Security Testing

Checking for what shouldn't happen

Azqa Nadeem

PhD Student @ Cyber Security Group

Agenda for today

- Part I
  - Latest security news
  - Security vulnerabilities in Java
  - Types of Security testing
    - SAST vs. DAST
- Part II
  - SAST under the hood
    - Pattern Matching
    - Control Flow Analysis
    - Data Flow Analysis
  - SAST Tools performance

TU Delft

Announcements

- Assignment 2 – Security module
- Exam questions

EEMCS DIVERSITY LUNCHES

E EEMCS D Diversity I Inclusion T Team TU Delft

14th May 2019 12:30h Square, in Pulse. Building 33A →

WOMEN OF COMPUTER SCIENCE

Two former students from Computer Science are going to share their experience at EEMCS and their current career:

- ❑ Valentine Mairet
- ❑ Ginger Geneste

Everyone is welcome to join ☺
Register to get free lunch! QR code:

Agenda for today

- **Part I**
  - **Latest security news**
  - **Security vulnerabilities in Java**
  - **Types of Security testing**
    - **SAST vs. DAST**
- Part II
  - SAST under the hood
    - Pattern Matching
    - Control Flow Analysis
    - Data Flow Analysis
  - SAST Tools performance

TU Delft

Software testing

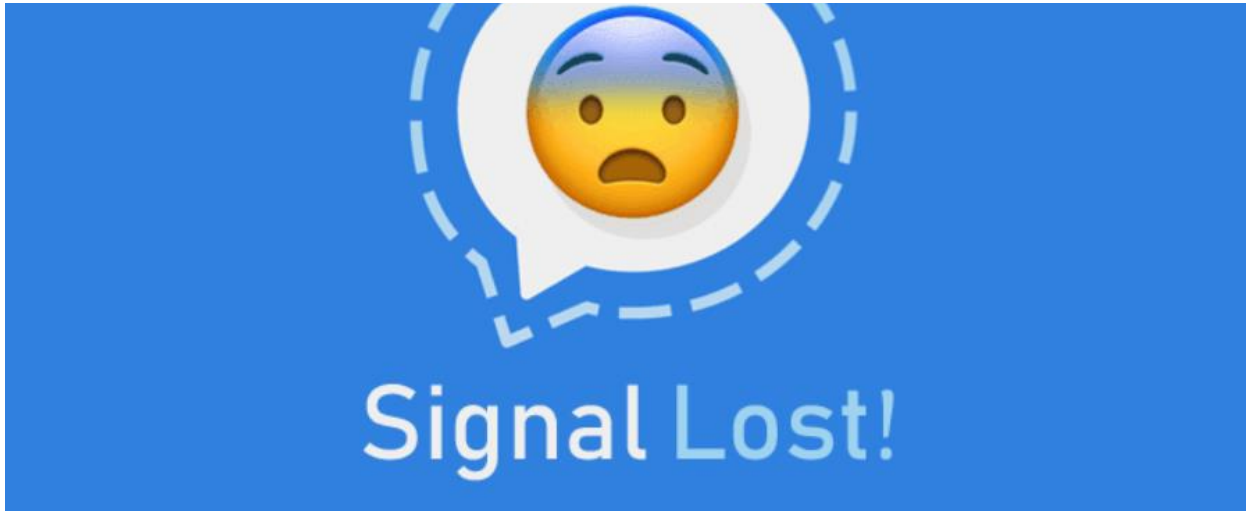
vs.

Security testing

Impact – Stolen chats

Another severe flaw in Signal desktop app lets hackers steal your chats in plaintext

May 16, 2018 Swati Khandelwal

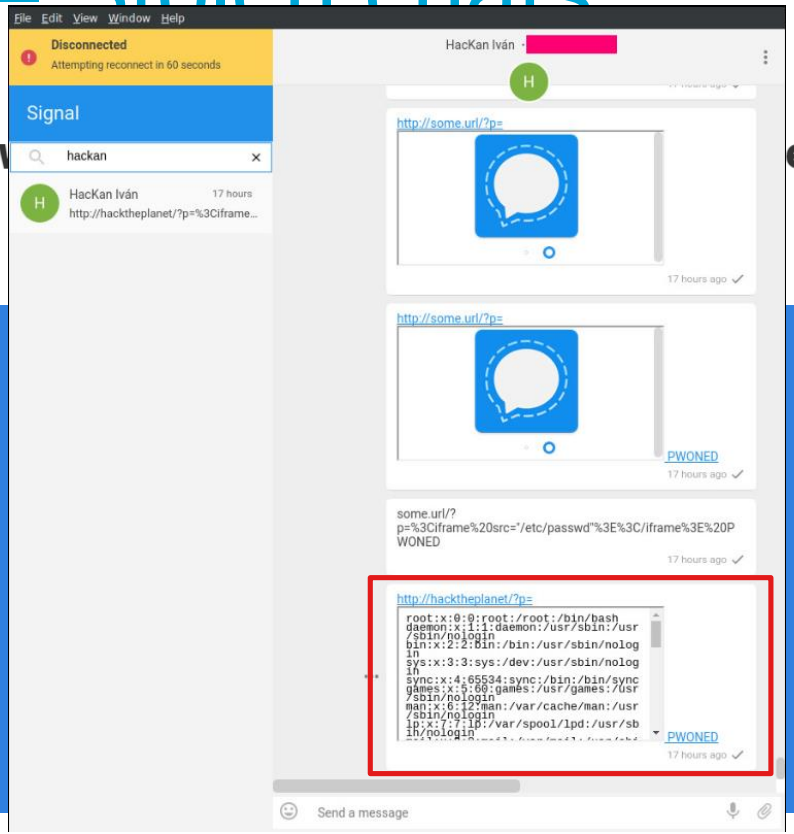


https://ivan.barreraoro.com.ar/signal-desktop-html-tag-injection/

Impact – Stolen chats

Another severe flaw … eal your chats in plaintext

May 16, 2018 Swati Khandelwal

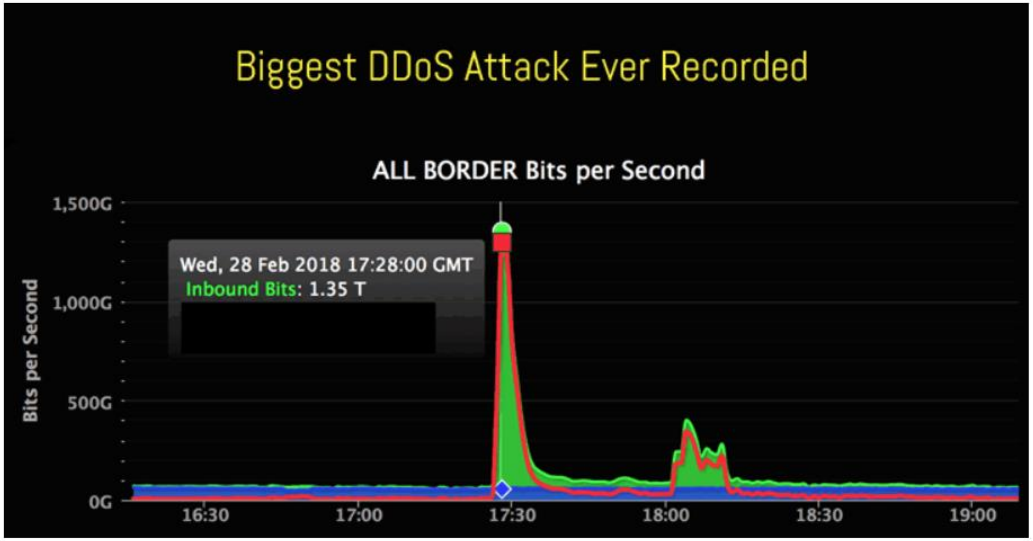


https://ivan.barreraoro.com.ar/signal-desktop-html-tag-injection/

Impact – Github down

Biggest-Ever DDoS Attack (1.35 Tbs) Hits Github Website

March 02, 2018 Mohit Kumar



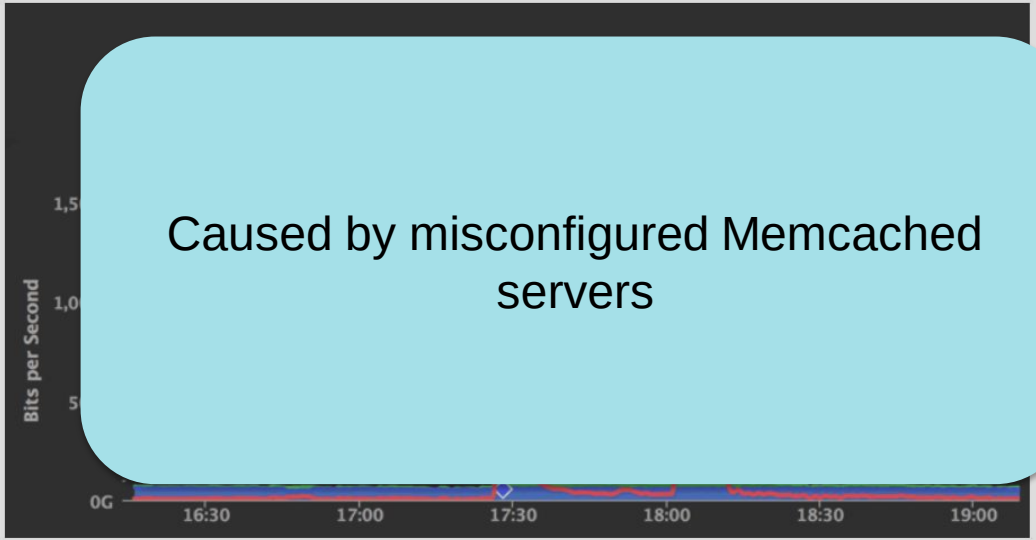
On Wednesday, February 28, 2018, GitHub's code hosting website hit with the largest-ever distributed denial of service (DDoS) attack that peaked at record 1.35 Tbps.

https://thehackernews.com/2018/03/biggest-ddos-attack-github.html

Impact – Github down

Biggest-Ever DDoS Attack (1.35 Tbs) Hits Github Website

March 02, 2018 Mohit Kumar



Caused by misconfigured Memcached servers

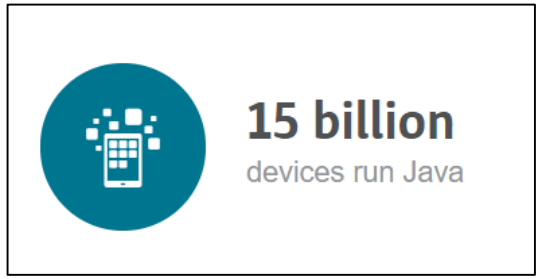
On Wednesday, February 28, 2018, GitHub's code hosting website hit with the largest-ever distributed denial of service (DDoS) attack that peaked at record 1.35 Tbps.

https://thehackernews.com/2018/03/biggest-ddos-attack-github.html

Is Java Secure?

- Secure from memory corruption
- … but not completely

- Potential targets
  - Java Virtual Machine
  - Libraries in native code



https://w3techs.com/technologies/details/pl-java/all/all

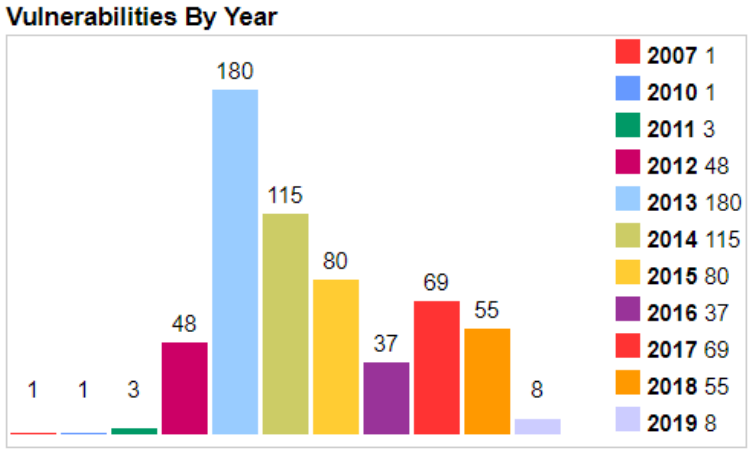
Vulnerability databases

- OWASP Top Ten project
  - Awareness document
  - Web application security

- NIST National Vulnerability Database
  - U.S govt. repository
  - General security flaws

Name	
CVE-2019-9624	Webmin 1.900 allows remote attackers to execute arbitra
CVE-2019-5312	An issue was discovered in weixin-java-tools v3.3.0. Ther
CVE-2019-3801	Cloud Foundry cf-deployment, versions prior to 7.9.0, con the dependency, and inject malicious code into the compo
CVE-2019-2699	Vulnerability in the Java SE component of Oracle Java SE via multiple protocols to compromise Java SE. While the v vulnerability applies to Java deployments, typically in clier rely on the Java sandbox for security. This vulnerability ca and Availability impacts). CVSS Vector: (CVSS:3.0/AV:N/
CVE-2019-2698	Vulnerability in the Java SE component of Oracle Java SE access via multiple protocols to compromise Java SE. Suc Start applications or sandboxed Java applets (in Java SE 8 deployments, typically in servers, that load and run only t (CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H).

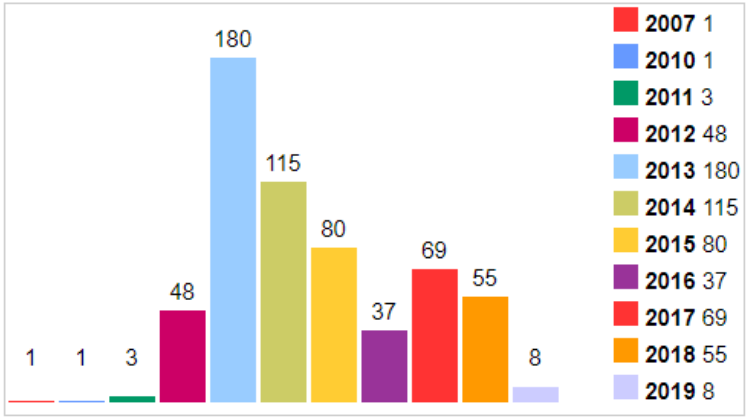
JRE vulnerabilities



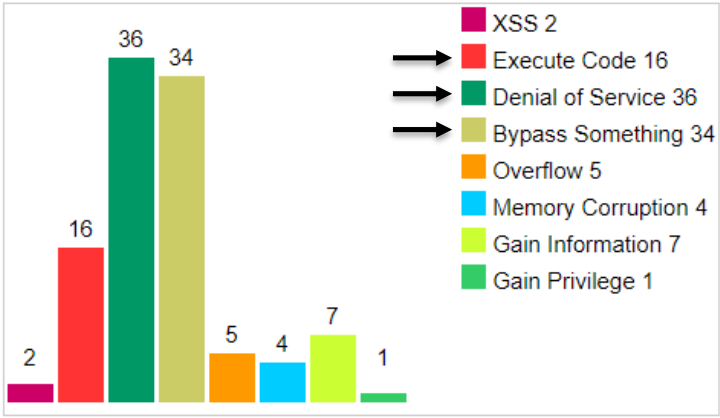
https://www.cvedetails.com/product/19116/Oracle-JDK.html?vendor_id=93

JRE vulnerabilities

Vulnerabilities By Year



Vulnerabilities By Type



https://www.cvedetails.com/product/19116/Oracle-JDK.html?vendor_id=93

Some Examples

What's wrong?

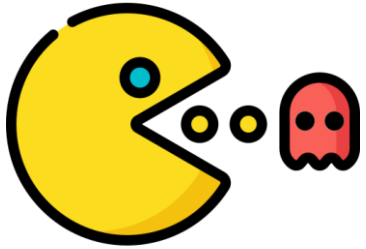
```
Socket socket = null;
BufferedReader readerBuffered = null;
InputStreamReader readerInputStream = null;

/* Read data using an outbound tcp connection */
socket = new Socket("host.example.org", 39544);

/* read input from socket */
readerInputStream = new InputStreamReader(socket.getInputStream(), "UTF-8");
readerBuffered = new BufferedReader(readerInputStream);

/* Read data using an outbound tcp connection */
String data = readerBuffered.readLine();

Class<?> tempClass = Class.forName(data);
Object tempClassObject = tempClass.newInstance();

IO.writeLine(tempClassObject.toString()); /* Use tempClassObject in some way */
```

Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack

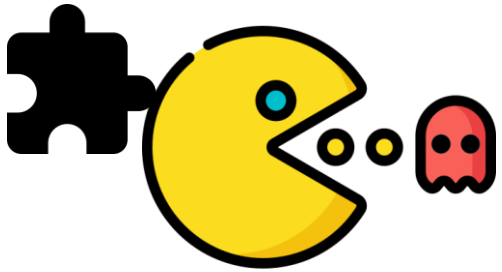
Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack



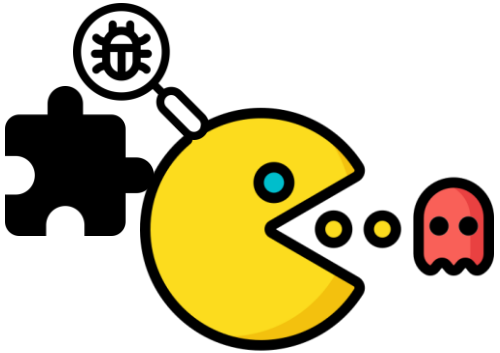
Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack



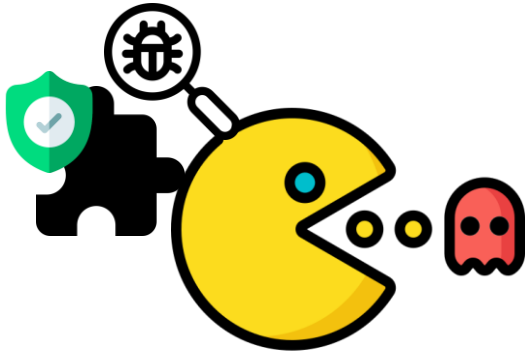
Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack



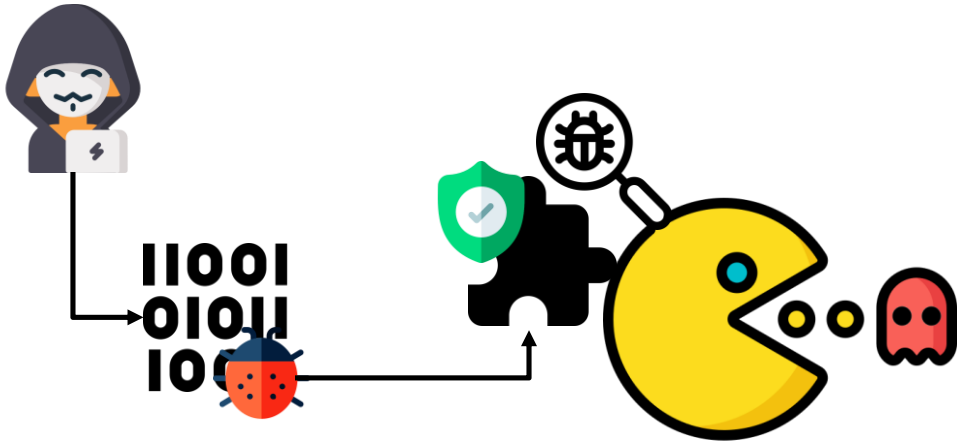
Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack



Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack



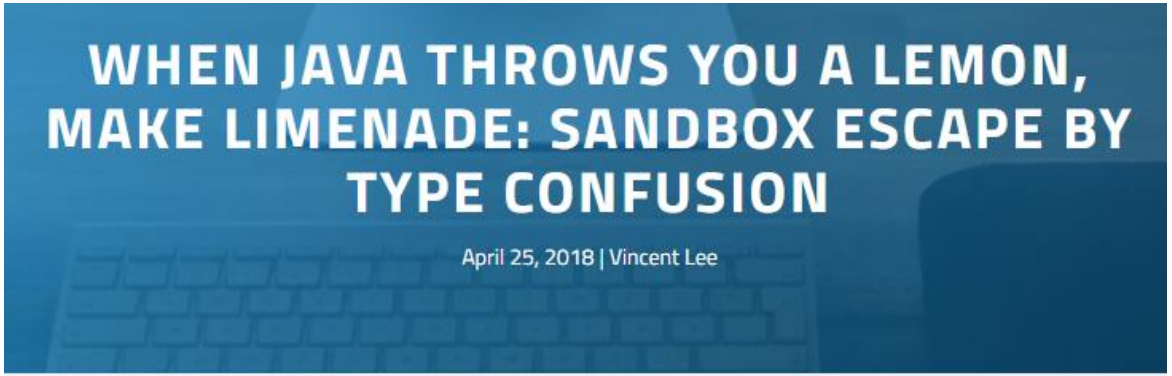
Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack
- Top vulnerability in OWASP Top 10

Code Injection vulnerability

- Execute code in unauthorized applications
- Victim to Update Attack
- Top vulnerability in OWASP Top 10
- Tricky to fix
  - Stop adding plugins
  - Limit privileges

