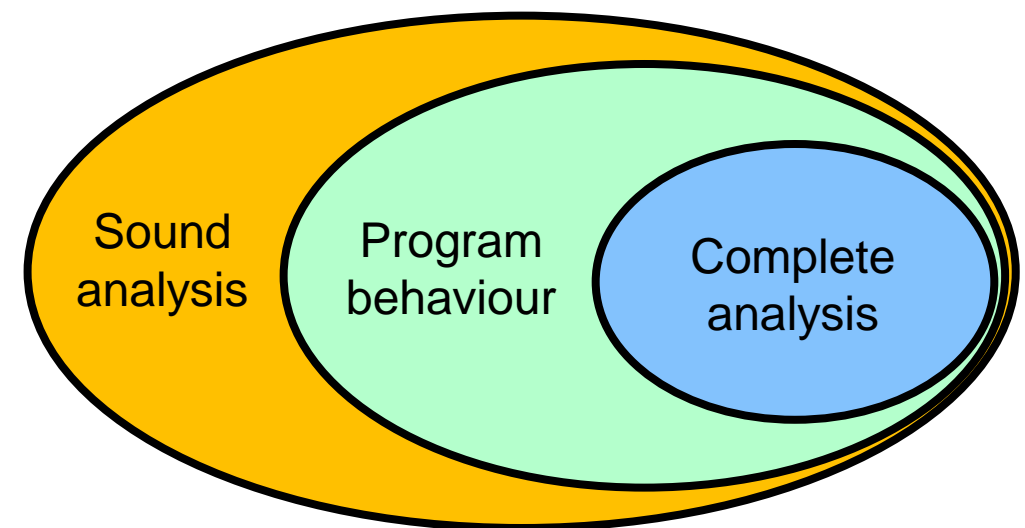
- The analysis under-approximates how the program behaves
- When it reports a violation of the property, it is guaranteed to be correct
- But it will miss some violations

Basics of static analysis

➤ Example: A non field-sensitive analysis

- The analysis manipulates base objects, but cannot model fields

```
a.f = getUserInput();  
writeToDatabase(a.f);  
writeToDatabase(a.g);
```



➤ A sound analysis:

- Would over-approximate a.f to a, and consider a (and all its fields) dangerous
- It would then report both a.f and a.g as dangerous database writes

➤ A complete analysis:

- Would under-approximate a.f, and not consider anything dangerous
- It would report neither a.f nor a.g

Basics of static analysis

➤ Precision:

- In practice, no analysis is totally sound or totally complete. They try to model the program's behaviour with as much precision as possible

➤ In practice, the tradeoff is between precision and scalability

- The more precision one wants to add, the more complex the program model is
- Static analysis needs to model the whole code base

Data-flow analysis

➤ Data-flow analysis

- Is a static analysis method
- It iterates through program points and collects information about a property of the program until a fixed point is reached.

➤ Fixed point

- For each program point, the collected information will not change anymore.

Data-flow analysis

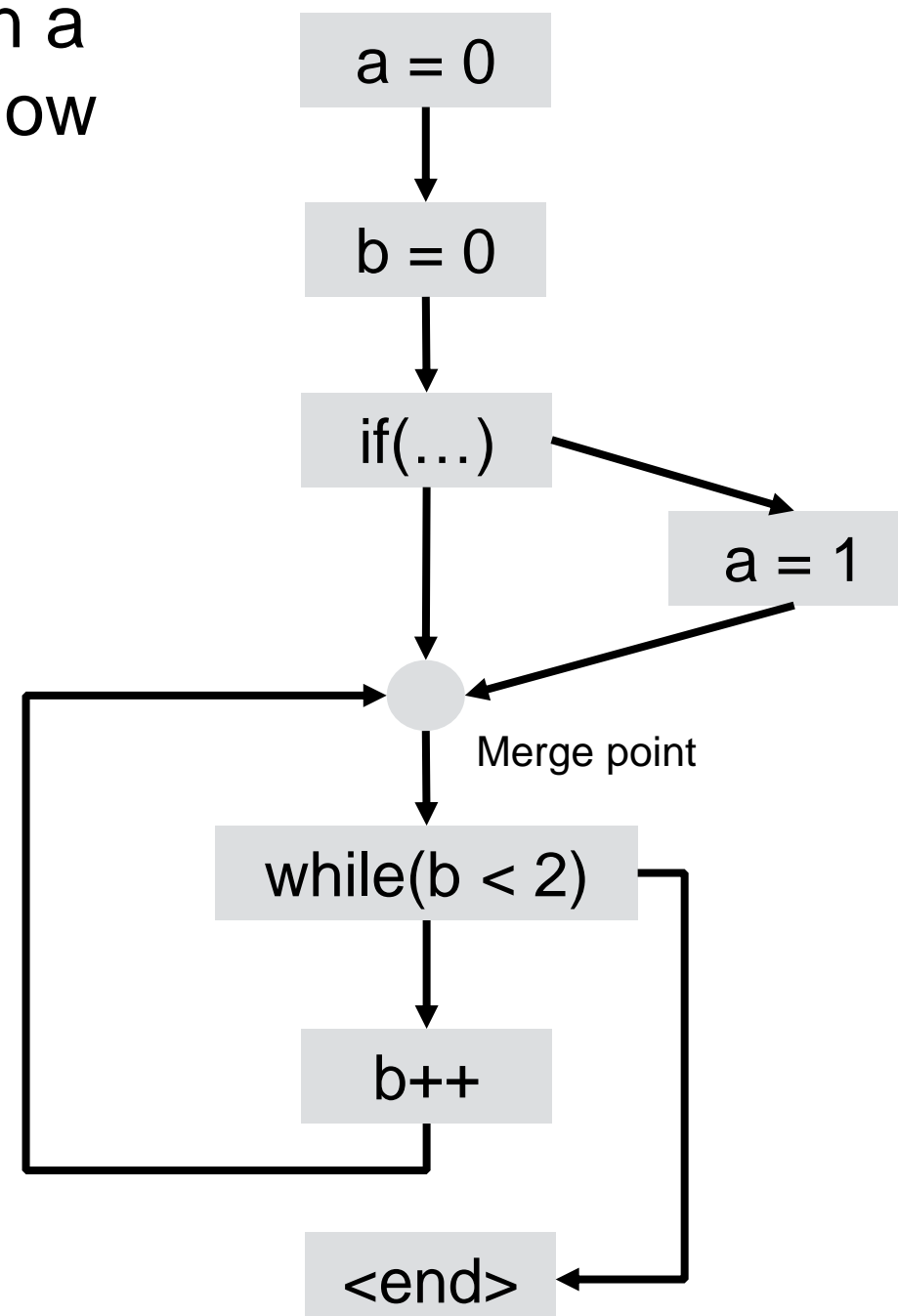
- Example: range analysis
 - Property: what are the values of the variables in this program?

```
int a = 0;  
int b = 0;  
if(...)   
    a = 1;  
while(b < 2){  
    b++;  
}
```


