# Static Analysis

## Dataflow Analysis

# Roadmap

- **Overview.**
- Four Analysis Examples.
- Analysis Framework – Soot.
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- Taint Analysis.

# Overview

- Static analysis is a program analysis technique performed without actually executing programs.
- Data flow analysis is a process of deriving information about the run time behavior of a program.
- Usage: compiler, IDE and security.

# SSA

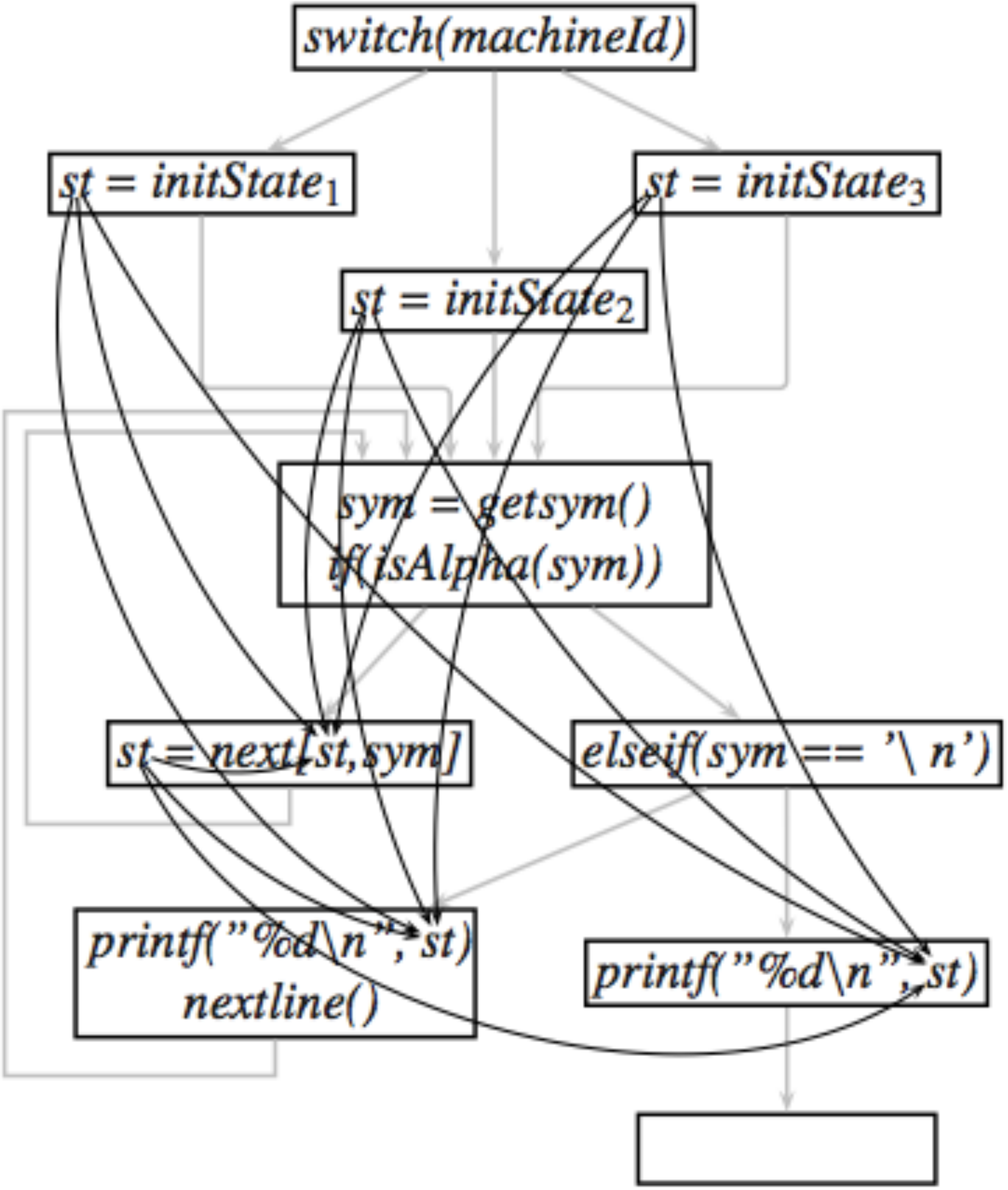
- Requires that each variable is assigned exactly once.