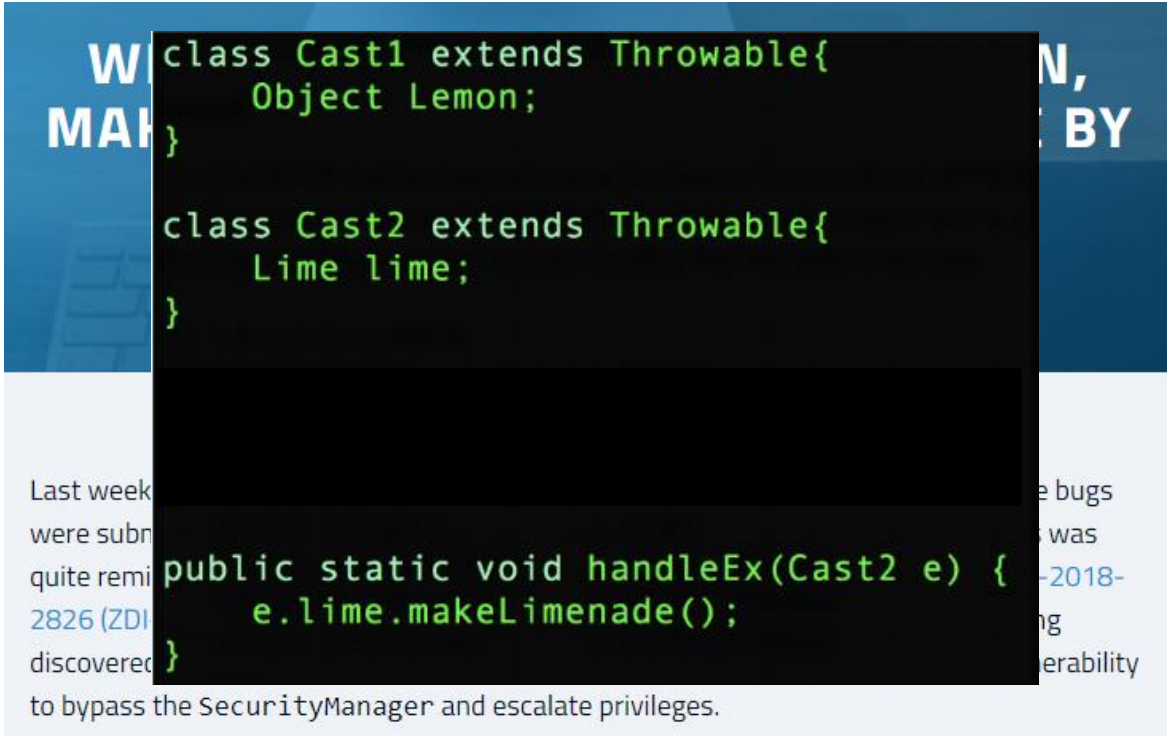
Type confusion vulnerability



Last week, Oracle released their quarterly Critical Patch Update (CPU). Seven of these bugs were submitted through the Zero Day Initiative (ZDI) program, and one of these bugs was quite reminiscent of the Java submissions in late 2012 and early 2013. The bug, CVE-2018-2826 (ZDI-18-307), is a sandbox escape vulnerability due to insufficient type checking discovered by XOR19. An attacker with low execution privileges may exploit this vulnerability to bypass the `SecurityManager` and escalate privileges.

https://www.thezdi.com/blog/2018/4/25/when-java-throws-you-a-lemon-make-limenade-sandbox-escape-by-type-confusion

Type confusion vulnerability



https://www.thezdi.com/blog/2018/4/25/when-java-throws-you-a-lemon-make-limenade-sandbox-escape-by-type-confusion

Bypassing Java Security Manager

- Exploit Type confusion vulnerability

https://access.redhat.com/security/cve/cve-2014-3558

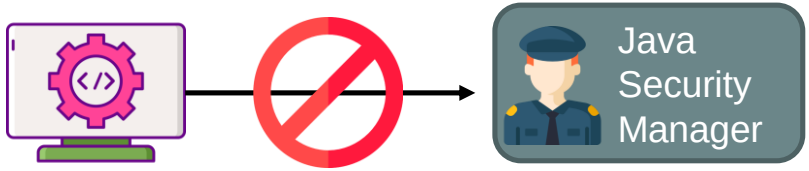
Bypassing Java Security Manager

- Exploit Type confusion vulnerability

https://access.redhat.com/security/cve/cve-2014-3558

Bypassing Java Security Manager

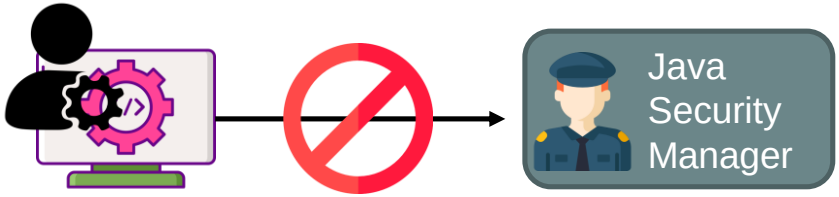
- Exploit Type confusion vulnerability



https://access.redhat.com/security/cve/cve-2014-3558

Bypassing Java Security Manager

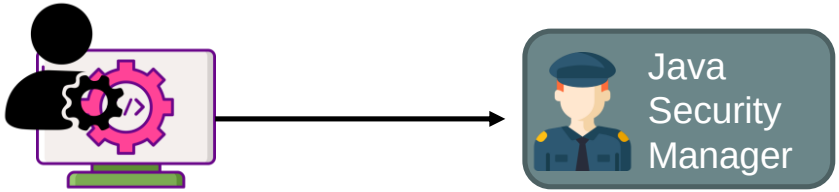
- Exploit Type confusion vulnerability
- Escalated privileges



https://access.redhat.com/security/cve/cve-2014-3558

Bypassing Java Security Manager

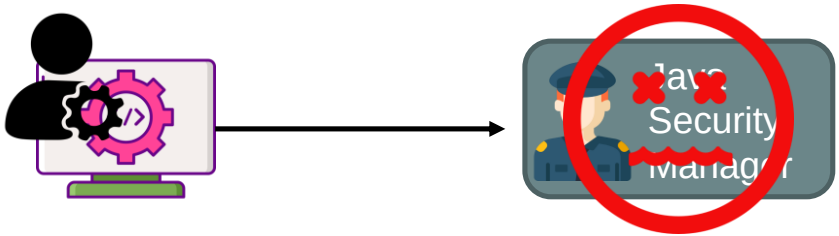
- Exploit Type confusion vulnerability
- Escalated privileges



https://access.redhat.com/security/cve/cve-2014-3558

Bypassing Java Security Manager

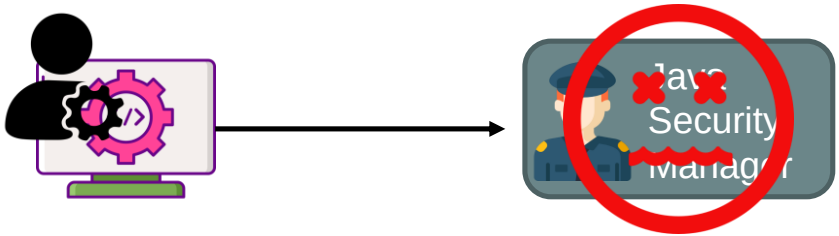
- Exploit Type confusion vulnerability
- Escalated privileges
  - Set JSM to `null`



https://access.redhat.com/security/cve/cve-2014-3558

Bypassing Java Security Manager

- Vulnerable: Hibernate → Reflection helper
- Exploit Type confusion vulnerability
- Escalated privileges
  - Set JSM to `null`



https://access.redhat.com/security/cve/cve-2014-3558

Arbitrary Code Execution (ACE)

- Vulnerable: XStream → Converts XML to Object
- Deserialization vulnerability

https://access.redhat.com/security/cve/cve-2013-7285

Arbitrary Code Execution (ACE)

- Vulnerable: XStream → Converts XML to Object
- Deserialization vulnerability

https://access.redhat.com/security/cve/cve-2013-7285

Arbitrary Code Execution (ACE)

- Vulnerable: XStream → Converts XML to Object
- Deserialization vulnerability

https://access.redhat.com/security/cve/cve-2013-7285

Arbitrary Code Execution (ACE)

- Vulnerable: XStream → Converts XML to Object
- Deserialization vulnerability
  - Via malicious input XML

https://access.redhat.com/security/cve/cve-2013-7285

Arbitrary Code Execution (ACE)

- Vulnerable: XStream → Converts XML to Object
- Deserialization vulnerability
  - Via malicious input XML

https://access.redhat.com/security/cve/cve-2013-7285

Remote Code Execution (RCE)

https://pivotal.io/security/cve-2018-1273

Remote Code Execution (RCE)

https://pivotal.io/security/cve-2018-1273

Remote Code Execution (RCE)

https://pivotal.io/security/cve-2018-1273

Remote Code Execution (RCE)

- Spring Data Commons → DB connections
- Property binder vulnerability
  - Via specially crafted request parameters

https://pivotal.io/security/cve-2018-1273

Oracle April 2018 CPU: Most Java flaws can be remotely exploited

By News | April 18, 2018 | Alerts

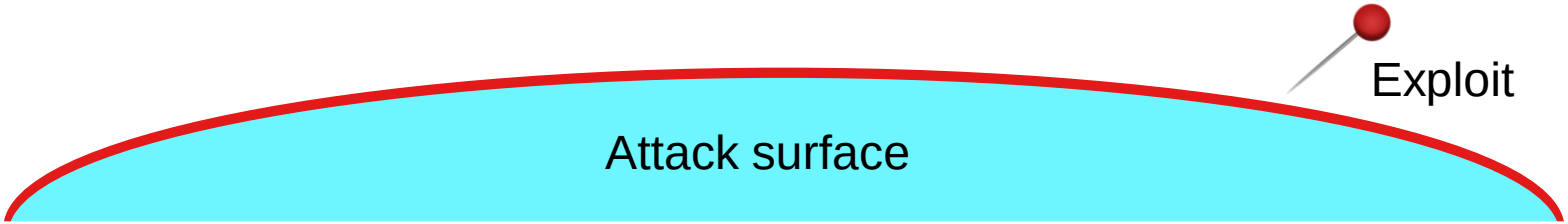
Half of the Java patches relate to Deserialization Flaws.

Customer Alert 20180418

Oracle Critical Patch Update April 2018 Released

https://www.waratek.com/alert-oracle-guidance-cpu-april-2018/

Why test for security?

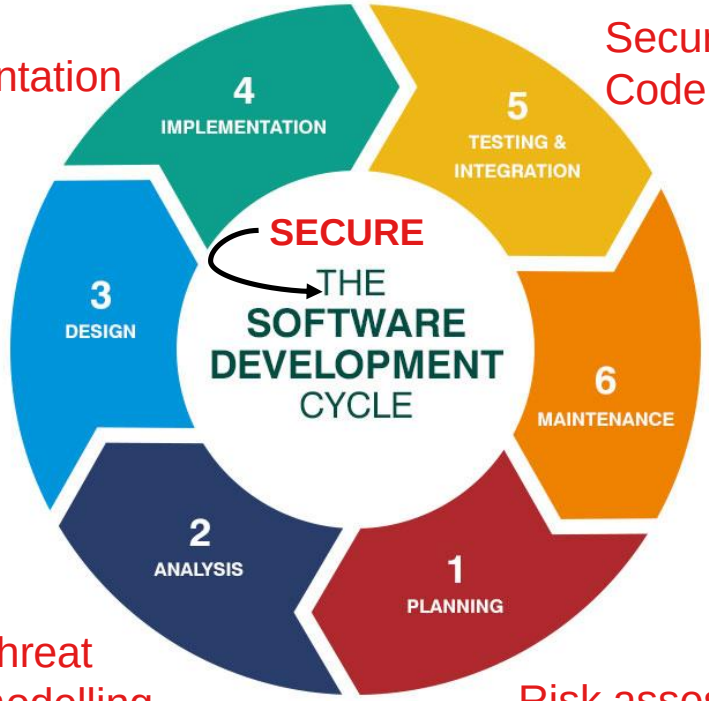


- Security testing → Non-functional testing
- *Who's job is to test for security?*

When to test for security?

Secure implementation

Security testing & Code reviews



Design for security

Patching & Updating

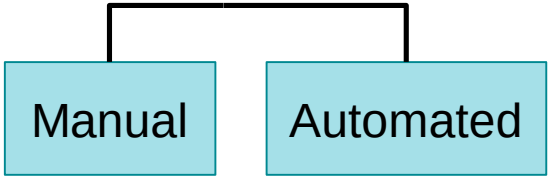
Threat modelling

Risk assessment & Abuse cases

https://www.dignitasdigital.com/blog/easy-way-to-understand-sdlc/

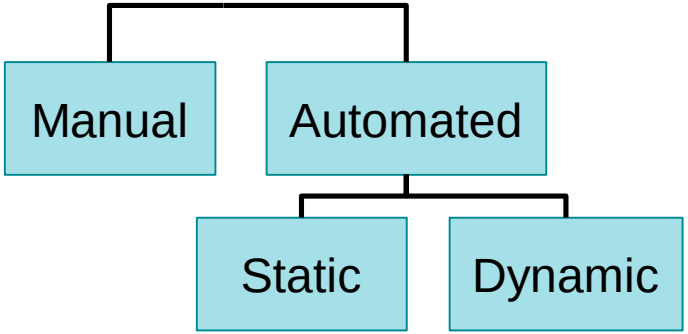
Classes of Security Testing

- Manual vs. Automated Testing



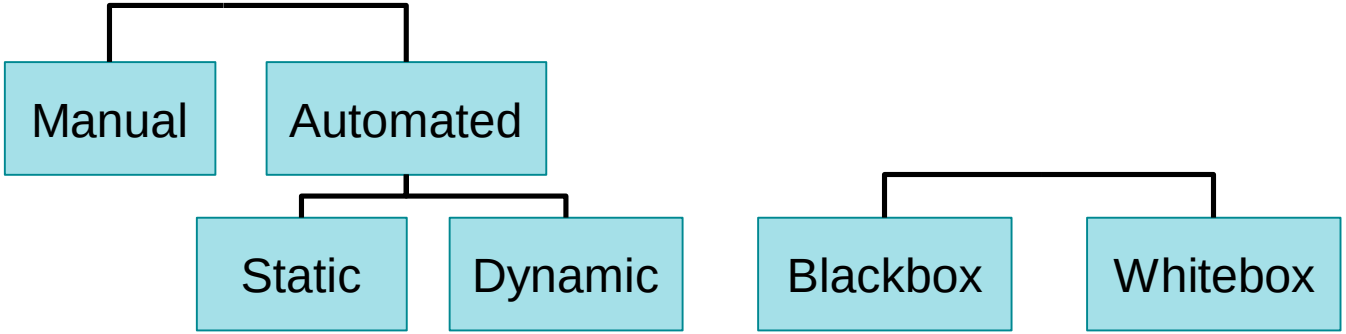
Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing



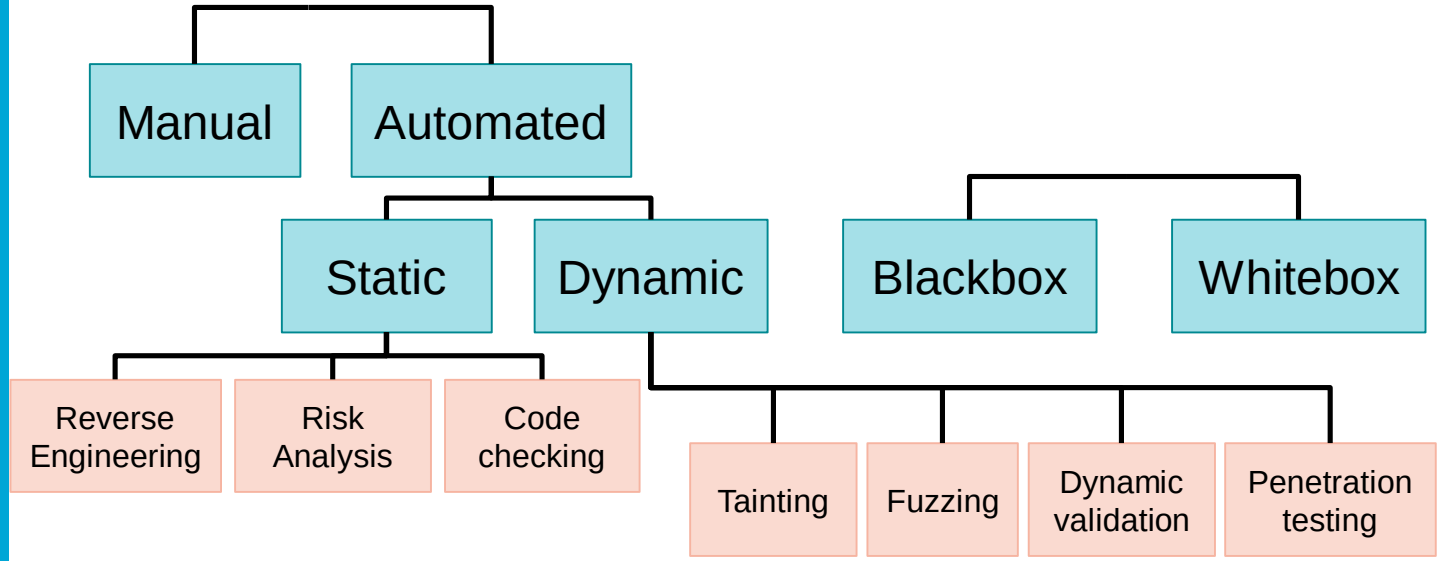
Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



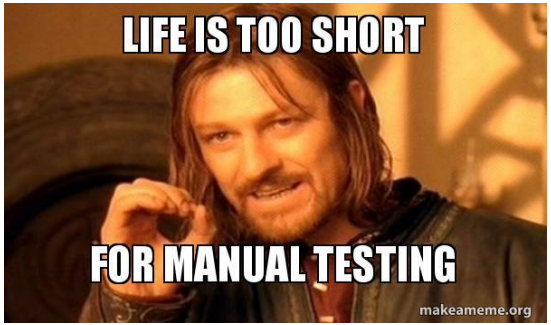
Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



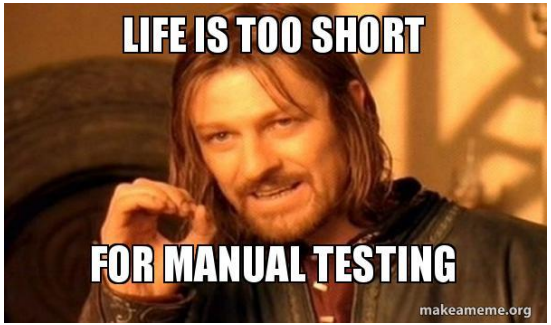
Manual vs. Automated Testing

- Manual
  - Code reviews
  - Efficient use of human expertise
  - Labour intensive



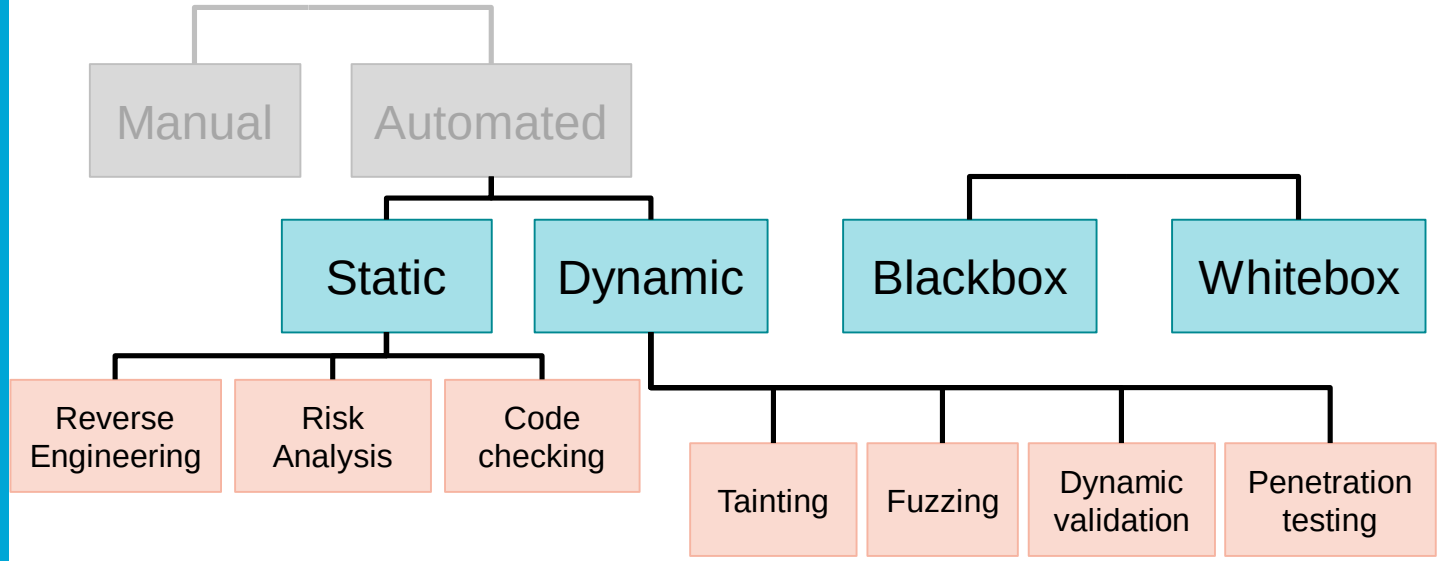
Manual vs. Automated Testing

- Manual
  - Code reviews
  - Efficient use of human expertise
  - Labour intensive
- Automated
  - Automated code checking
  - Can check MLOC in seconds
  - Incomparable to human expertise



Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



Static vs. Dynamic Testing

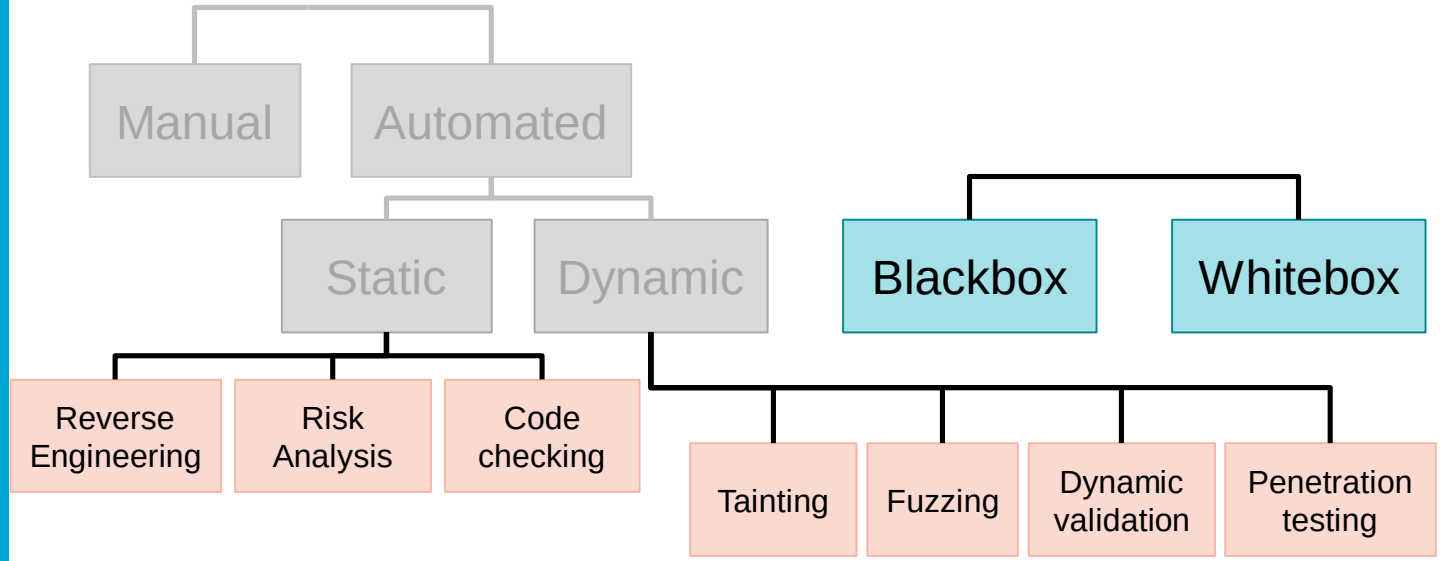
- (Automated) Static analysis
  - Code review by computers
  - Checks all possible code paths
  - Relatively easy to extract results
  - Limited capabilities

Static vs. Dynamic Testing

- (Automated) Static analysis
  - Code review by computers
  - Checks all possible code paths
  - Relatively easy to extract results
  - Limited capabilities
- Dynamic analysis
  - Execute code and observe behaviour
  - Checks functional code paths only
  - Much advanced analysis
  - Difficult to set up

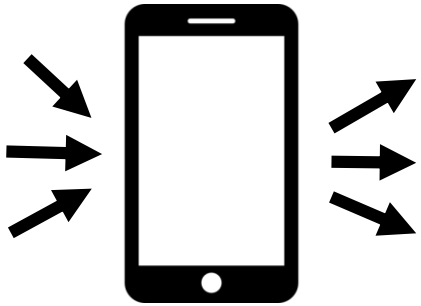
Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



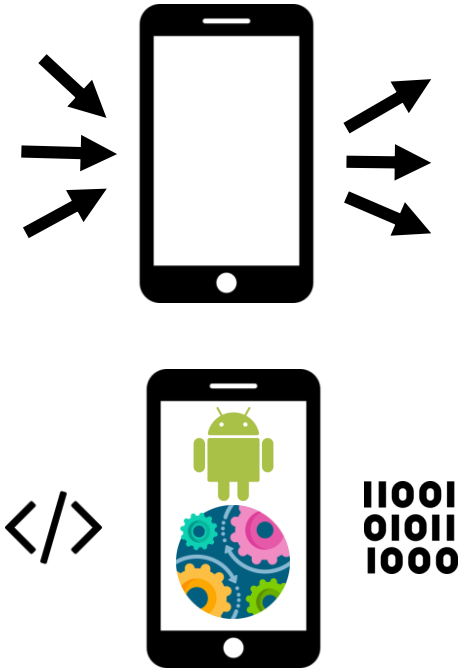
Black vs. White box Testing

- Black box
  - Unknown internal structure
  - Study Input → Output correlation
  - Generic technique
  - Requires end-to-end system
  - May miss components



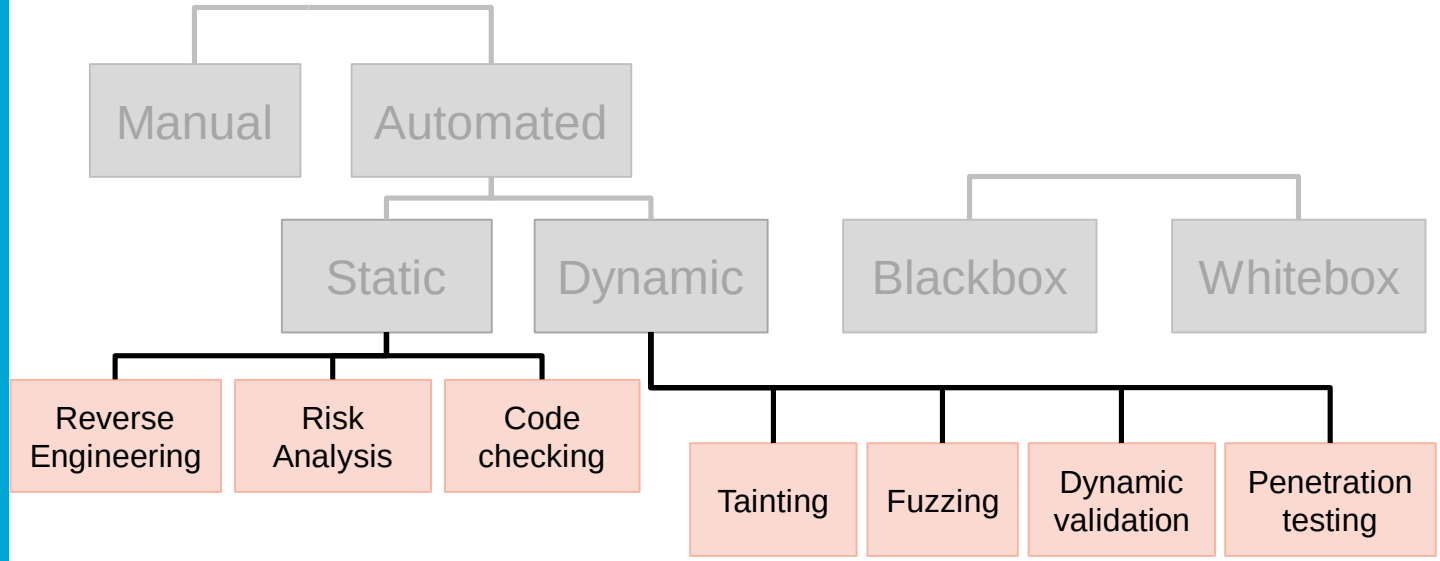
Black vs. White box Testing

- Black box
  - Unknown internal structure
  - Study Input → Output correlation
  - Generic technique
  - Requires end-to-end system
  - May miss components
- White box
  - Known internal structure
  - Analysis of internal structure
  - GUI not necessarily required
  - Thorough testing and debugging
  - Time consuming



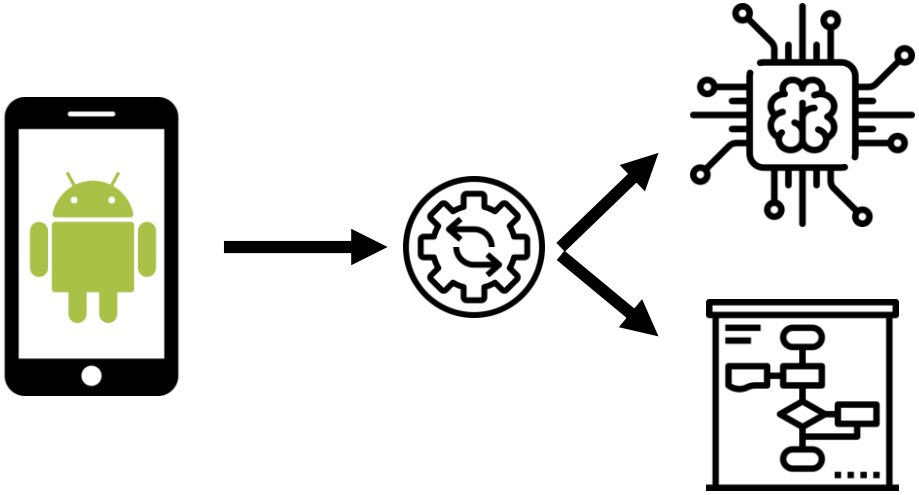
Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



Static Application Security Testing

- Reverse engineering (System level)
  - Disassemble application to extract internal structure
  - Black box to White box
  - Useful for gaining information



Static Application Security Testing

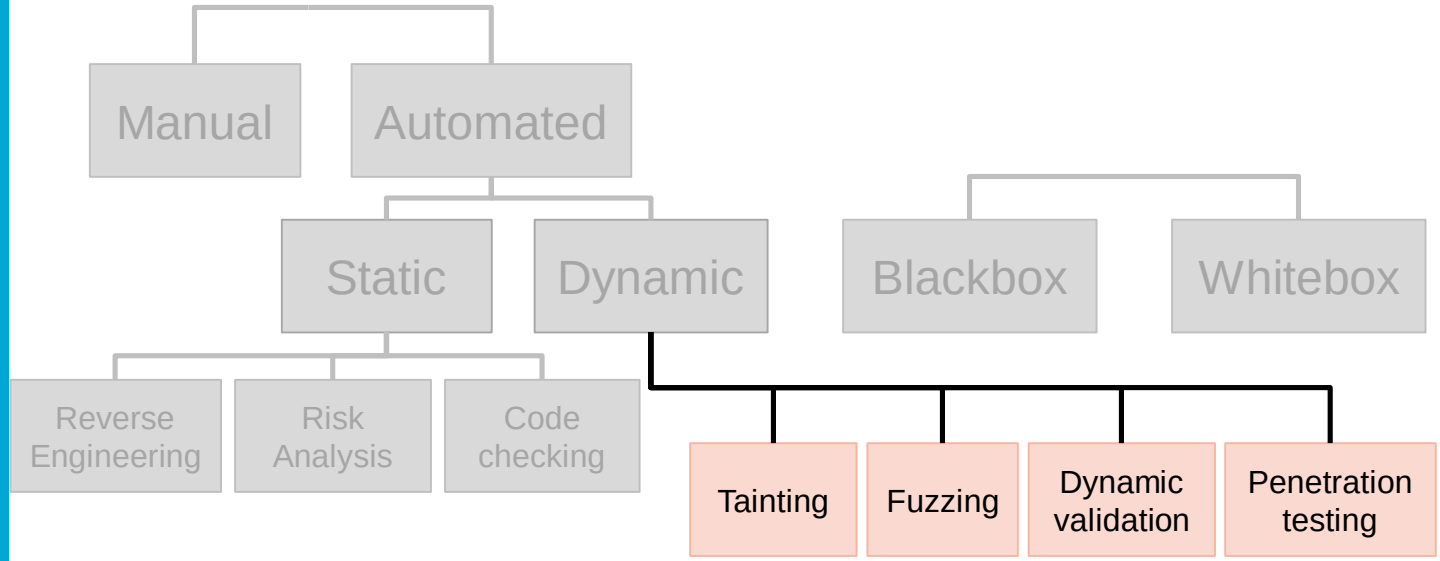
- Reverse engineering (System level)
- Risk-based testing (Business level)
  - Model worst case scenarios
  - Threat modelling for test case generation

Static Application Security Testing

- Reverse engineering (System level)
- Risk-based testing (Business level)
- Static code checker (Unit level)
  - Checks for rule violations via code structure
  - Parsers, Control Flow graphs, Data flow analysis
  - Identifies bad coding practices, potential security issues, etc.

Classes of Security Testing

- Manual vs. Automated Testing
- Static vs. Dynamic Testing
- Black vs. White box Testing



Dynamic Application Security Testing

- Taint analysis
  - Tracking variable values controlled by user
- Fuzzing
  - Bombard with garbage data to cause crashes
- Dynamic validation
  - Functional testing based on requirements
- Penetration testing
  - End-to-end black box testing

Topic for next lecture

Summary Part I

- Java vulnerabilities have large attack surfaces
- Crucial to adapt Secure SDLC
- Threat modelling can drive test case generation
- Static analysis checks code without executing it
- Dynamic analysis executes code and observes behavior

Quiz Time!

Which type of testing aims to convert a black box system to white box?

Reverse Engineering

Quiz Time!

Which vulnerability allows a remote attacker to change which instruction will be executed next?

Remote Code Execution

Quiz Time!

Why is Java safe from buffer overflows?

It’s not!

Agenda for today

- Part I
  - Latest security news
  - Security vulnerabilities in Java
  - Types of Security testing
    - SAST vs. DAST
- **Part II**
  - **SAST under the hood**
    - **Pattern Matching**
    - **Control Flow Analysis**
    - **Data Flow Analysis**
  - **SAST Tools performance**

TU Delft

Why doesn't the perfect static analysis tool exist?

TU Delft

Static Analysis

- Soundness
- Completeness

Static Analysis

- Soundness
  - No missed vulnerability (0 FNs)
  - No alarm → no vulnerability exists
- Completeness

Static Analysis

- Soundness
  - No missed vulnerability (0 FNs)
  - No alarm → no vulnerability exists
- Completeness
  - No false alarms (0 FPs)
  - Raises an alarm → vulnerability found

Static Analysis

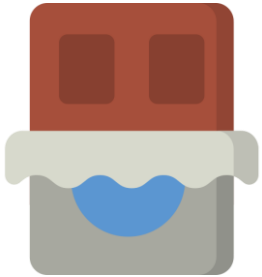
- Soundness
  - No missed vulnerability (0 FNs)
  - No alarm → no vulnerability exists
- Completeness
  - No false alarms (0 FPs)
  - Raises an alarm → vulnerability found

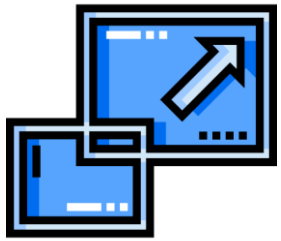
- Ideally: ↑Soundness + ↑Completeness
- Reality: Compromise on FPs or FNs

TU Delft

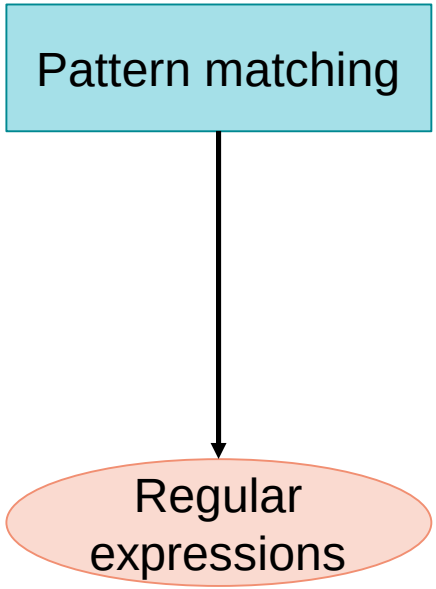
Usable SAST Tools

- ↓ FPs vs. ↓ FNs
- ↑ Interpretability
- ↑ Scalability

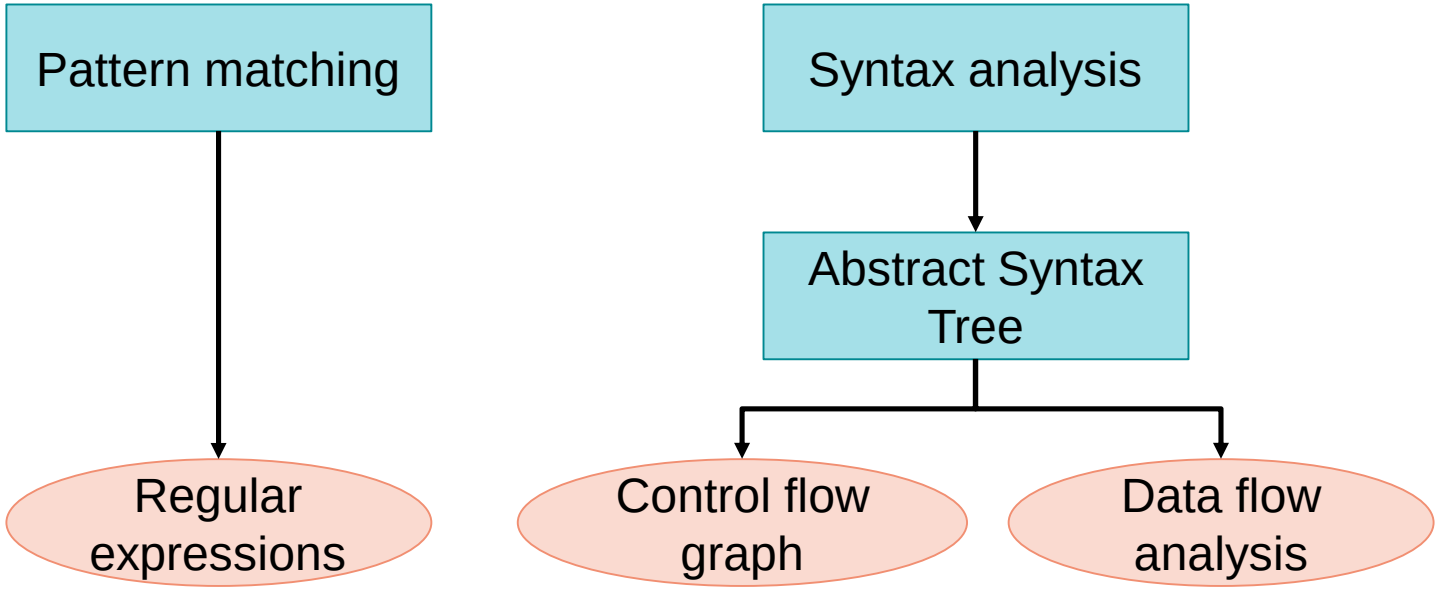




SAST under the hood



SAST under the hood



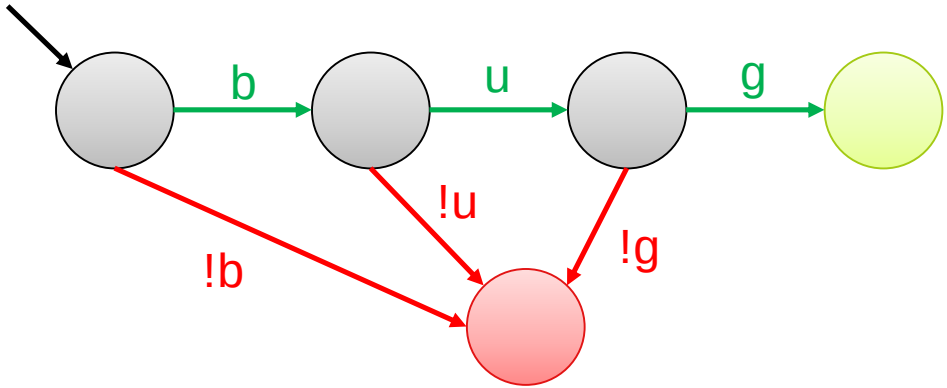
Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