Intra-procedural analysis

- Intra-procedural analyses (within a method) operate on a Control Flow Graph (CFG)

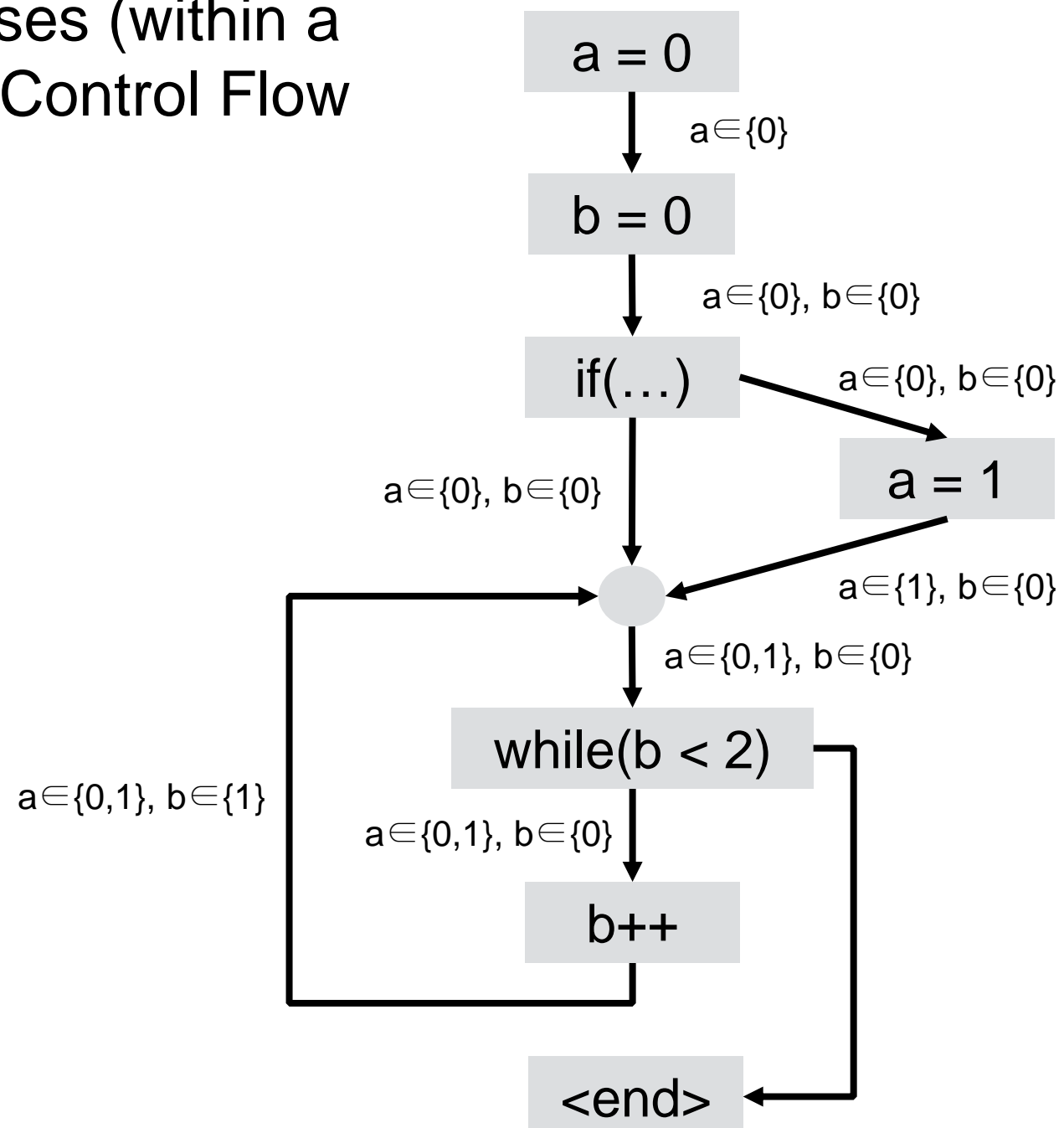
```
int a = 0;  
int b = 0;  
if(...)  
    a = 1;  
while(b < 2){  
    b++;  
}
```



Intra-procedural analysis

- Intra-procedural analyses (within a method) operate on a Control Flow Graph (CFG)

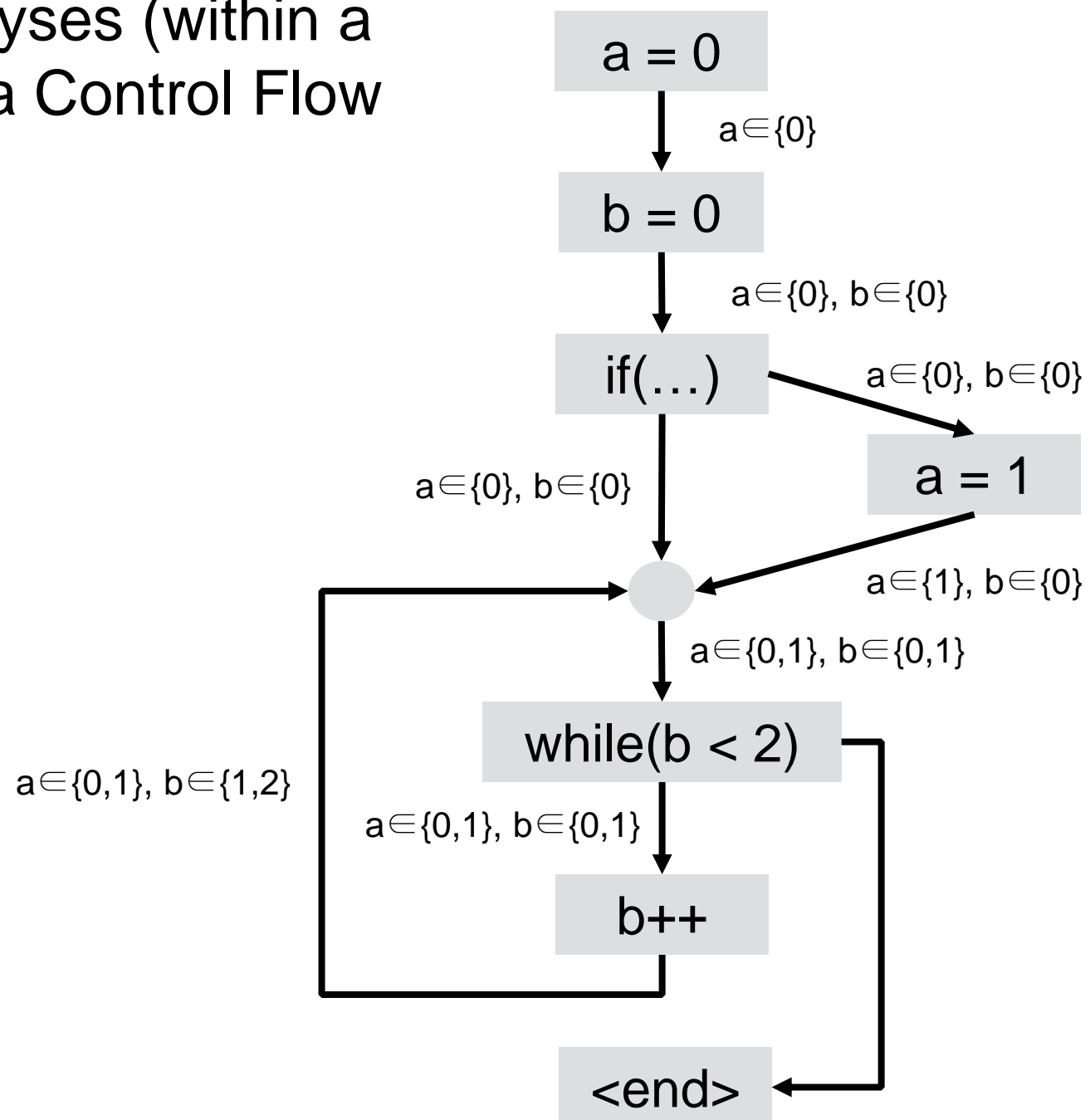
```
int a = 0;  
int b = 0;  
if(...)  
    a = 1;  
while(b < 2){  
    b++;  
}
```



Intra-procedural analysis

- Intra-procedural analyses (within a method) operate on a Control Flow Graph (CFG)

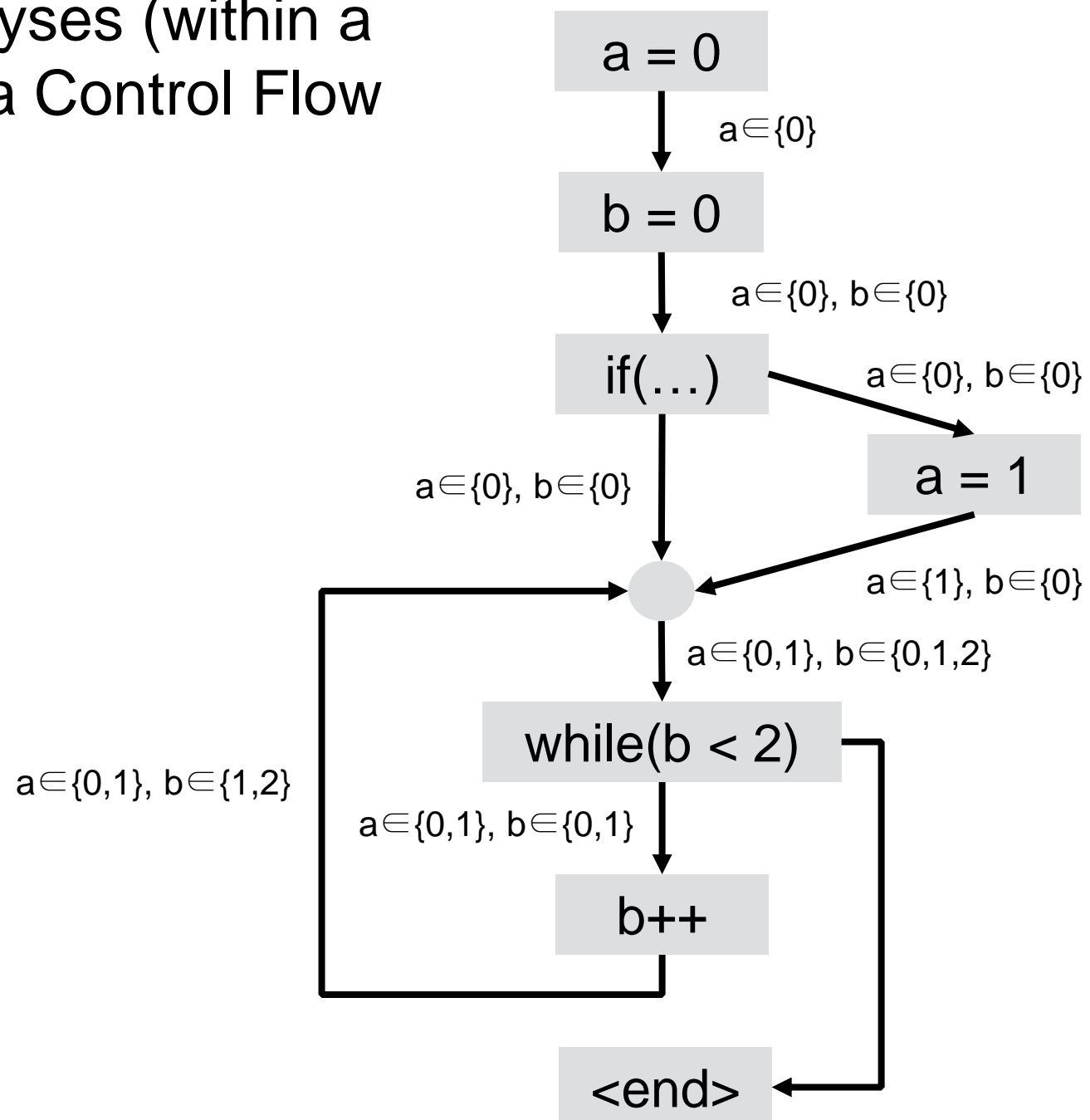
```
int a = 0;  
int b = 0;  
if(...)  
    a = 1;  
while(b < 2){  
    b++;  
}
```