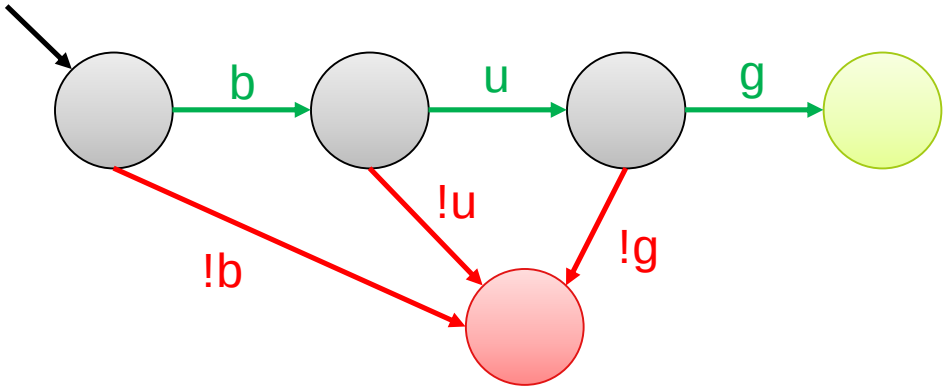
- *Find all instances of “bug”*



Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

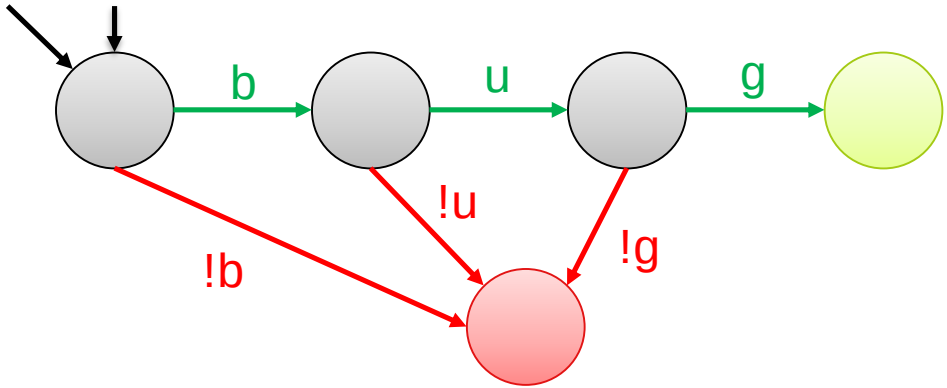


bug

Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

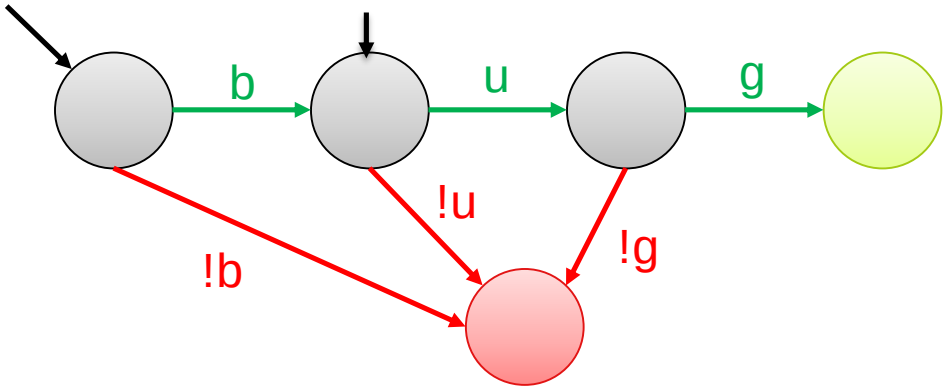


bug

Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

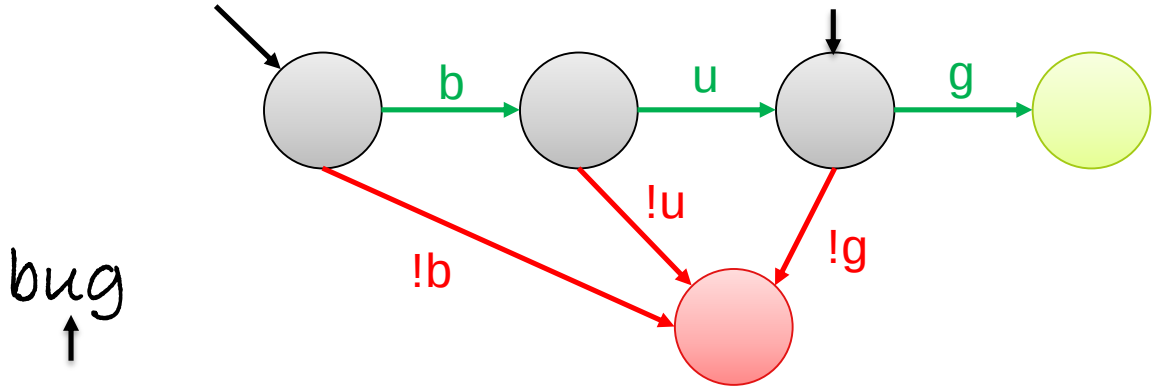


bug

Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

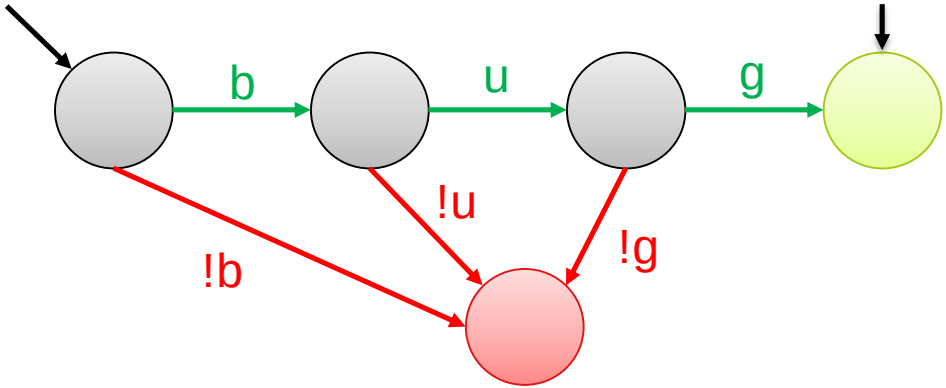
- *Find all instances of “bug”*



Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

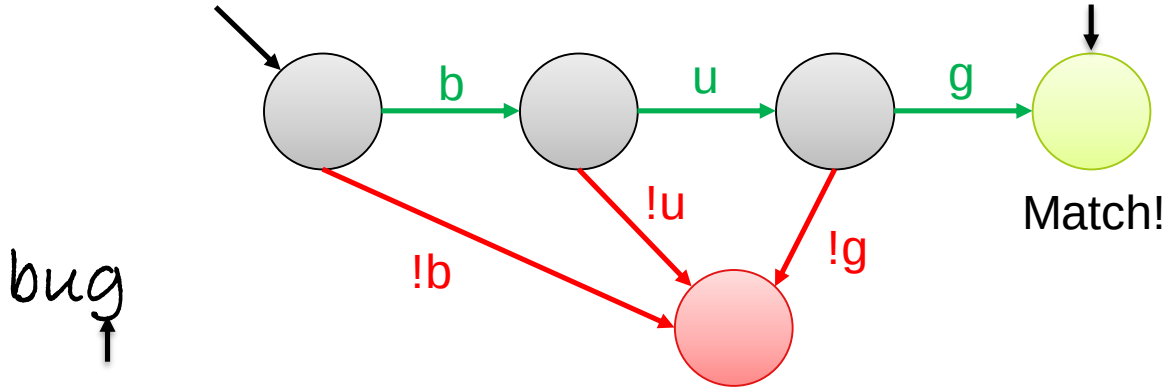


bug

Pattern Matching

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

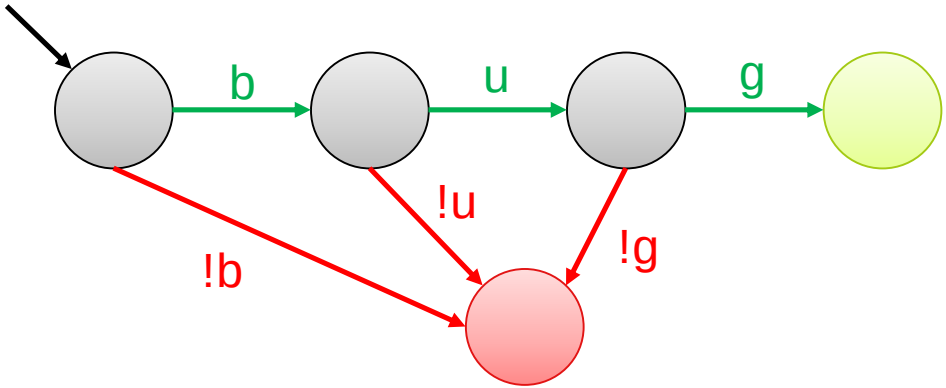
- *Find all instances of “bug”*



Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

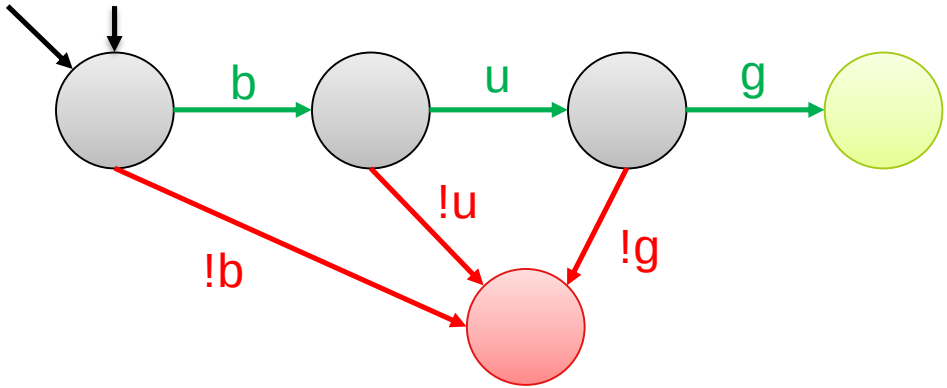


bag

Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

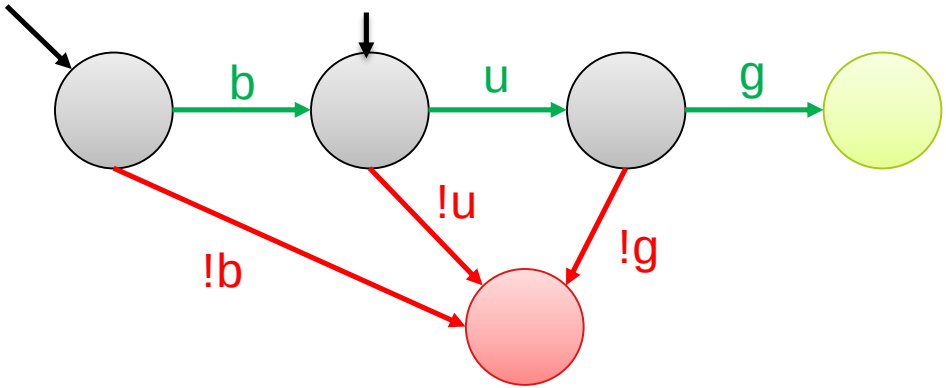


bag

Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

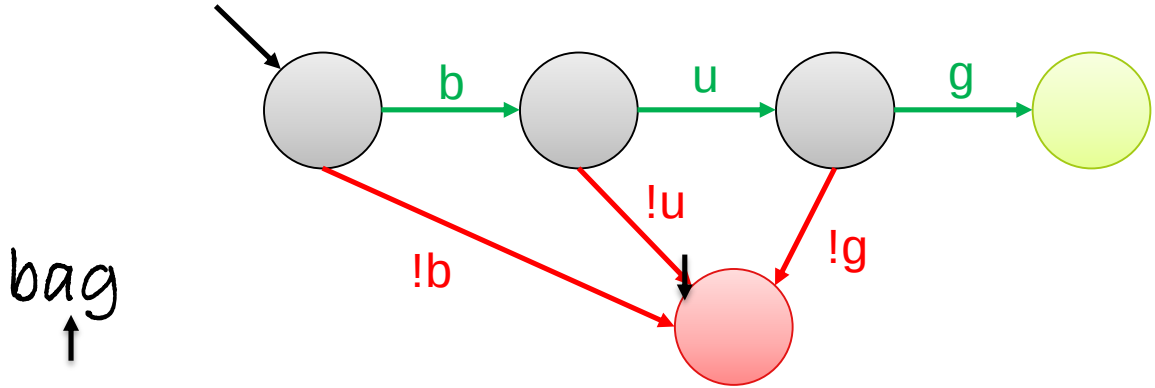


bag

Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

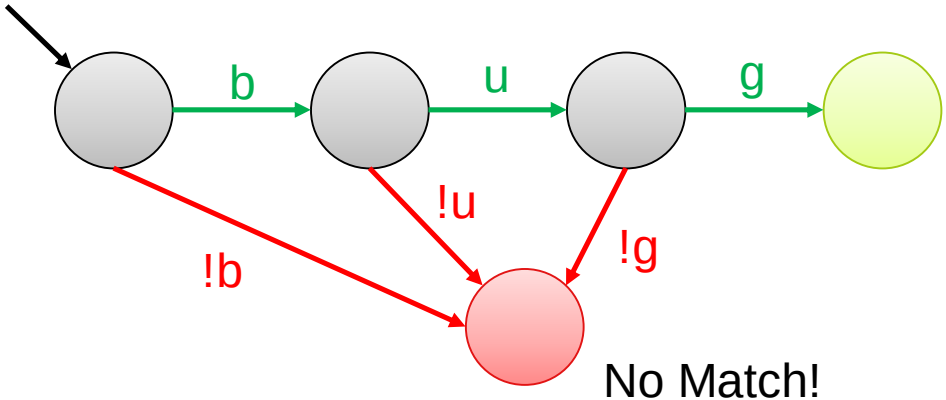
- *Find all instances of “bug”*



Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of “bug”*

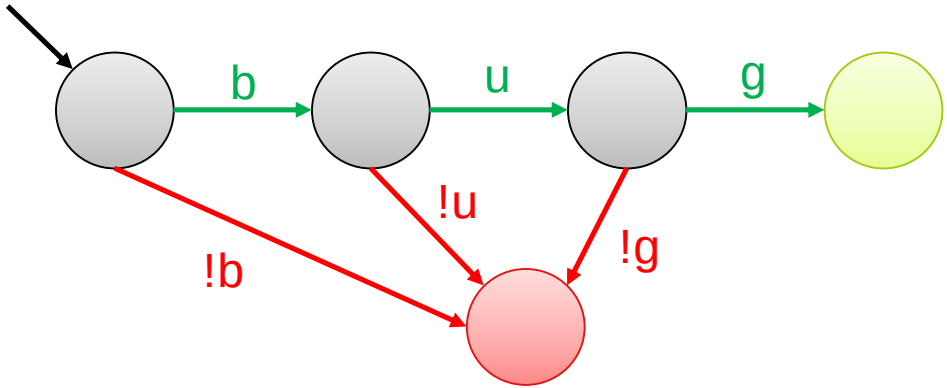


bag

Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

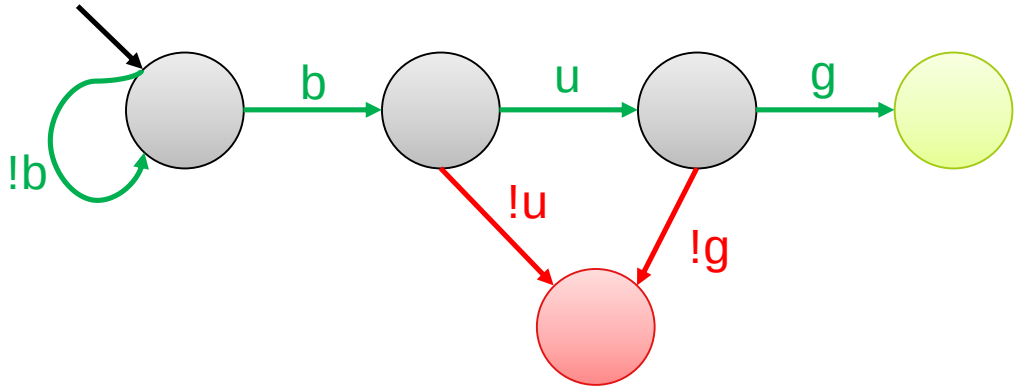
- *Find all instances of ".*bug"*



Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

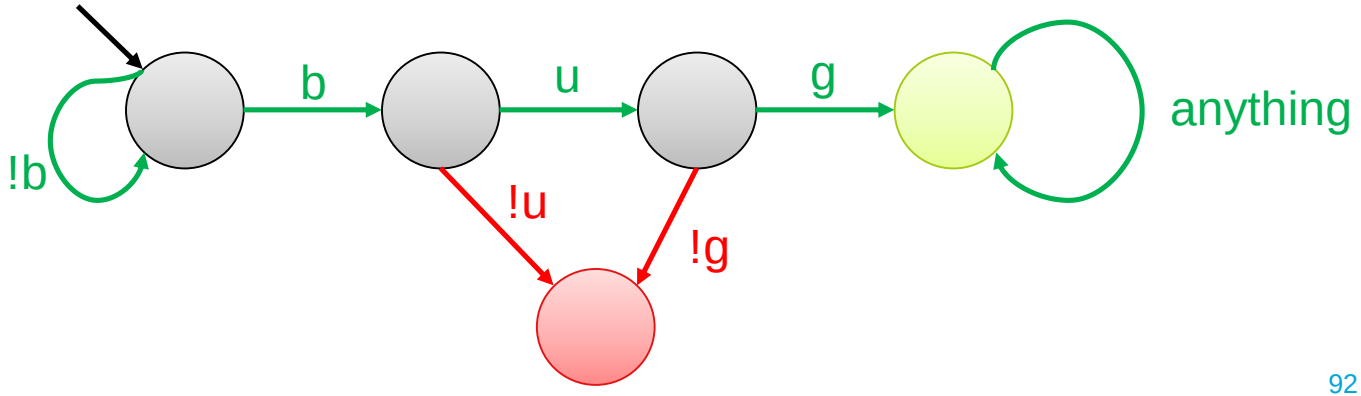
- *Find all instances of ".*bug"*



Pattern Matching via Regex

- Look for predefined patterns in code
  - Regular Expressions
  - Finite State Automata

- *Find all instances of ".*bug.*"*



Pattern Matching via Regex

- Finds low hanging fruit
  - Misconfigurations (port 22 open for everyone)
  - Bad imports (System.io.*)
  - Call to dangerous functions (strcpy, memcpy)

Pattern Matching via Regex

- Finds low hanging fruit
  - Misconfigurations (port 22 open for everyone)
  - Bad imports (System.io.*)
  - Call to dangerous functions (strcpy, memcpy)
- Shortcomings
  - Lots of FPs
  - Limited support

Pattern Matching via Regex

- Finds low hanging fruit
  - Misconfigurations (port 22 open for everyone)
  - Bad imports (System.io.*)
  - Call to dangerous functions (strcpy, memcpy)
- Shortcomings
  - Lots of FPs
  - Limited support

```
boolean DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```

Pattern Matching via Regex

- Finds low hanging fruit
  - Misconfigurations (port 22 open for everyone)
  - Bad imports (System.io.*)
  - Call to dangerous functions (strcpy, memcpy)
- Shortcomings
  - Lots of FPs
  - Limited support

```
boolean DEBUG = false;

if (DEBUG) {
  → System.out.println("Debug line 1");
  → System.out.println("Debug line 2");
  → System.out.println("Debug line 3");
    }
```

Syntactic Analysis

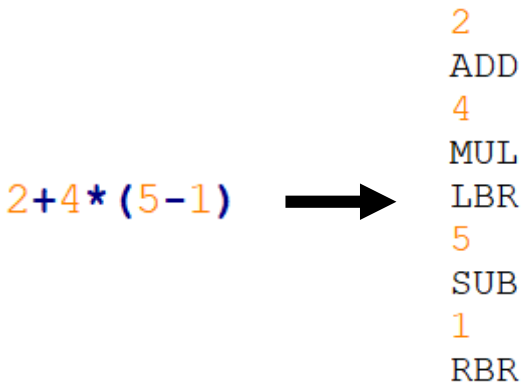
- Performed via Parsers

- Tokens → Hierarchal data structures
  - Parse Tree – Concrete representation
  - Abstract Syntax Tree – Abstract representation

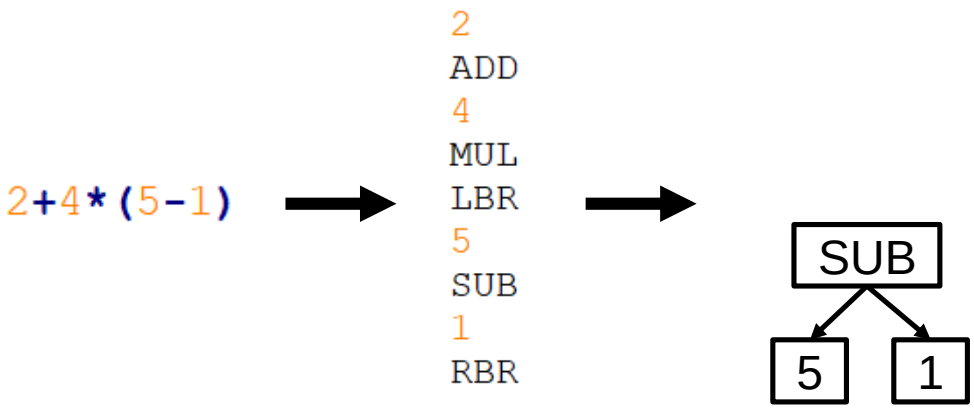
Abstract Syntax Tree (AST)