Intra-procedural analysis

- Intra-procedural analyses (within a method) operate on a Control Flow Graph (CFG)

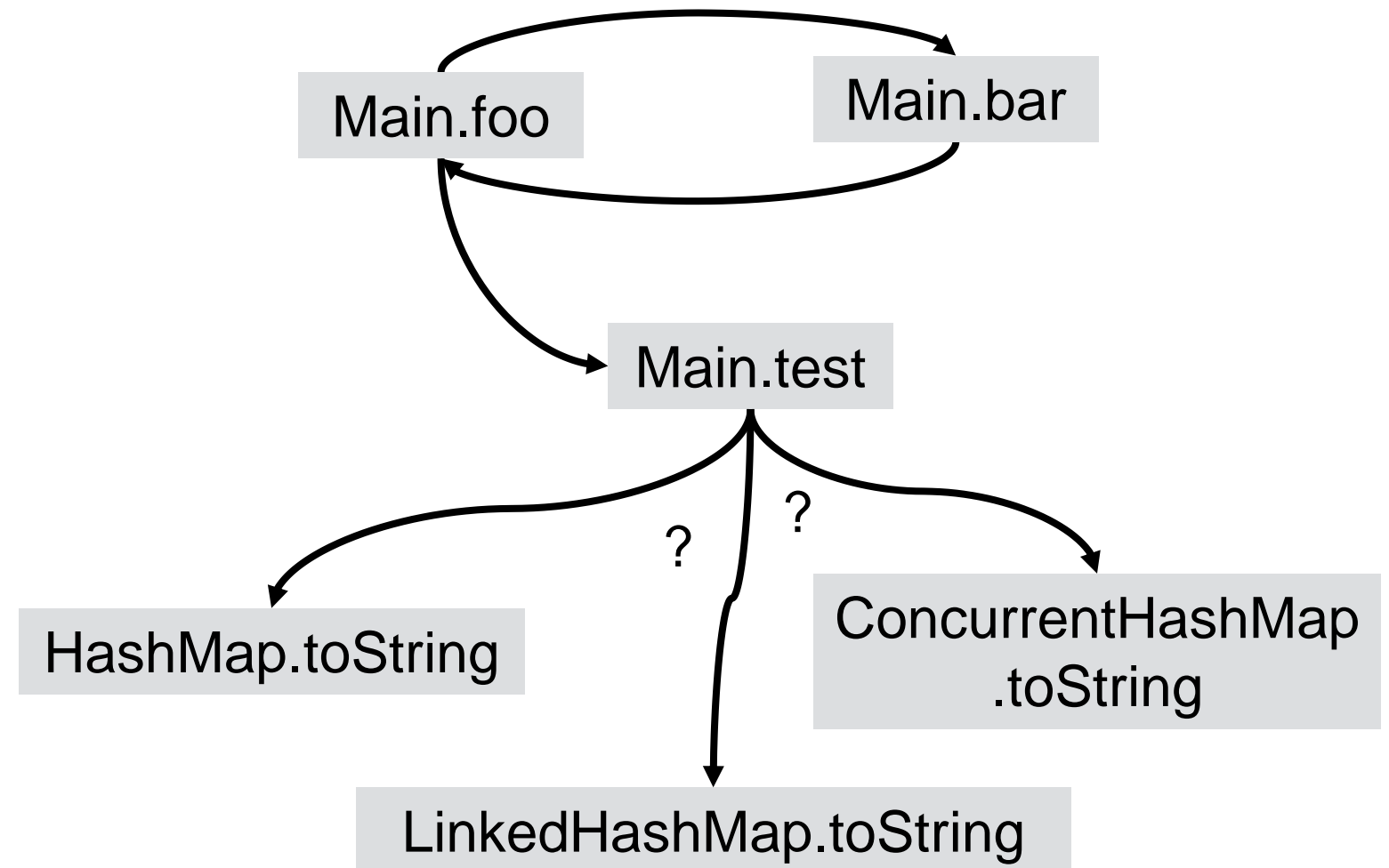
```
int a = 0;
int b = 0;
if(...)
    a = 1;
while(b < 2){
    b++;
}
```



Inter-procedural analysis

- The call-graph's precision influences the analysis' workload and precision

```
public foo(){  
    ...  
    bar();  
    test(new HashMap());  
    ...  
}  
  
public bar(){  
    foo();  
    ...  
}  
  
public test(Map m){  
    m.toString();  
}
```



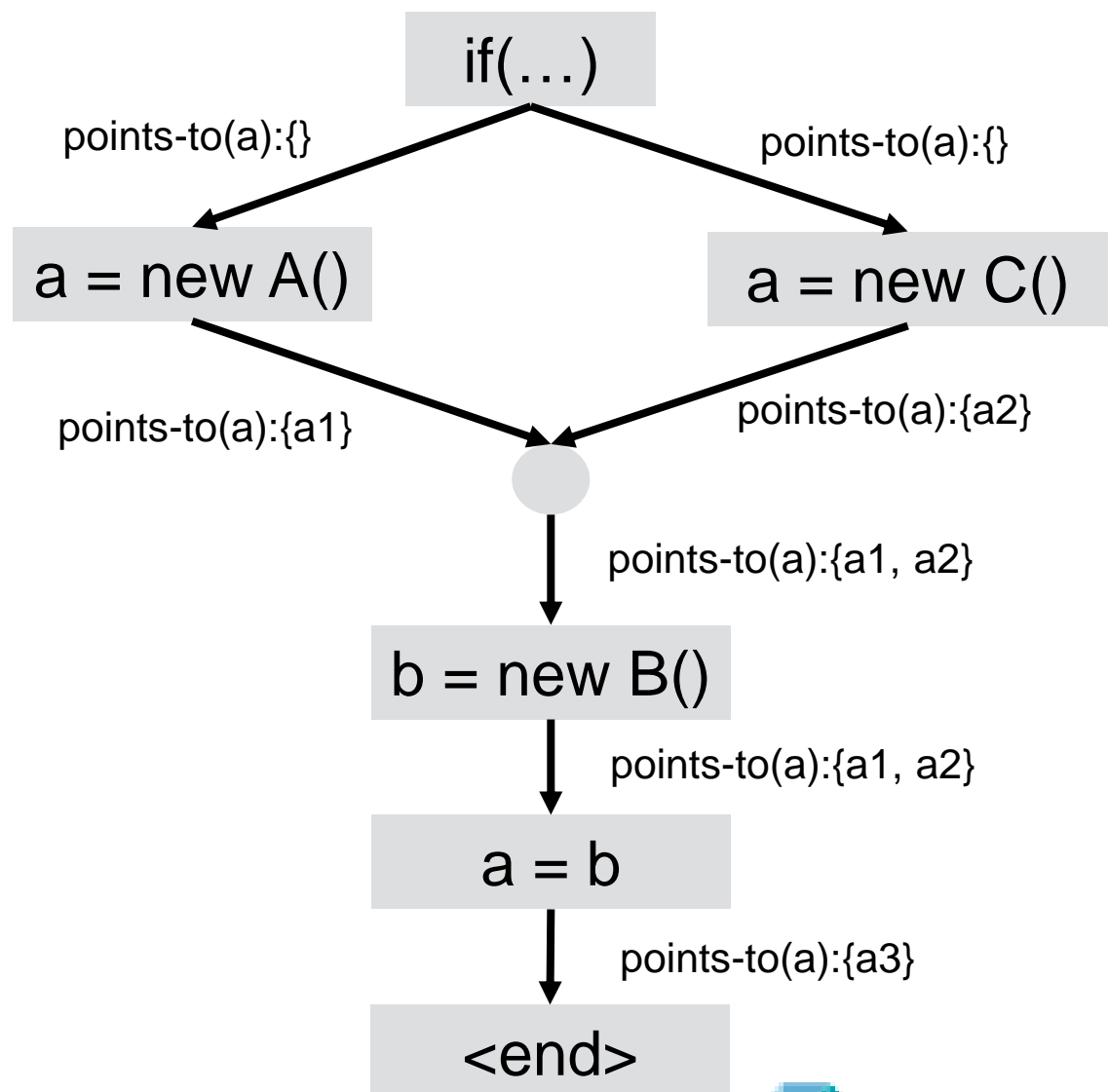
Pointer analysis

- Reasons about allocation sites / memory locations
- Two types of pointer analysis
 - Points-to analysis
 - Where are the allocation sites of a ?
 - $\text{points-to}(a) = \{a1, a2\}$
 - Alias analysis
 - Do a and b refer to the same object?
 - $\text{alias}(a,b) = \text{true/false}$

Simple points-to analysis

- Property: what is the points-to set of `a` in this program?

```
if(...)  
a1:   a = new A();  
else  
a2:   a = new C();  
a3:   b = new B();  
      a = b;
```