```
2+4*(5-1)
```

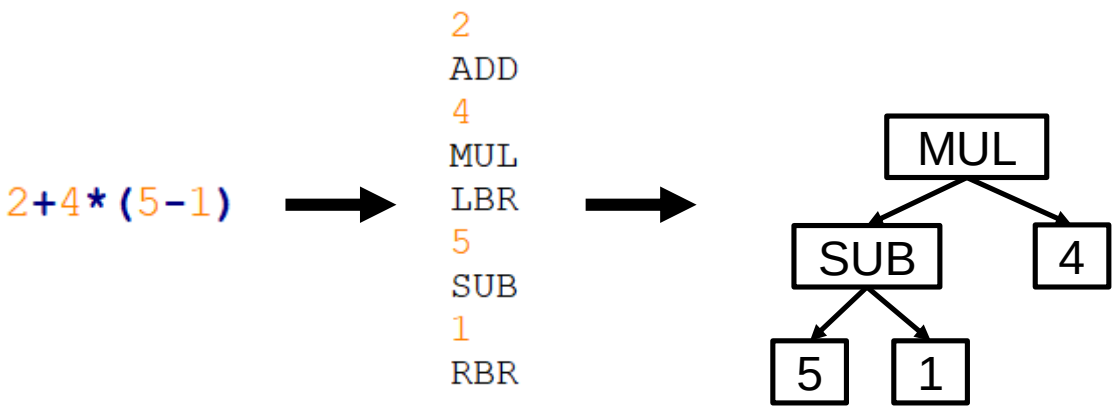
Abstract Syntax Tree (AST)



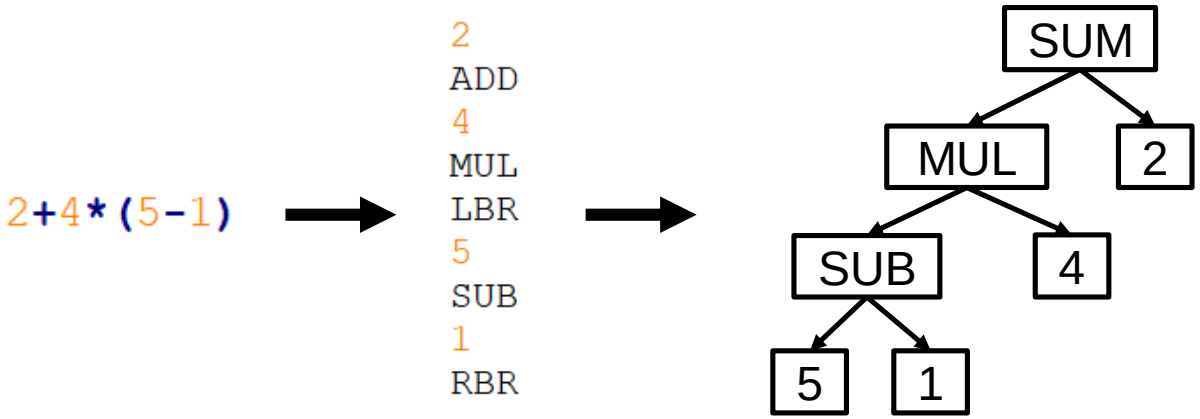
Abstract Syntax Tree (AST)



Abstract Syntax Tree (AST)



Abstract Syntax Tree (AST)



Abstract Syntax Tree (AST)

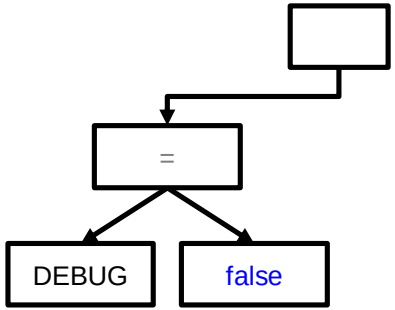
```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```

Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```

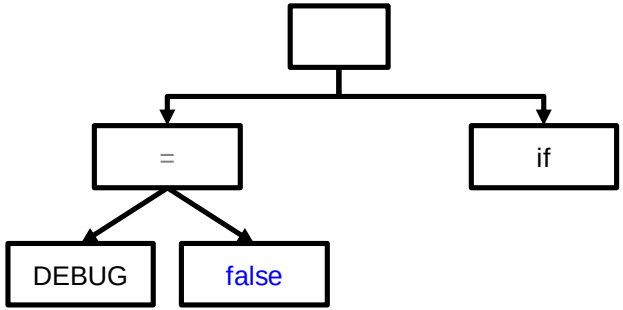
Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```



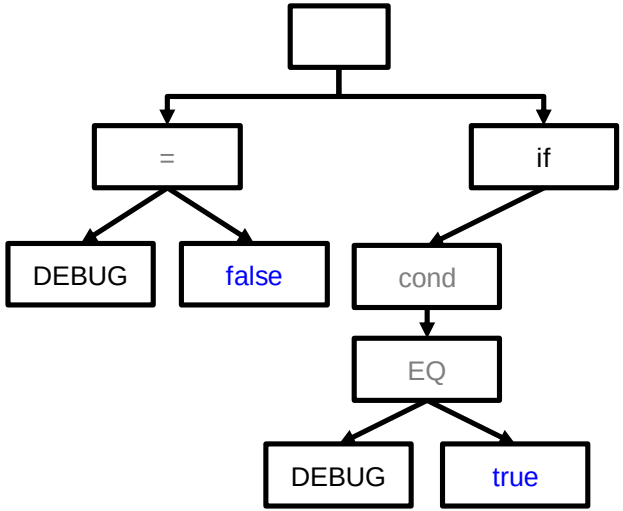
Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```



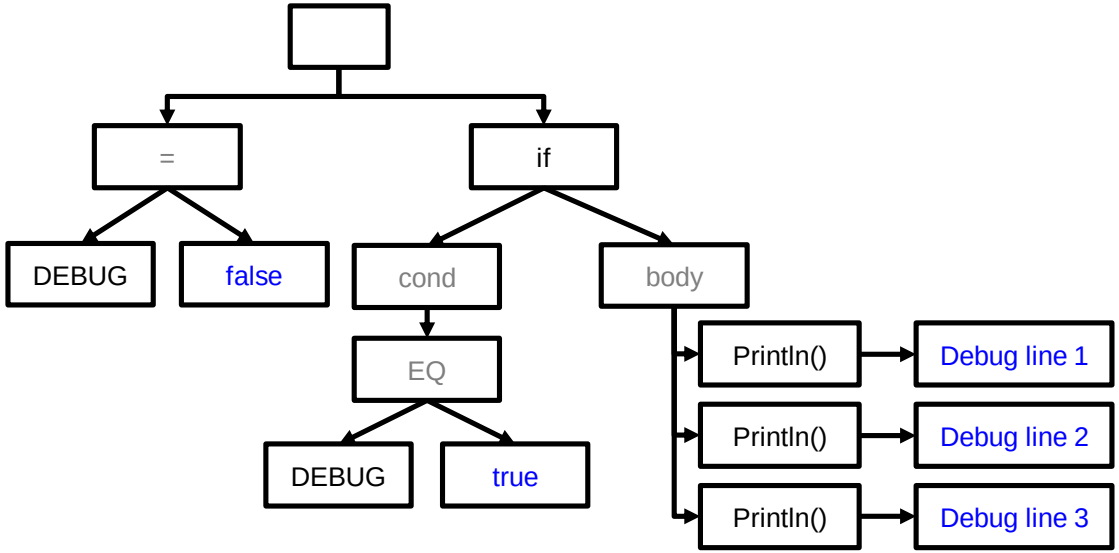
Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```



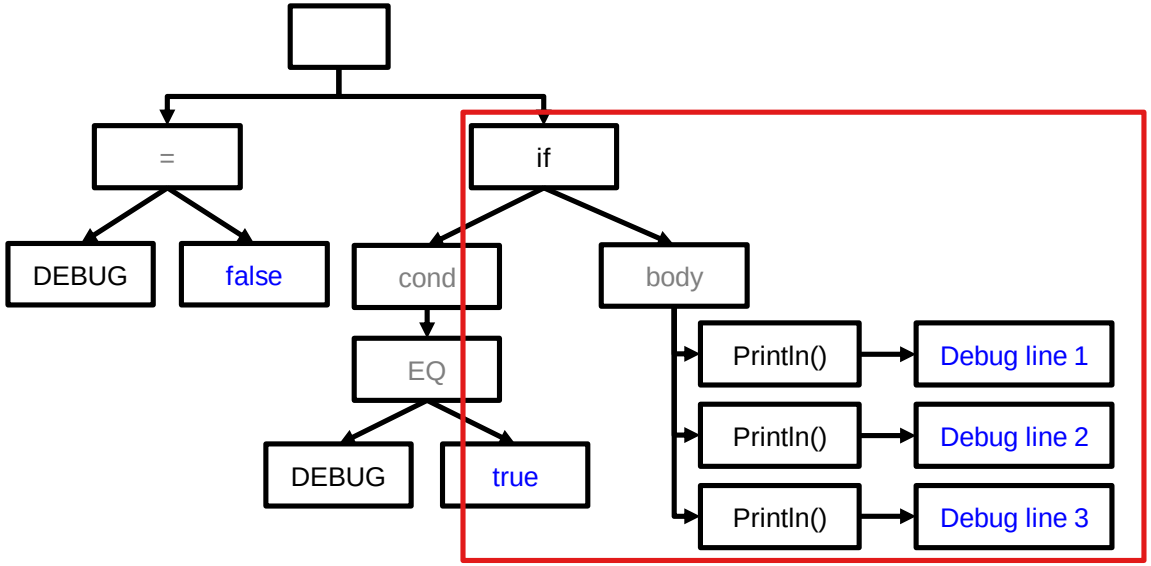
Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```



Abstract Syntax Tree (AST)

```
DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
    }
```



Syntactic Analysis via AST

Syntactic Analysis via AST

Rule # 1: Allow 3 methods

Syntactic Analysis via AST

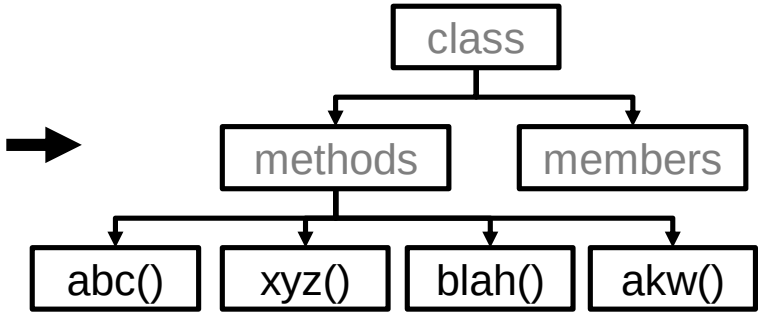
Rule # 1: Allow 3 methods

```
public class test {
    public void abc() {...}
    public void xyz() {...}
    public void blah() {...}
    public int akw() {...}
}
```

Syntactic Analysis via AST

Rule # 1: Allow 3 methods

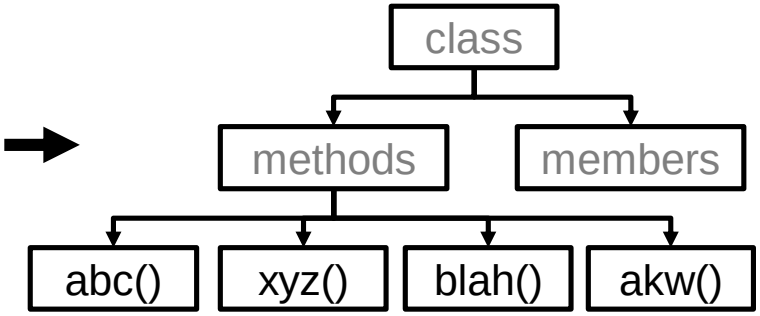
```
public class test {
    public void abc() {...}
    public void xyz() {...}
    public void blah() {...}
    public int akw() {...}
}
```



Syntactic Analysis via AST

Rule # 1: Allow 3 methods

```
public class test {
    public void abc() {...}
    public void xyz() {...}
    public void blah() {...}
    public int akw() {...}
}
```



Error: Too many methods!

Syntactic Analysis via AST

Rule # 2: printf(format_string, args_to_print)

Syntactic Analysis via AST

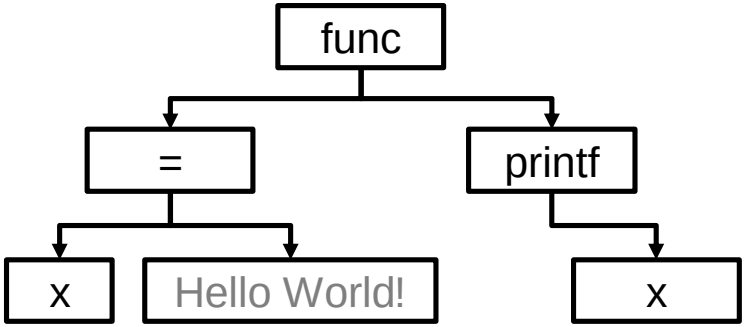
Rule # 2: printf(format_string, args_to_print)

```
x = "Hello World!";
printf(x);
```

Syntactic Analysis via AST

Rule # 2: printf(format_string, args_to_print)

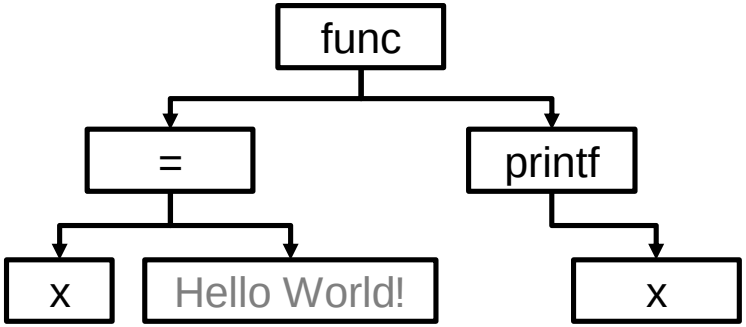
```
x = "Hello World!";
printf(x);
```



Syntactic Analysis via AST

Rule # 2: printf(format_string, args_to_print)

```
x = "Hello World!";
printf(x);
```



Error: Missing param!

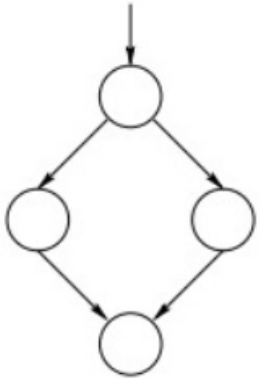
Control Flow Graphs

- Shows all execution paths a program *might* take
- Trace execution without executing program
- Nodes → Basic blocks
- Transitions → Control transfers

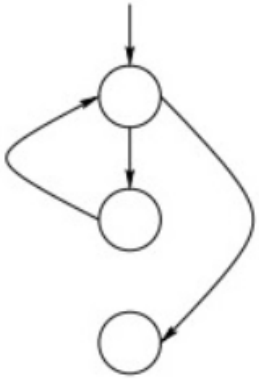
https://dzone.com/articles/how-draw-control-flow-graph

Control Flow Graphs

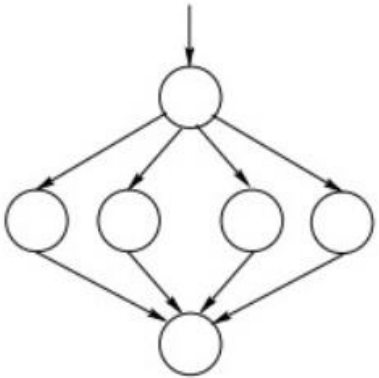
- Shows all execution paths a program *might* take
- Trace execution without executing program
- Nodes → Basic blocks
- Transitions → Control transfers



If-then-else



while



case

https://dzone.com/articles/how-draw-control-flow-graph

Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

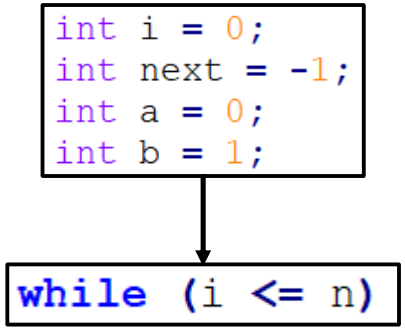
Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

```
int i = 0;
int next = -1;
int a = 0;
int b = 1;
```

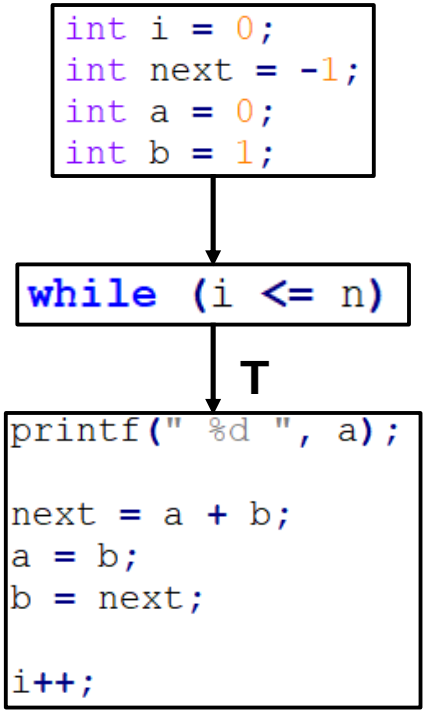
Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```



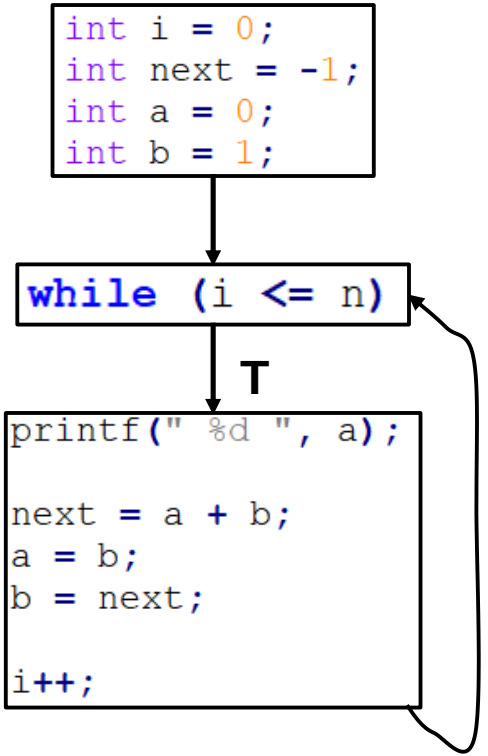
Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```



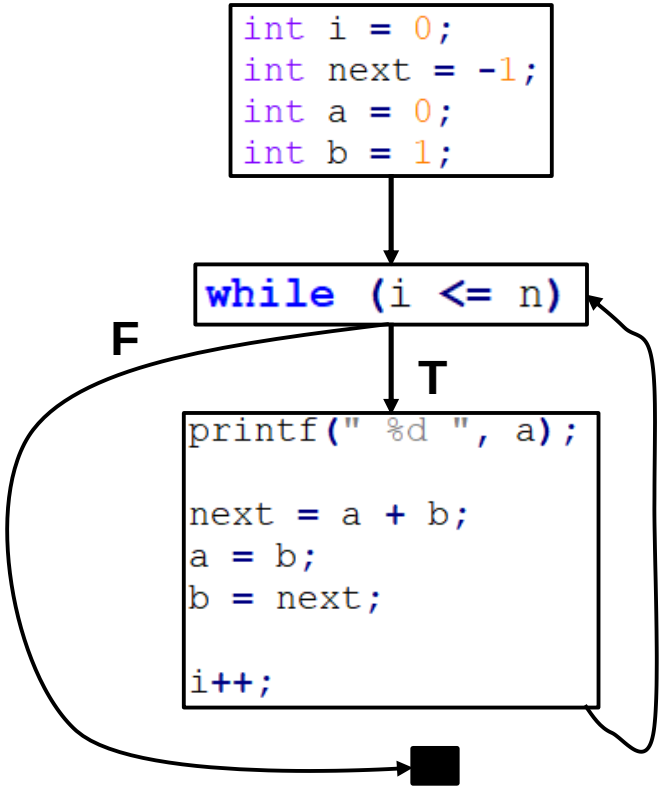
Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```