Simple points-to analysis

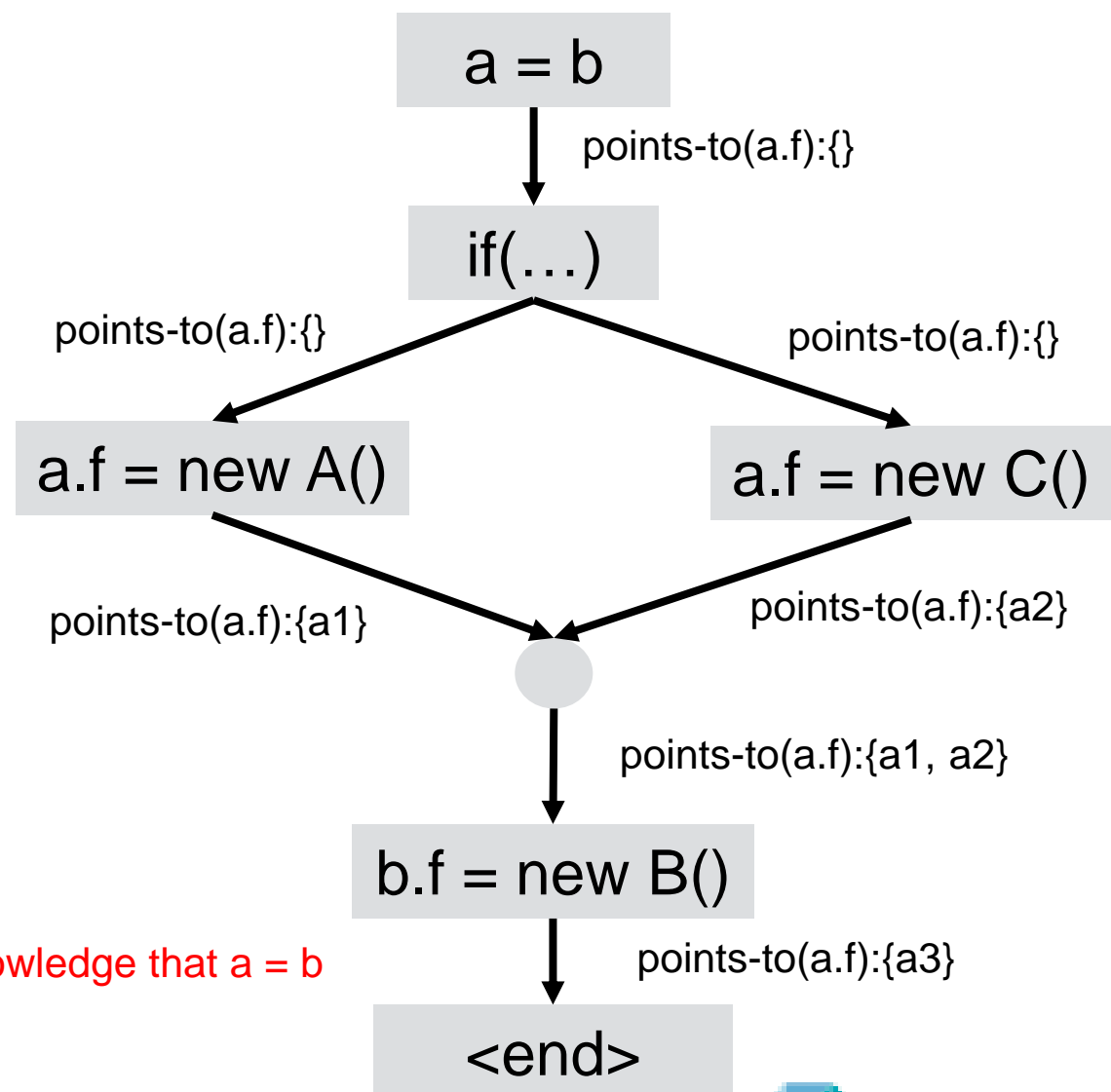
- More precision?
 - Field sensitivity:
 - In some languages, the analysis needs to model object fields as well.

Points-to analysis with access paths

- Property: what is the points-to set of `a.f` in this program?

```
a = b;  
if(...)  
a1:   a.f = new A();  
else  
a2:   a.f = new C();  
a3:   b.f = new B();
```

Must have knowledge that `a = b`



Pointer analysis

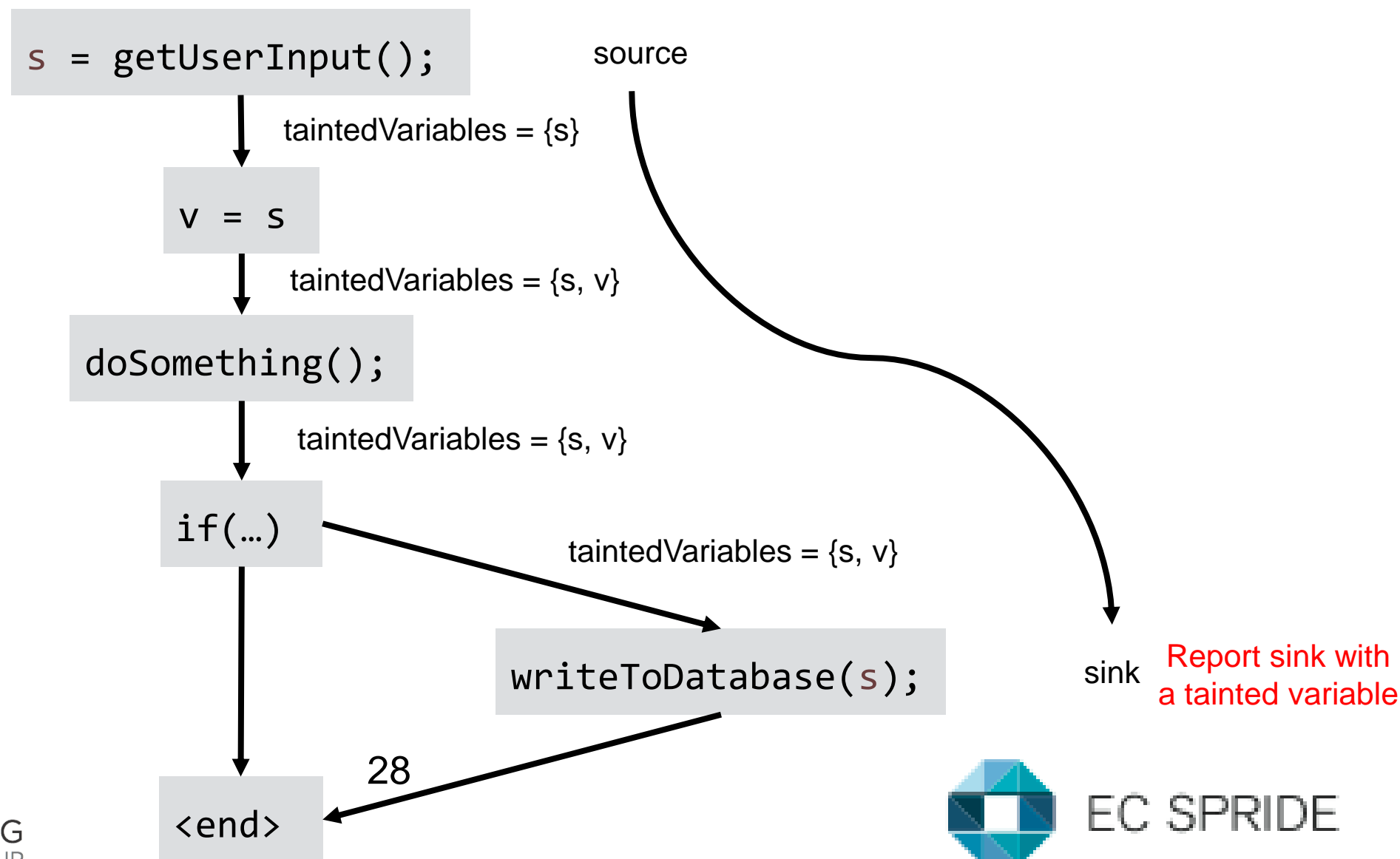
- As soon as field-sensitivity is introduced, pointer analysis becomes trickier
 - Loops, recursion, function returns, field writes, etc. can potentially create/kill an allocation.
- This is also true for any kind of precision introduced in the analysis.

Pointer analysis

- Pointer analyses are often used for:
 - Call-graph construction
 - Type information
 - Strong/weak updates
 - Other client analyses. Ex: taint analysis
- The precision of a pointer analysis directly influences the precision and workload of its client analysis.

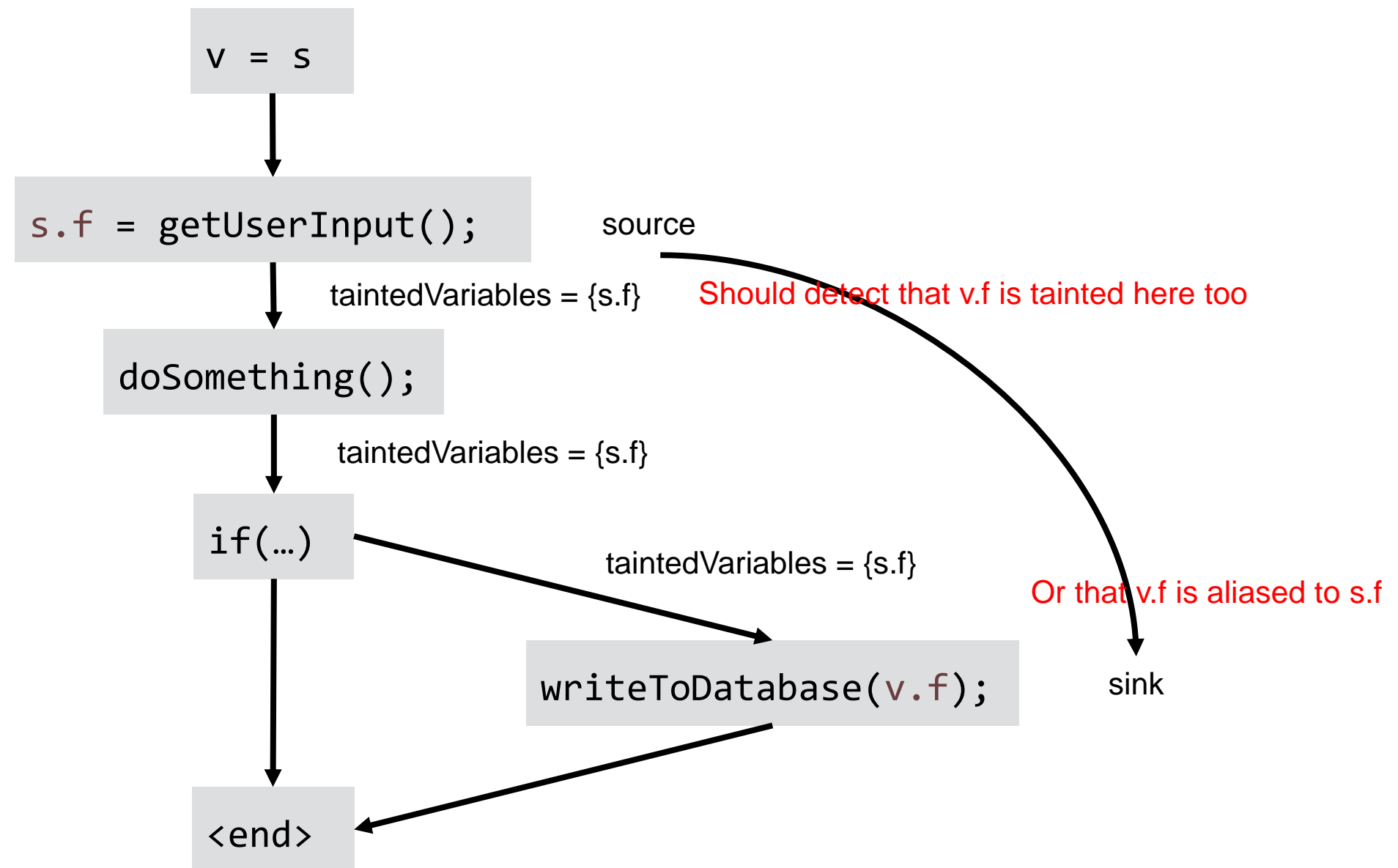
Taint analysis

- Is used to find privacy leaks
 - Matches sources and sinks to find paths of unsanitized data
 - Finds: uses of unvalidated inputs and leaks of private information to the outside world



Taint analysis and alias information

- Alias information can be used to enhance the precision of the analysis



Taint analysis

- To be as precise as possible, an analysis should take various information into account
 - Alias information
 - Language-specific information (ex: Android lifecycle modelling)
 - Analysis-specific information
 - Model sources and sinks (for a taint analysis)
 - Flow, path, context sensitivity
 - Etc.

Example 1

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    try {
        String studentId = request.getParameter("studId");
        Connection conn = null;
        try {
            Class.forName(driver).newInstance();
            conn = DriverManager.getConnection(url + dbName, userName, password);

            Statement st = conn.createStatement();
            String query = 'SELECT * FROM Students where studId=' + studentId + '';
            out.println('Query : ' + query);
            System.out.printf(query);
            ResultSet res = st.executeQuery(query);

            PrintWriter out = response.getWriter();
            while (res.next()) {
                String s = res.getString('classes');
                out.println('\t\t' + s);
            }
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } finally {
        out.close();
    }
}
```

studentId

studentId, query

SQL Injection
Can be found with taint analysis

Example 2

```
int main(void) {  
    char buff[15];  
    printf("\n Enter the password : \n");  
    gets(buff);  
  
    if(strcmp(buff, getPassword())) {  
        printf ("\n Wrong Password \n");  
    } else {  
        printf ("\n Correct Password \n");  
        // doRootStuff();  
    }  
    return 0;  
}
```



Buffer overflow

Can be detected with a taint analysis
but it alone will not be enough
(Will yield some false positives)

Example 3