Control Flow Graphs

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

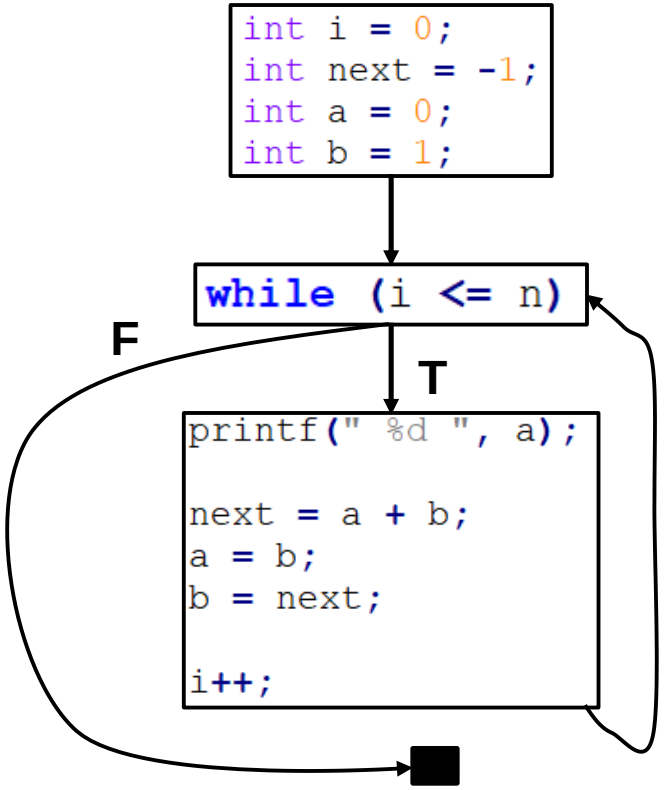


Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

Only traces control

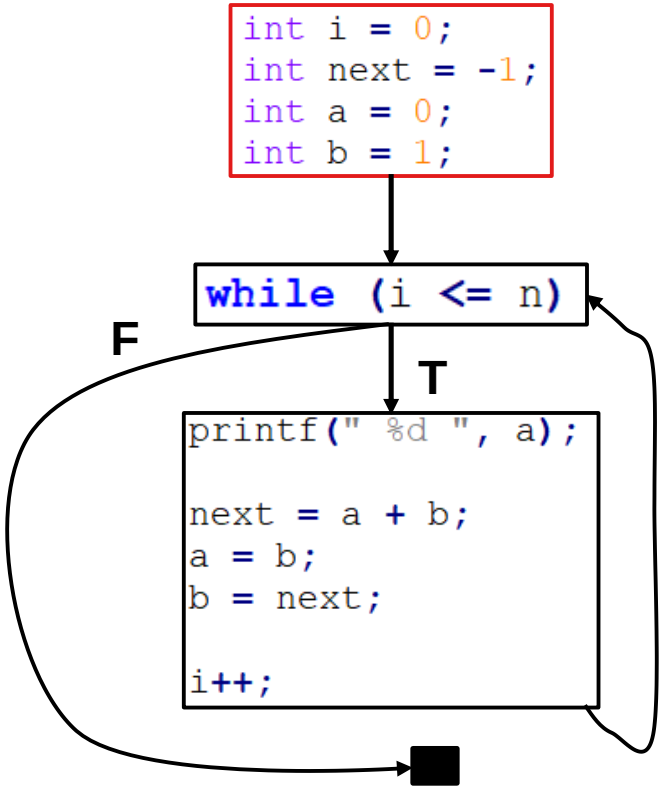


Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

Only traces control

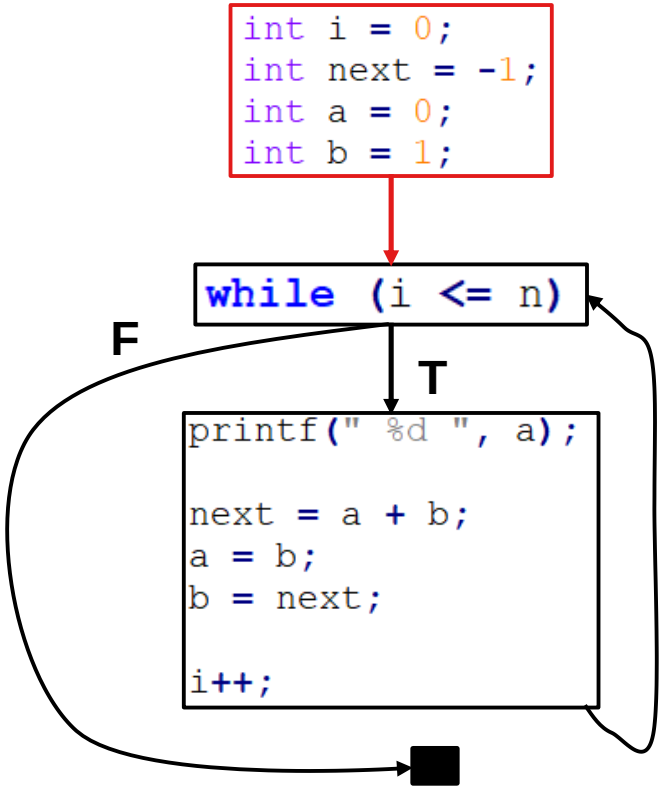


Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

Only traces control

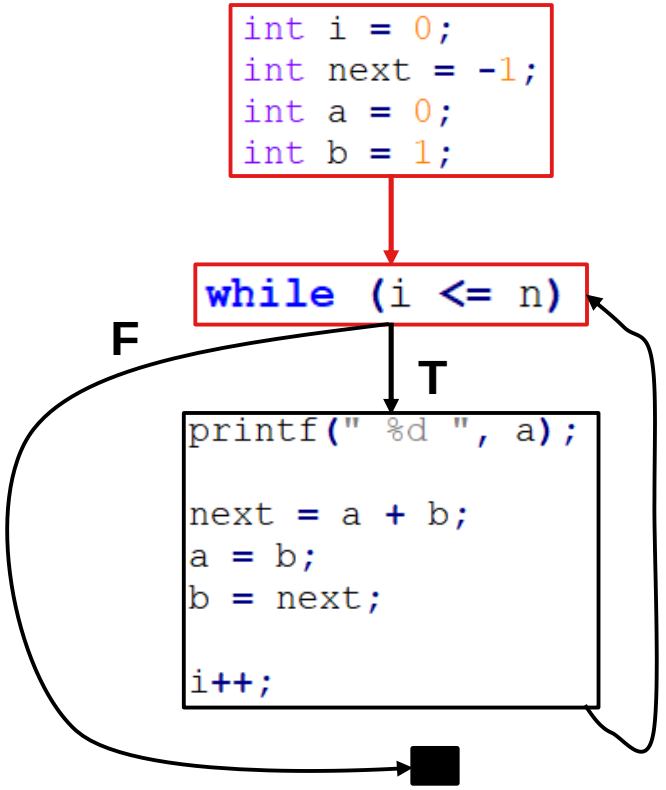


Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

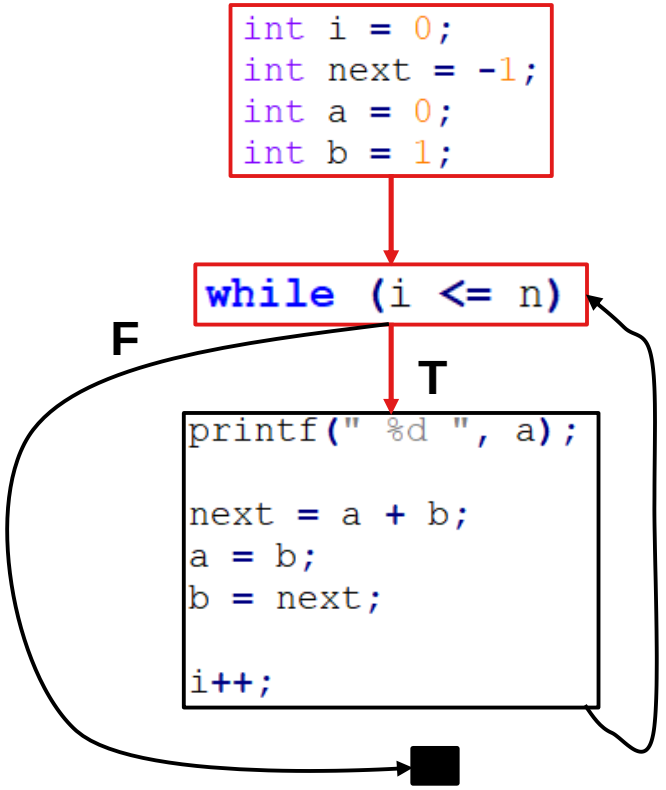
Only traces control



Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

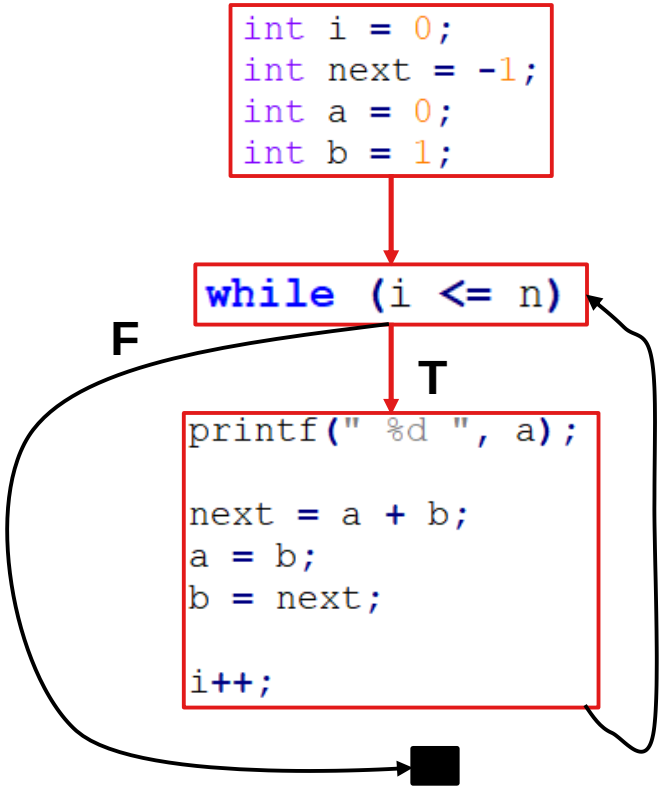


Only traces control

Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```



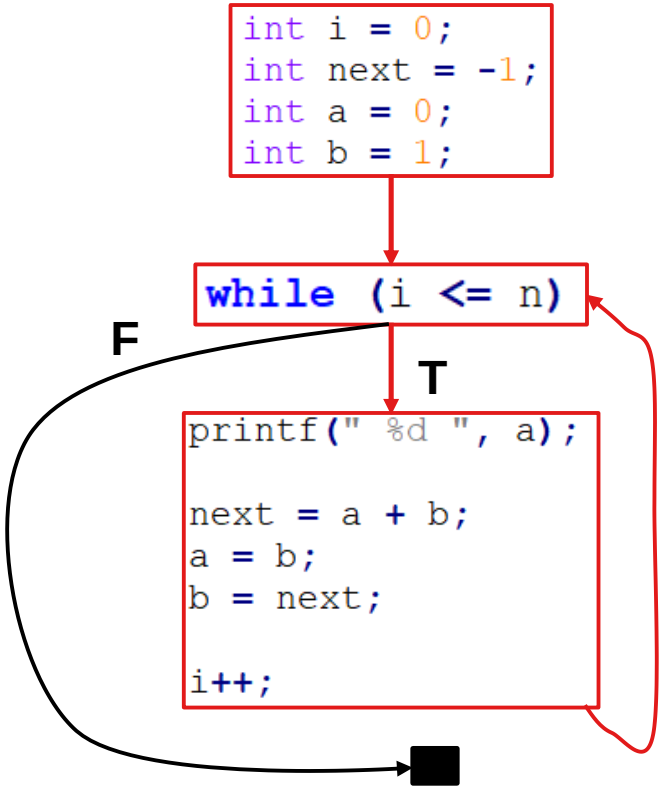
Only traces control

Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```

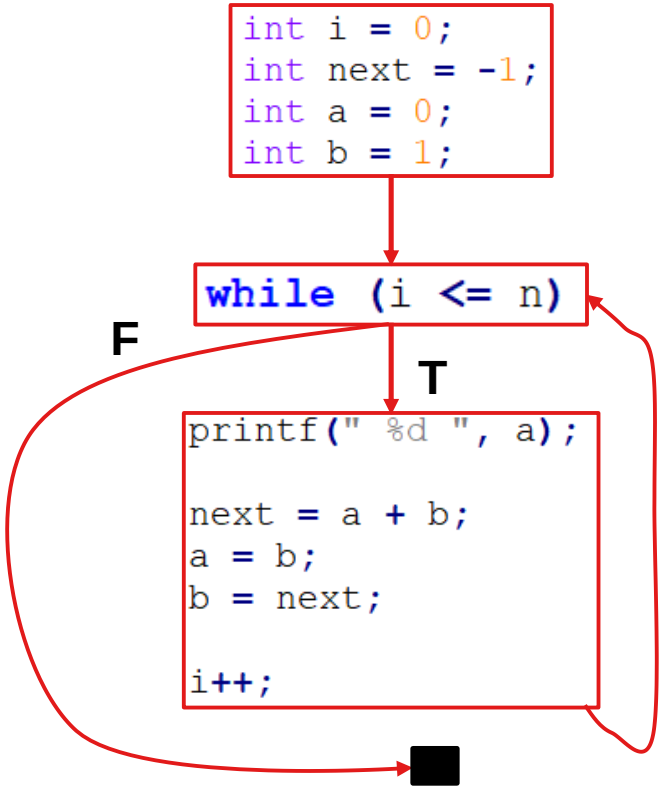
Only traces control



Control Flow Graphs

n=?

```
public void fibb(int n) {
    int i = 0;
    int next = -1;
    int a = 0;
    int b = 1;

    while (i <= n) {
        printf(" %d ", a);

        next = a + b;
        a = b;
        b = next;

        i++;
    }
}
```



Only traces control

Data Flow Analysis

- Tracks data values throughout program
- Shows all values variables *might* have

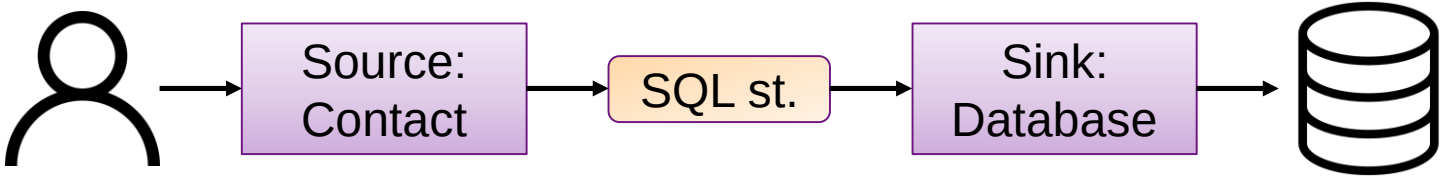
- User controlled variable (Source) → Tainted
- Rest (Sink) → Untainted

Data Flow Analysis

- Prove that
  - No untainted data is expected
  - No tainted data is used

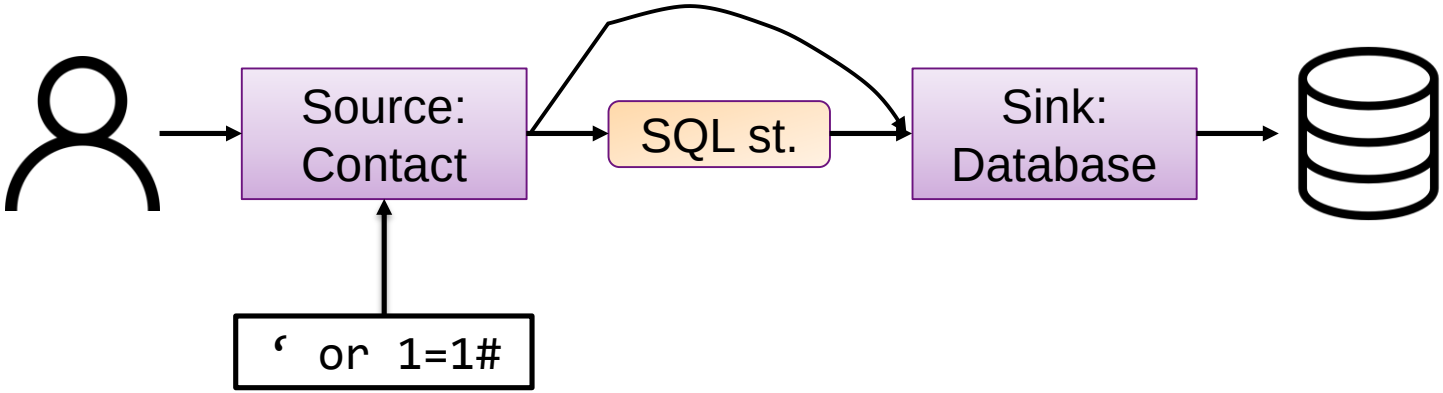
Data Flow Analysis

- Prove that
  - No untainted data is expected
  - No tainted data is used



Data Flow Analysis

- Prove that
  - No untainted data is expected
  - No tainted data is used



Source/Sink Clash

```
/* uses badsource and badsink */
public void bad(HttpServletRequest request, HttpServletResponse response)
        throws Throwable
{
    String data;

    /* POTENTIAL FLAW: Read data from a
     * querystring using getParameter
     */
    data = request.getParameter("name");

    if (data != null)
    {
        /* POTENTIAL FLAW: Display of data in web page
         * after using replaceAll() to remove script
         * tags, which will still allow XSS with strings
         * like <scr<script>ipt> (CWE 182: Collapse
         * of Data into Unsafe Value)
         */
        response.getWriter().println("<br>bad(): data = "
         + data.replaceAll("(<script>)", ""));
    }


}
```

`data` is tainted

`println()` expects untainted

Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```

Data Flow Analysis

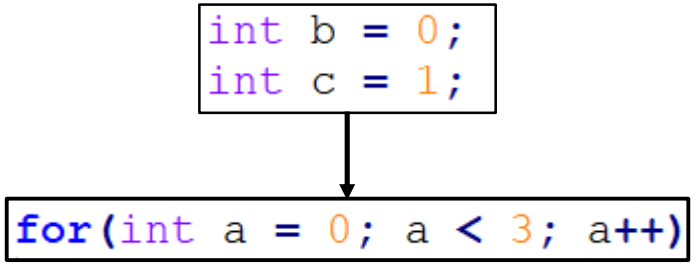
- Reaching definitions
  - Top-down approach
  - Possible values of a variable

```
int b = 0;
int c = 1;
```

```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```

Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

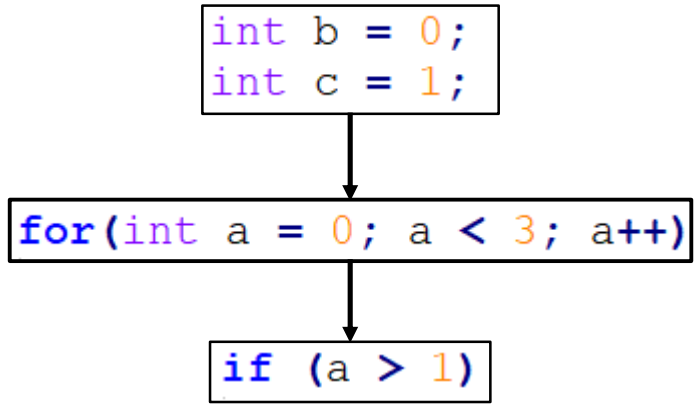



```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```

Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

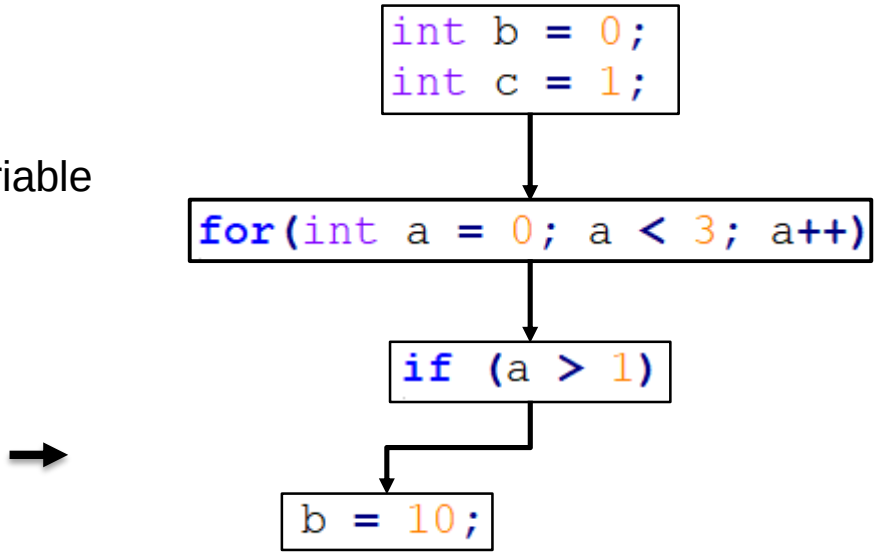
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```



Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

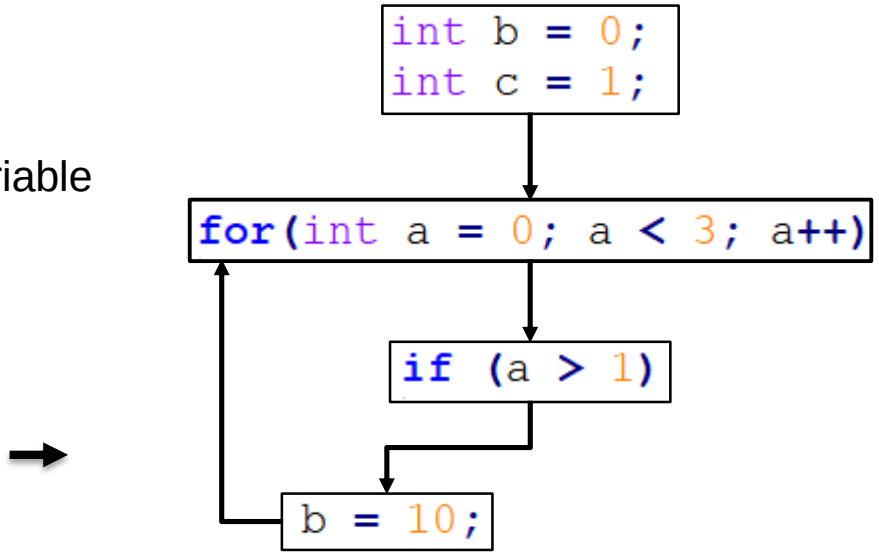
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```



Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

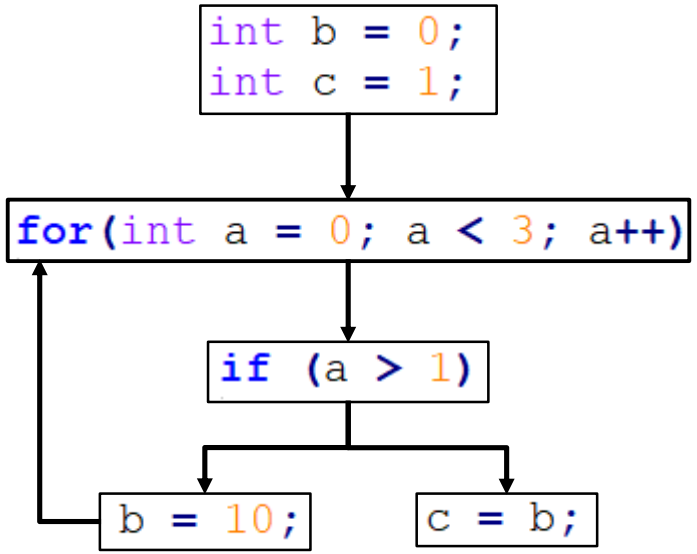
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```



Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

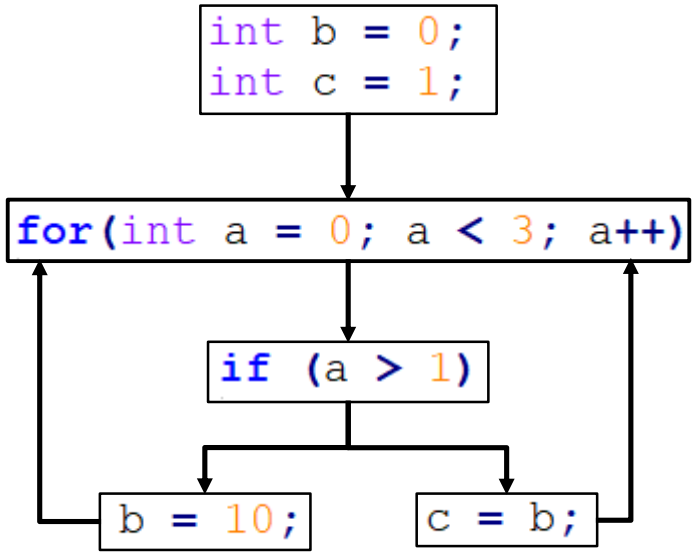
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```



Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

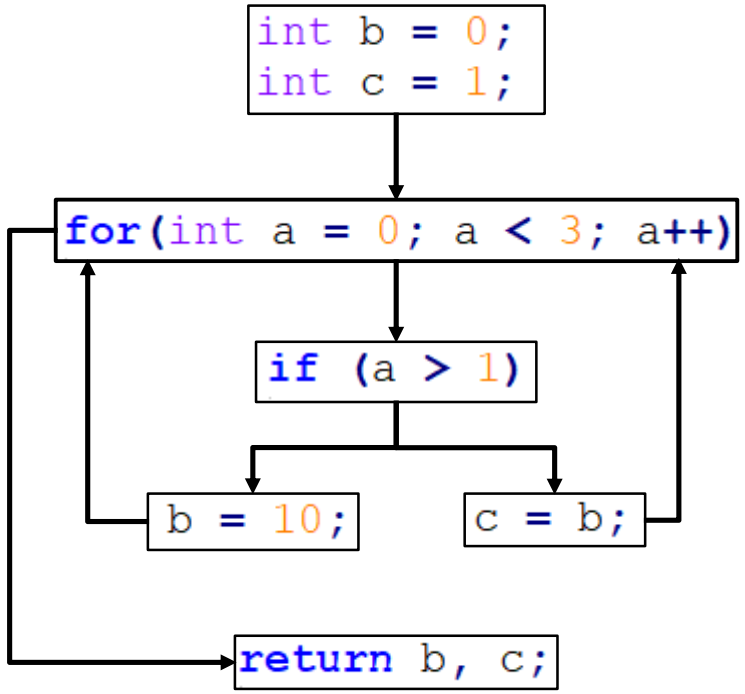
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```



Data Flow Analysis

- Reaching definitions
  - Top-down approach
  - Possible values of a variable

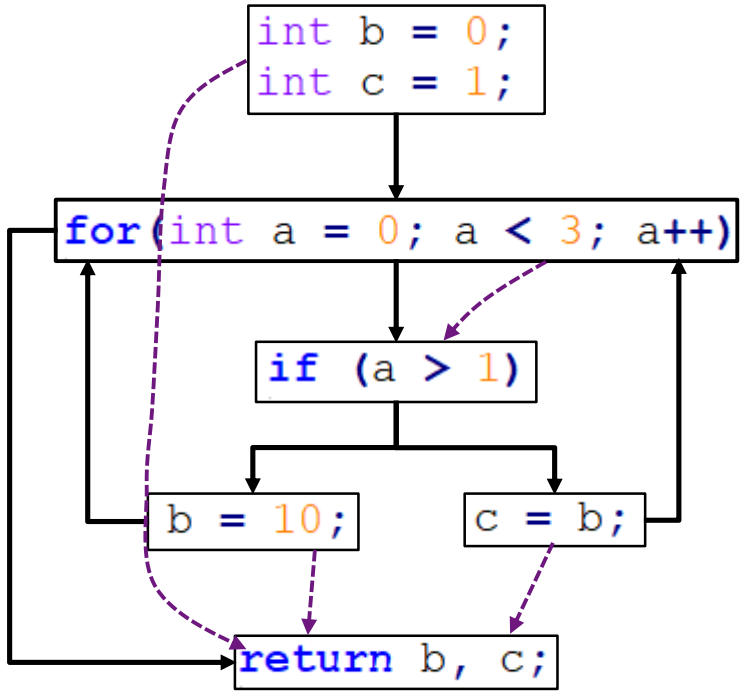
```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```

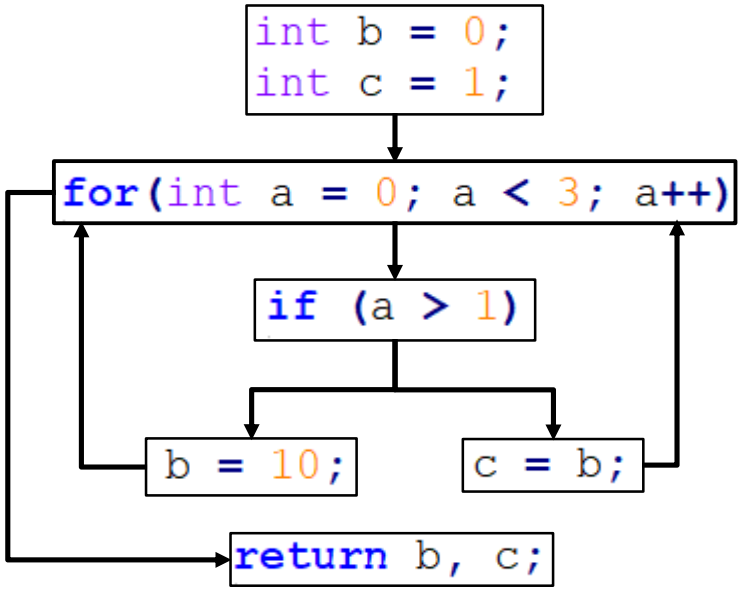


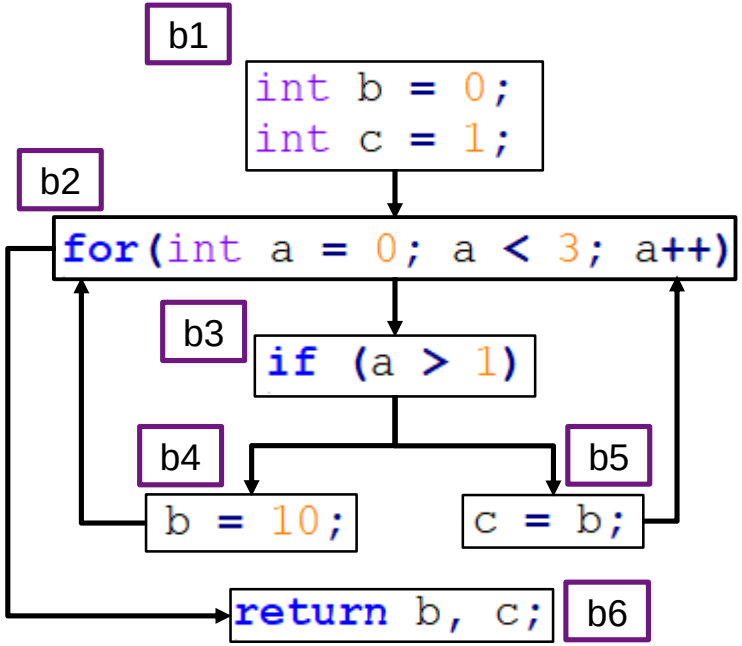
Data Flow Analysis

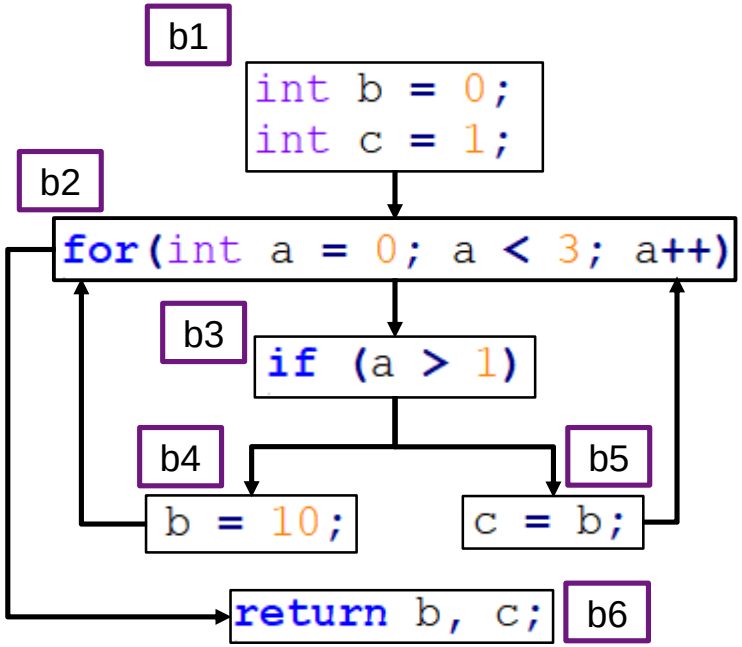
- Reaching definitions
  - Top-down approach
  - Possible values of a variable

```
int b = 0;
int c = 1;