```
private A getNewHandler() {  
    A a = null;  
    try {  
        a1:    a = new A();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return a.getHandler();  
}
```

nullVariables = {a}

nullVariables = {}

nullVariables = {a}

nullVariables = {a} points-to{a} = {a1, null}

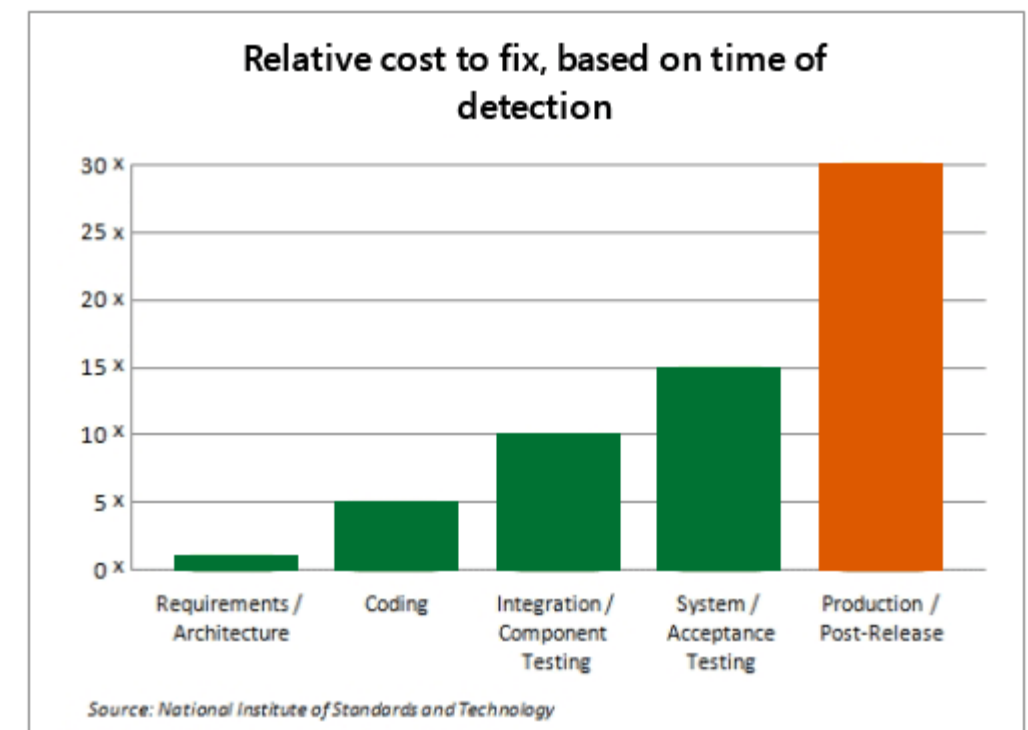
Null pointer dereference
Can be detected with nullness analysis
Or points-to information

Static analysis in practice

- What is it used for?
 - Compilers (optimization, type checking, etc.)
 - Bug and vulnerability finding
 - Formal verification

- For bug finding
 - Increase in companies' use of the tools
 - Find bugs earlier in the software development lifecycle
 - Run on nightly builds
 - Relatively fast
 - Less expensive than manual audits
 - HP Forfify, IBM Appscan, FindBugs...

https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis



Static analysis in practice

- Developer feedback:
 - Takes hours to run
 - Has false positives and false negatives
 - Warnings are hard to understand and to fix
 - Developers are not trained enough to configure and use the tool to its full potential

The screenshot displays the TeamMentor static analysis tool interface. The top-left pane, titled 'SCA Analysis Results - With TeamMentor', shows a 'Security Auditor View' with a filter set. It lists various security issues categorized by severity: Critical (4286), High (2335), Medium (8), Low (1334), and Info (7963). The 'Group By' is set to 'Category'. The main list shows 'Command Injection' as the most frequent issue, with 3 instances. The top-right pane, titled 'Project Summary', shows the source code for 'Exec.java'. The bottom-left pane, titled 'Analysis Evidence', lists the specific code locations where the issues were found, such as 'ParameterParser.java:615 - getParameterValues(return)' and 'Challenge2Screen.java:641 - getRawParameter(return)'. The bottom-right pane, titled 'Issue', provides a detailed description of the 'Exec.java:107 (Command Injection)' issue, explaining that the 'execOptions()' method in 'Exec.java' calls 'exec()' with a command built from untrusted data, which can cause the program to execute malicious commands on behalf of an attacker. It also includes a 'More Information...' link and a 'Recommendations...' link.

<https://www.securityinnovation.com/training/application-security/knowledgebase/team-mentor-plugins.html>

Static analysis in practice

➤ Developer feedback:

- Takes hours to run
 - Tradeoff between precision and performance. Good results take time to compute.
- Has false positives and false negatives
 - Achieving perfect soundness and completeness is not possible, although many efforts are made to improve this
- Warnings are hard to understand and to fix
- Developers are not trained enough to configure and use the tool to its full potential

Just-in-Time analysis

- Design an analysis that ensures:
 - Responsiveness
 - Precision
 - Fix easiness

- Demo

References and further reading

- IT Security Engineering, Prof. Philippe Janson, EPF Lausanne, 2012
- Designing Code Analyses for Large Software Systems, Prof. Eric Bodden, TU Darmstadt, 2015
- Mobile Application Security Through Static and Dynamic Analysis, Nguyen Quang Do, 2014
- DART: Demand-drive Flow, Field and Context-sensitive Points-to Analysis, Späth et al., 2015
- FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, Arzt et al.
- Toward a Just-in-Time Static Analysis, Nguyen Quang Do et al., 2015
- Data Flow Analysis: Theory and practice (Khedker et al.)
- Principles of Program Analysis (Flemming et al.)
- Precise Interprocedural Dataflow Analysis via Graph Reachability, Reps et al., 1995
- https://www.owasp.org/index.php/Static_Code_Analysis