for(int a = 0; a < 3; a++) {
    if (a > 1)
        b = 10;
    else
        c = b;
}
return b, c;
```

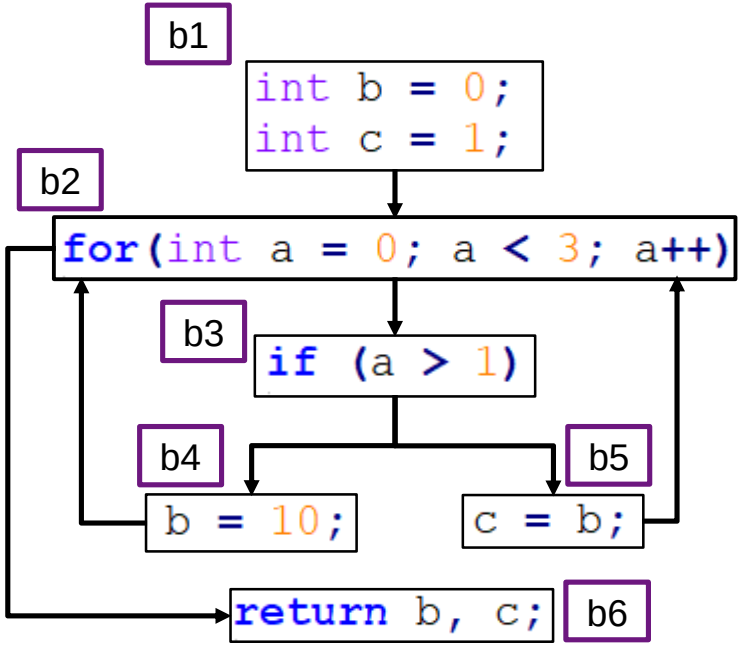



int b = 0;
int c = 1;
for(int a = 0; a < 3; a++)
if (a > 1)
b = 10;
c = b;
return b, c;

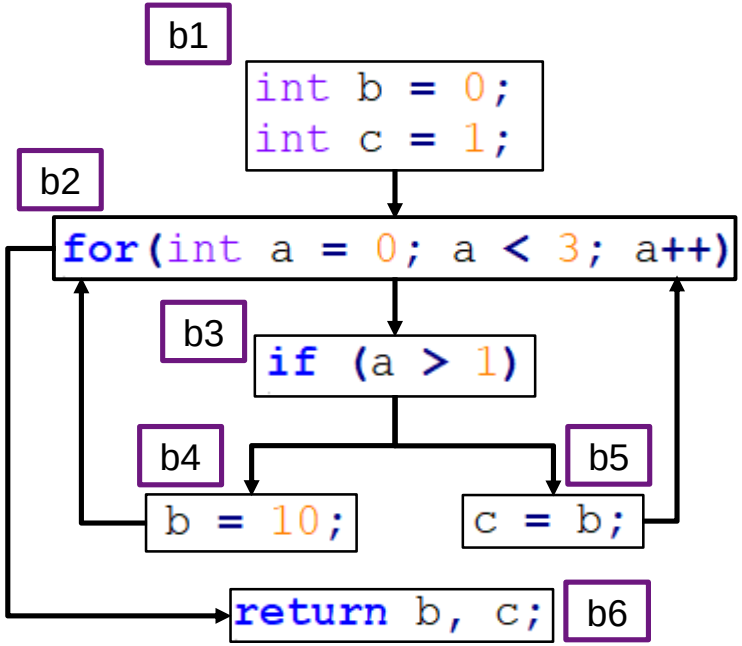

b1
int b = 0;
int c = 1;
b2
for(int a = 0; a < 3; a++)
b3
if (a > 1)
b4
b5
b = 10;
c = b;
return b, c;
b6



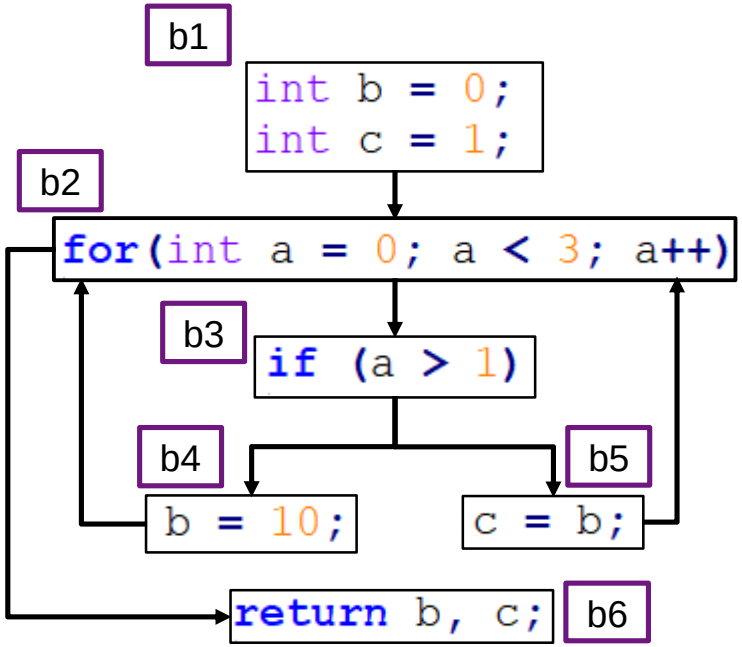
	a	b	c
b1			
b2			
b3			
b4			
b5			
b6			



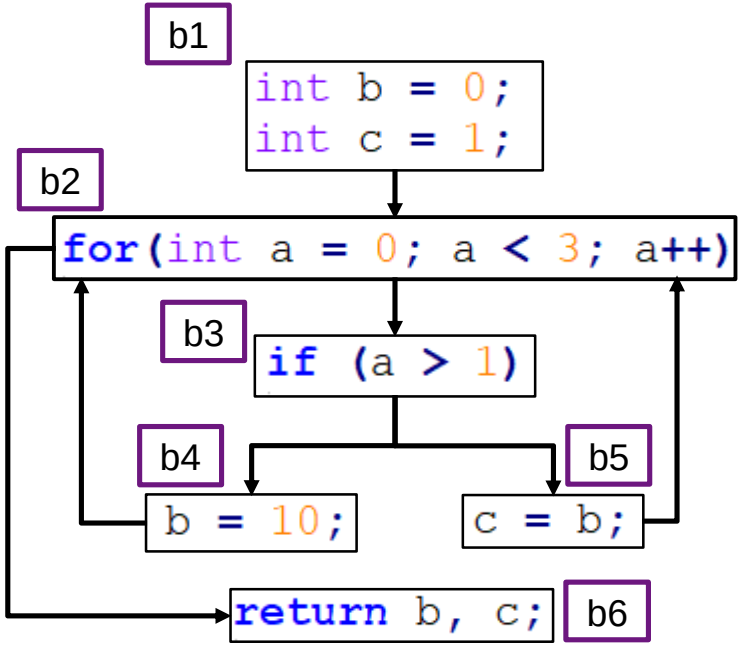
	a	b	c
b1	-	0	1
b2			
b3			
b4			
b5			
b6			



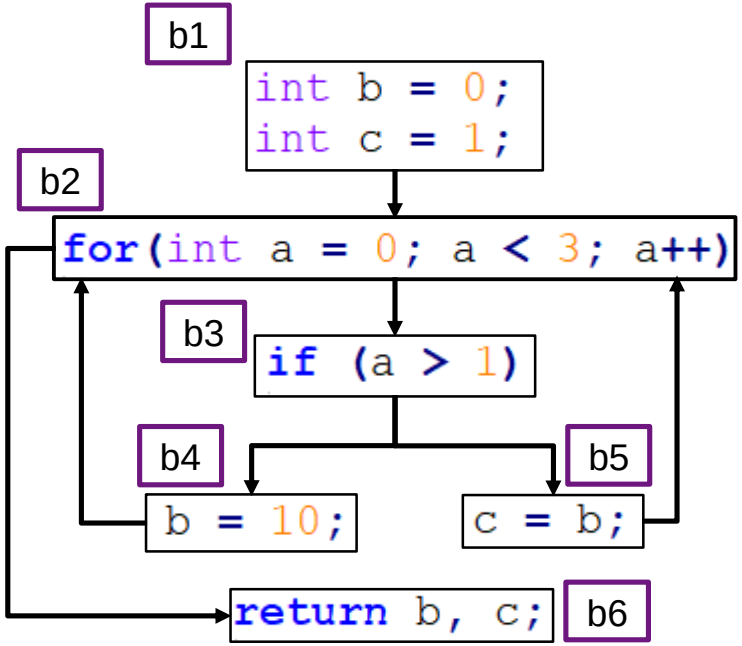
	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3			
b4			
b5			
b6			



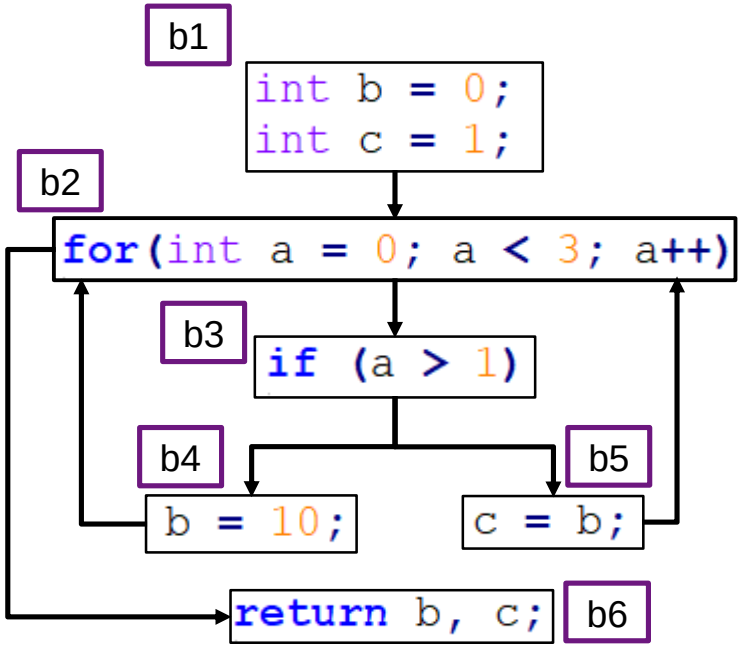
	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4			
b5			
b6			



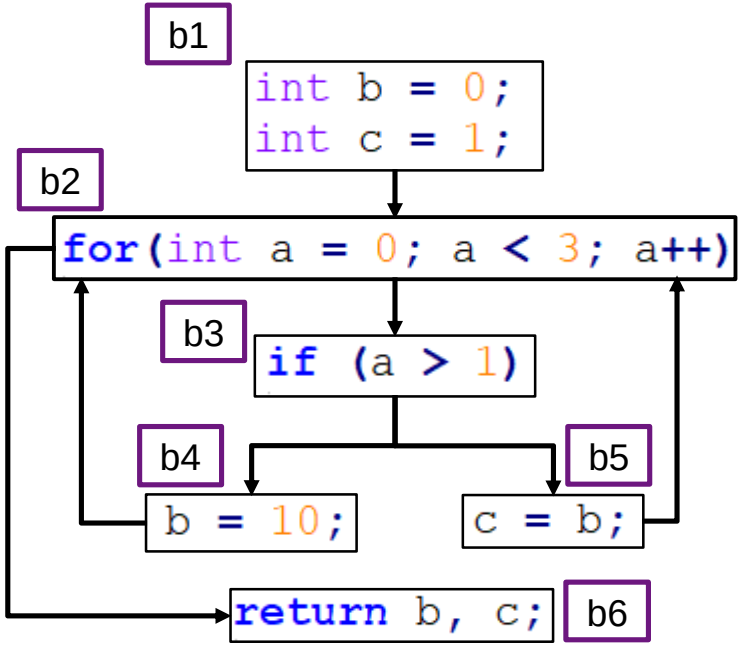
	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5			
b6			



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6			

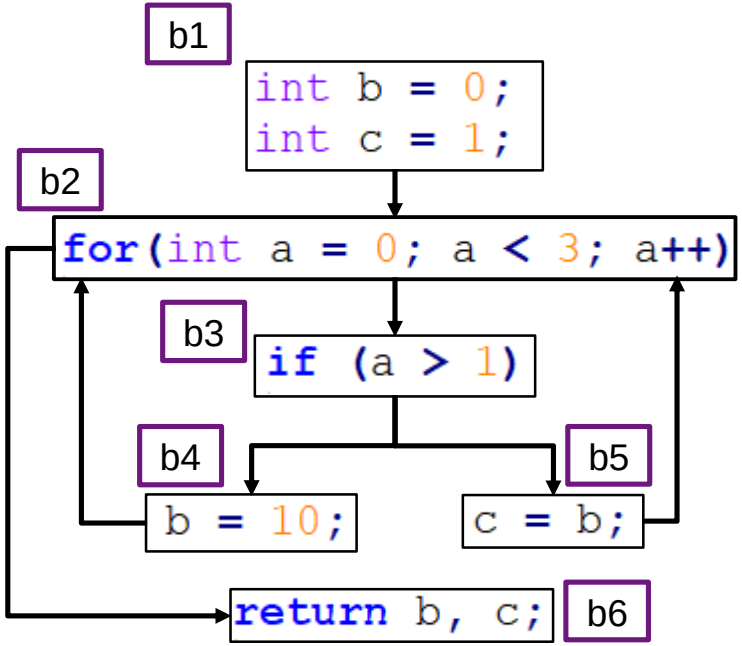


	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

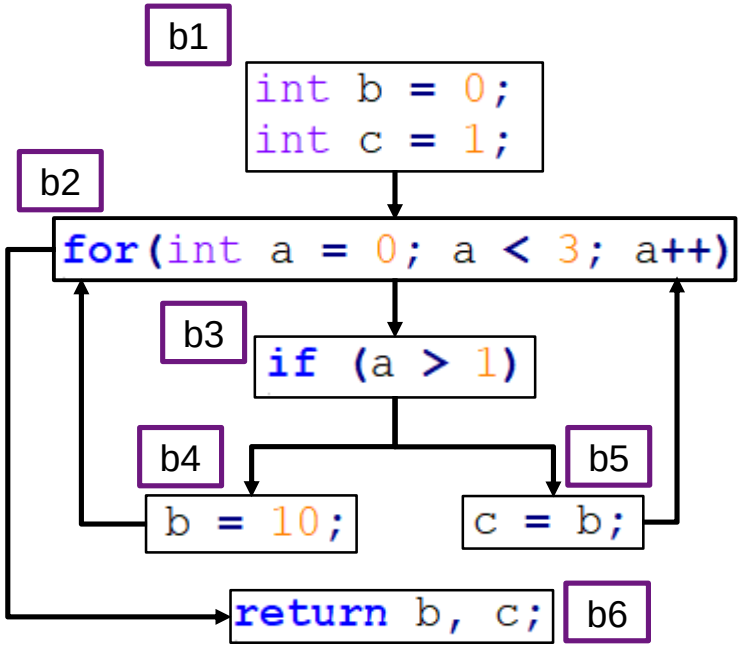
Data Flow Analysis



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

Data Flow Analysis

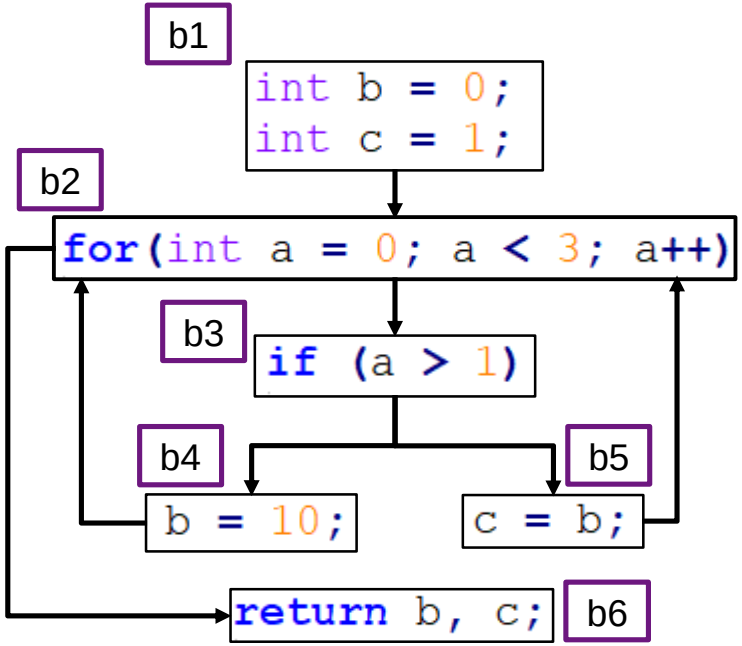
a = {0, 1, 2, 3, …}



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

Data Flow Analysis

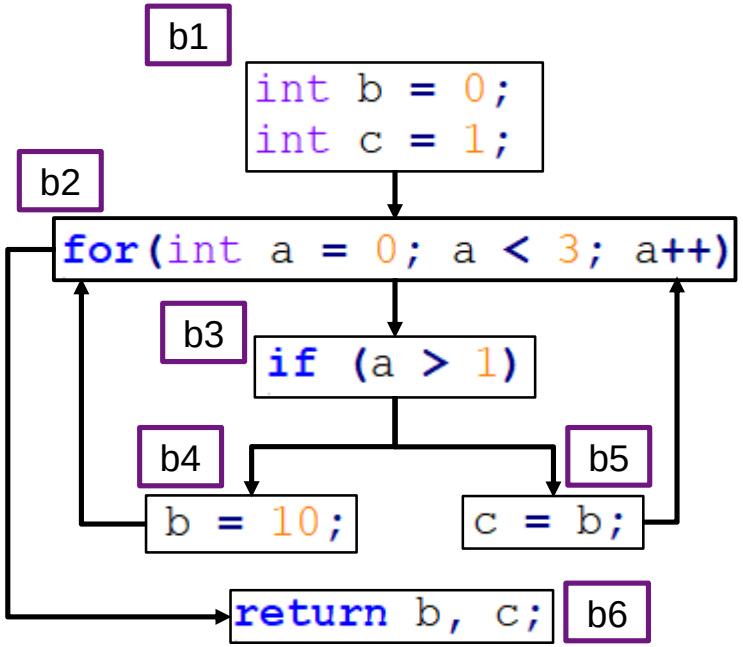
a = {0, 1, 2, 3, …}
b = {0, 10}



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

Data Flow Analysis

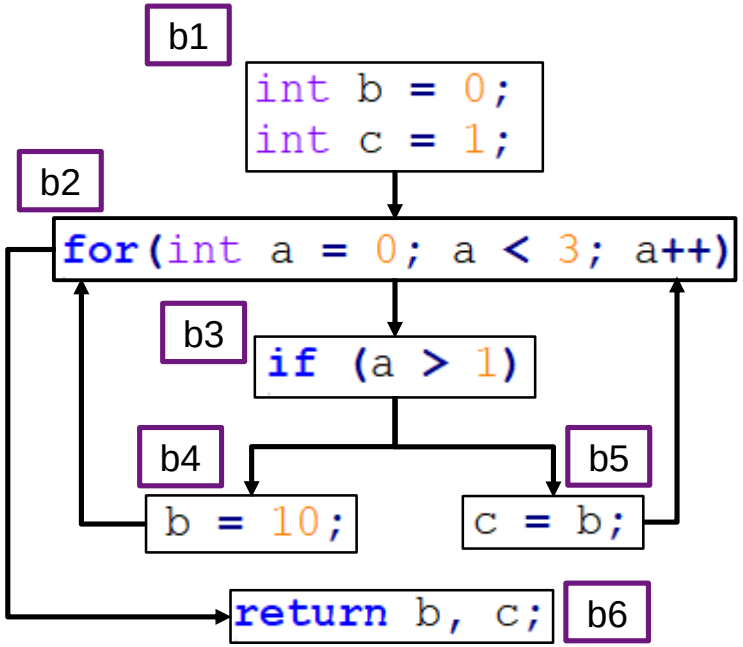
a = {0, 1, 2, 3, …}
b = {0, 10}
c = {1, b} → {0, 1, 10}



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

Data Flow Analysis

a = {0, 1, 2, 3, …}
b = {~~0~~, 10}
c = {1, b} → {0, ~~1~~, ~~10~~}



	a	b	c
b1	-	0	1
b2	0, a++	-	-
b3	-	-	-
b4	-	10	-
b5	-	-	b
b6	-	-	-

Data Flow Analysis

a = {0, 1, 2, 3, …}
b = {~~0~~, 10}
c = {1, b} → {0, ~~1~~, ~~10~~}

Sound but imprecise

Data Flow Analysis in Security

- Source/Sink clash

Data Flow Analysis in Security

- Source/Sink clash
  - Sanitization problems
  - Code injection (Update attack)
  - Deserialization vulnerability

Data Flow Analysis in Security

- Source/Sink clash
  - Sanitization problems
  - Code injection (Update attack)
  - Deserialization vulnerability
- Control and Data flow analysis

Data Flow Analysis in Security

- Source/Sink clash
  - Sanitization problems
  - Code injection (Update attack)
  - Deserialization vulnerability
- Control and Data flow analysis
  - Type confusion vulnerability
  - Use-after-free vulnerability

Data Flow Analysis in Security

- Source/Sink clash
  - Sanitization problems
  - Code injection (Update attack)
  - Deserialization vulnerability
- Control and Data flow analysis
  - Type confusion vulnerability
  - Use-after-free vulnerability

- *Denial of Service??*
- *Crashes??*

Static Analysis Tools

- Open source
  - 
  - checkstyle
  -  SpotBugs
  -  FindSecBugs
- Proprietary
  -  Coverity
  -  CheckMarx

Static Analysis Tools

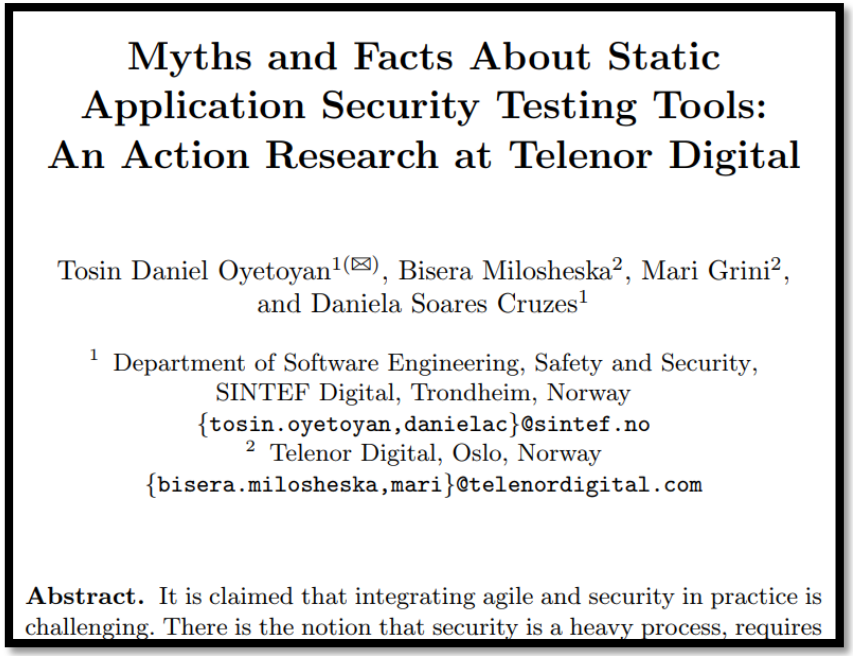
- Open source
  - PMD
    - Ruleset based code checker
  - checkstyle
    - Checks coding standards
  - SpotBugs
    - Checks Java bytecode for bad practices, code style, and injections
  - FindSecBugs
    - Checks for OWASP Top 10 vulnerabilities
- Proprietary
  - Coverity
    - SAST platform for defects and security vulnerabilities
  - CheckMarx
    - Full fledge platform for static analysis and exposure management

Static Analysis Tools

- Open source
  - PMD
    - Ruleset based code checker
  - checkstyle
    - Checks coding standards
  - SpotBugs
    - Checks Java bytecode for bad practices, code style, and injections
  - FindSecBugs
    - Checks for OWASP Top 10 vulnerabilities
- Proprietary
  - Coverity
    - SAST platform for defects and security vulnerabilities
  - CheckMarx
    - Full fledge platform for static analysis and exposure management

SAST Tools Performance

- Telenor Digital wants to incorporate security into SDLC
- Investigate developer perceptions of SAST tools



Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital

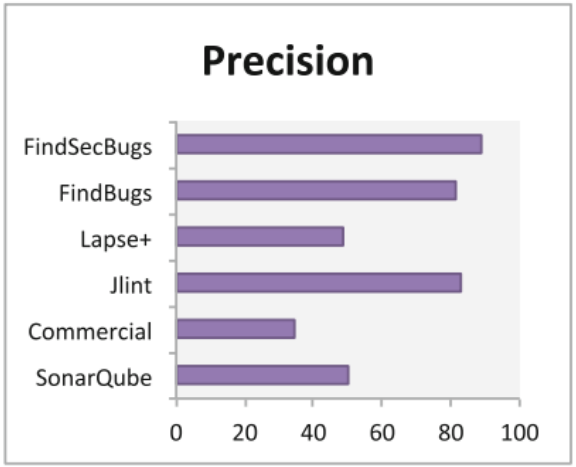
Tosin Daniel Oyetoyan[1(✉)], Bisera Milosheska[2], Mari Grini[2], and Daniela Soares Cruzes[1]

[1] Department of Software Engineering, Safety and Security, SINTEF Digital, Trondheim, Norway
{tosin.oyetoyan,danielac}@sintef.no
[2] Telenor Digital, Oslo, Norway
{bisera.milosheska,mari}@telenordigital.com

Abstract. It is claimed that integrating agile and security in practice is challenging. There is the notion that security is a heavy process, requires

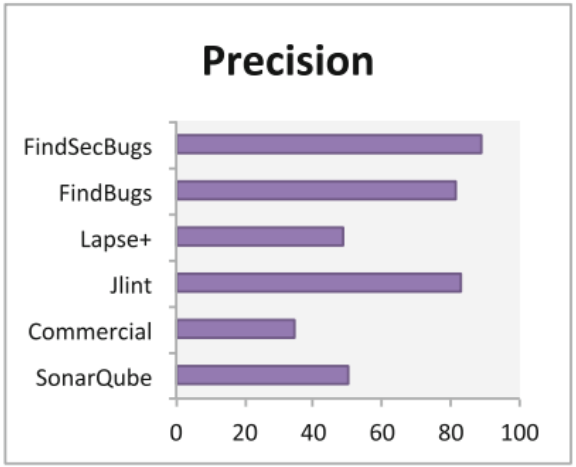
SAST Tools Performance

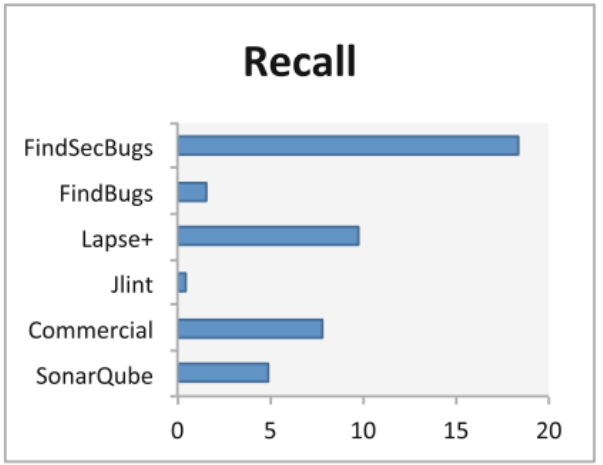
- Using Juliet Test Suite – 24,000 test cases
- **Precision** – Ability to guess correct type of flaw



SAST Tools Performance

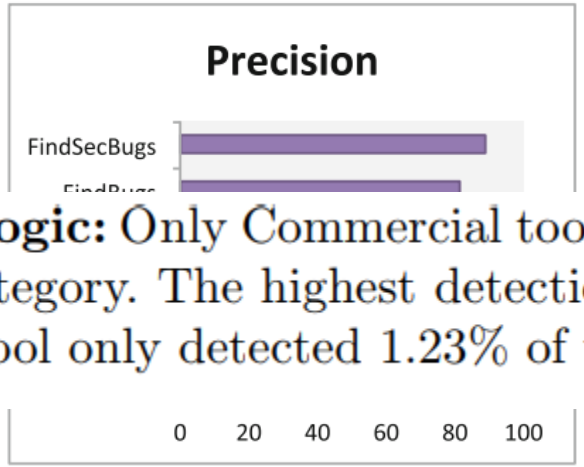
- Using Juliet Test Suite – 24,000 test cases
- **Precision** – Ability to guess correct type of flaw
- **Recall** – Ability to find flaws

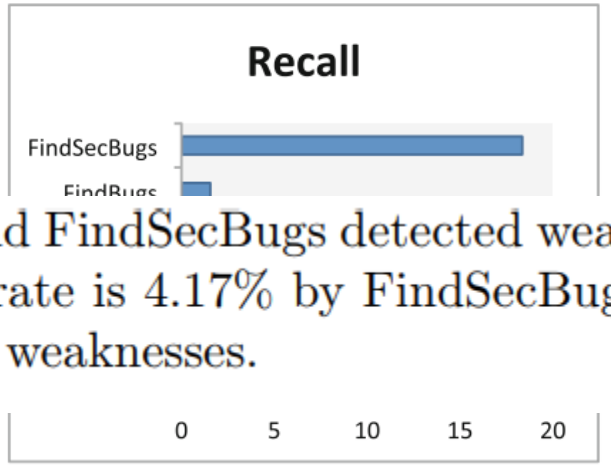




SAST Tools Performance

- Using Juliet Test Suite – 24,000 test cases
- **Precision** – Ability to guess correct type of flaw
- **Recall** – Ability to find flaws





Malicious Logic: Only Commercial tool and FindSecBugs detected weaknesses under this category. The highest detection rate is 4.17% by FindSecBugs while commercial tool only detected 1.23% of the weaknesses.

SAST Dev Perceptions

- *“. . . Making the things actually work, that usually is the worst thing. The hassle-factor is not to be underestimated. . . ”*
- *“. . . At least from my experience with the Sonar tool is that it sometimes complains about issues that are not really issues...”*
- “*. . . And of course in itself is not productive, nobody gives you a hug after fixing SonarQube reports...*”

SAST Dev Perceptions

- *“. . . Making the things actually work, that usually is the worst thing. The hassle-factor is not to be underestimated. . . ”*
- *“. . . At least from my experience with the Sonar tool is that it sometimes complains about issues that are not really issues...”*
- “*. . . And of course in itself is not productive, nobody gives you a hug after fixing SonarQube reports...*”

- Using one SAST tool is not enough
- Low capability of SAST tools in general.
- Commercial tool not an exception

Summary Part II

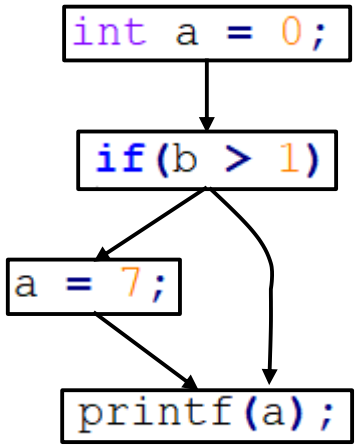
- Perfect static analysis is not possible
- Pattern matching can find limited but easy to find problems
- ASTs make code structure analysis easy
- Control and Data FGs are better at finding security vulnerabilities
- Current SAST Tools are
  - Useful
  - Difficult to integrate
  - Limited in capabilities

Additional Material

- https://www.theserverside.com/feature/Stay-ahead-of-Java-security-issues-like-SQL-and-LDAP-injections
- https://www.upguard.com/articles/top-10-java-vulnerabilities-and-how-to-fix-them
- https://en.wikipedia.org/wiki/Static_program_analysis
- https://youtu.be/Heor8BVa4A0
- https://youtu.be/7KCMK-LY-WM
- Aktas, Kursat, and Sevil Sen. "UpDroid: Updated Android Malware and Its Familial Classification." *Nordic Conference on Secure IT Systems*. Springer, Cham, 2018.

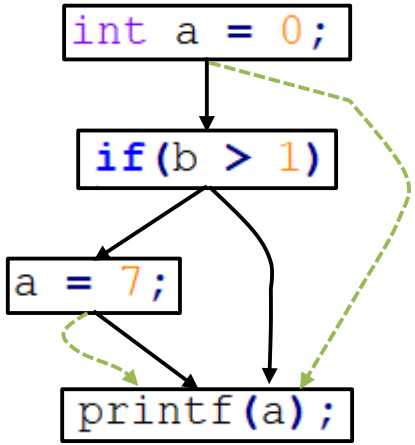
Time for questions

Data Flow Analysis



→ Control

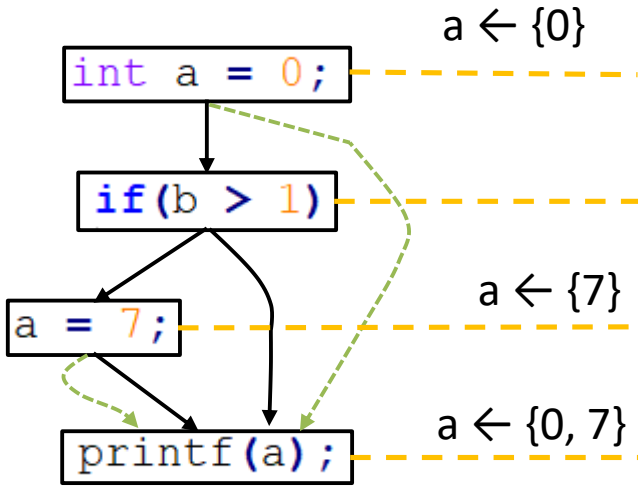
Data Flow Analysis



Control

Data

Data Flow Analysis



Control

Data

Overflow vulnerability

- This vulnerability allows remote attackers to execute arbitrary code on vulnerable installations of Oracle Java. The user must visit a malicious page or open a malicious file to exploit this vulnerability.
- The flaw exists within the handling of image data. The issue lies in insufficient validation of supplied image data inside the native function readImage(). An attacker can leverage this vulnerability to execute arbitrary code under the context of the current process.

https://www.zerodayinitiative.com/advisories/ZDI-16-032/