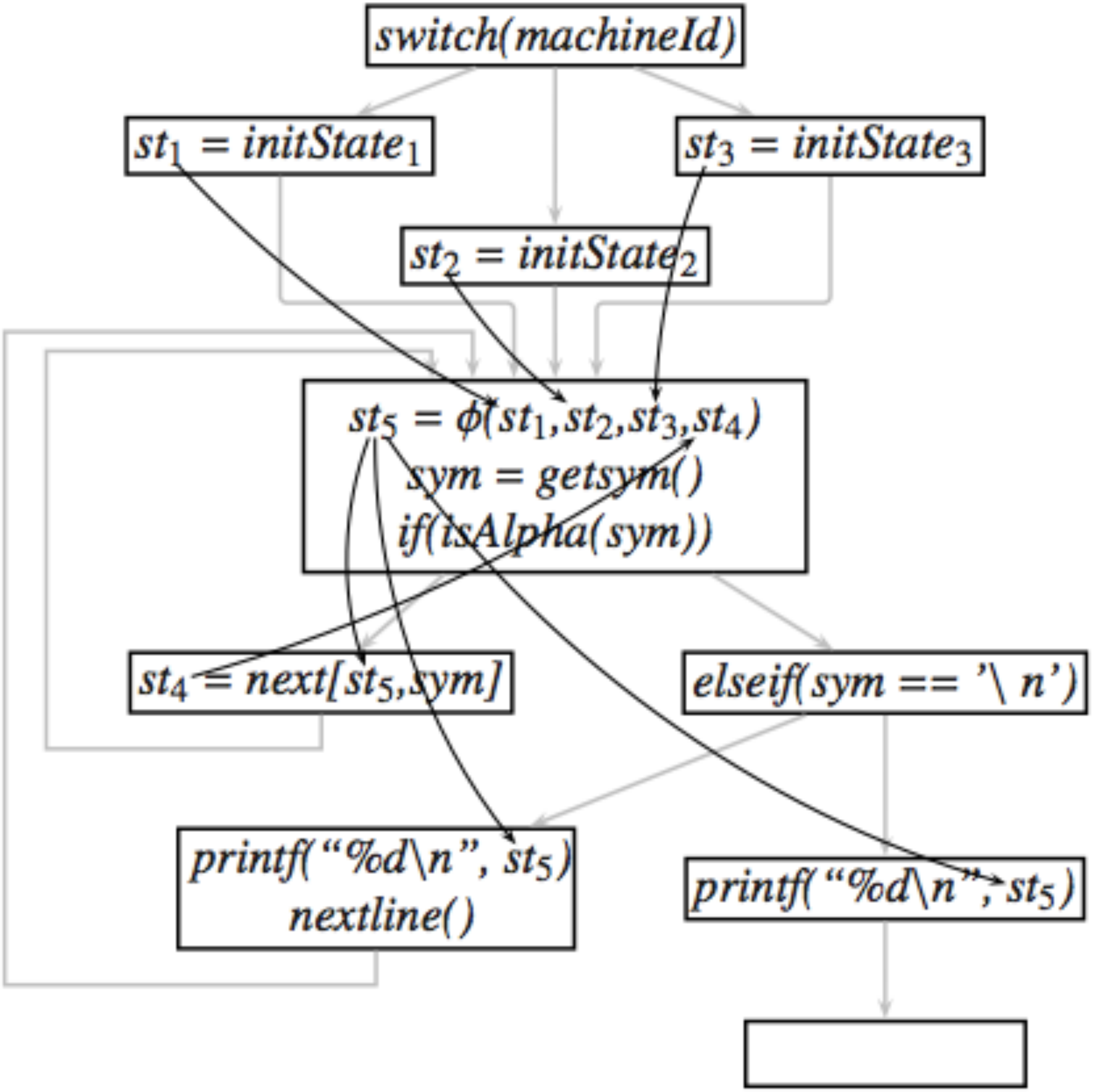
```
y := 1
y := 2
x := y
```

```
y1 := 1
y2 := 2
x1 := y2
```

- Def-use chain:
  - Def-use chains are used to propagate data flow information.
  - The analysis algorithm takes time proportional to the product of the total number of def-use edges
- Benefits:
  - Data flow analysis could be easier and faster.
  - Reduce the number of def-use chains. (m*n vs m+n)

```
switch(machineId)
{  case1:
     st = initState1;
     break;
   case2:
     st = initState2;
     break;
   case3:
     st = initState3;
}
while (1)
{  sym = getsym();
   if(isAlpha(sym))
     st = next[st,sym];
   elseif(sym == '\n')
   { printf("%d\n", st);
     nextline();
   }
   else
   { printf("%d\n", st);
     break;
   }
}
```

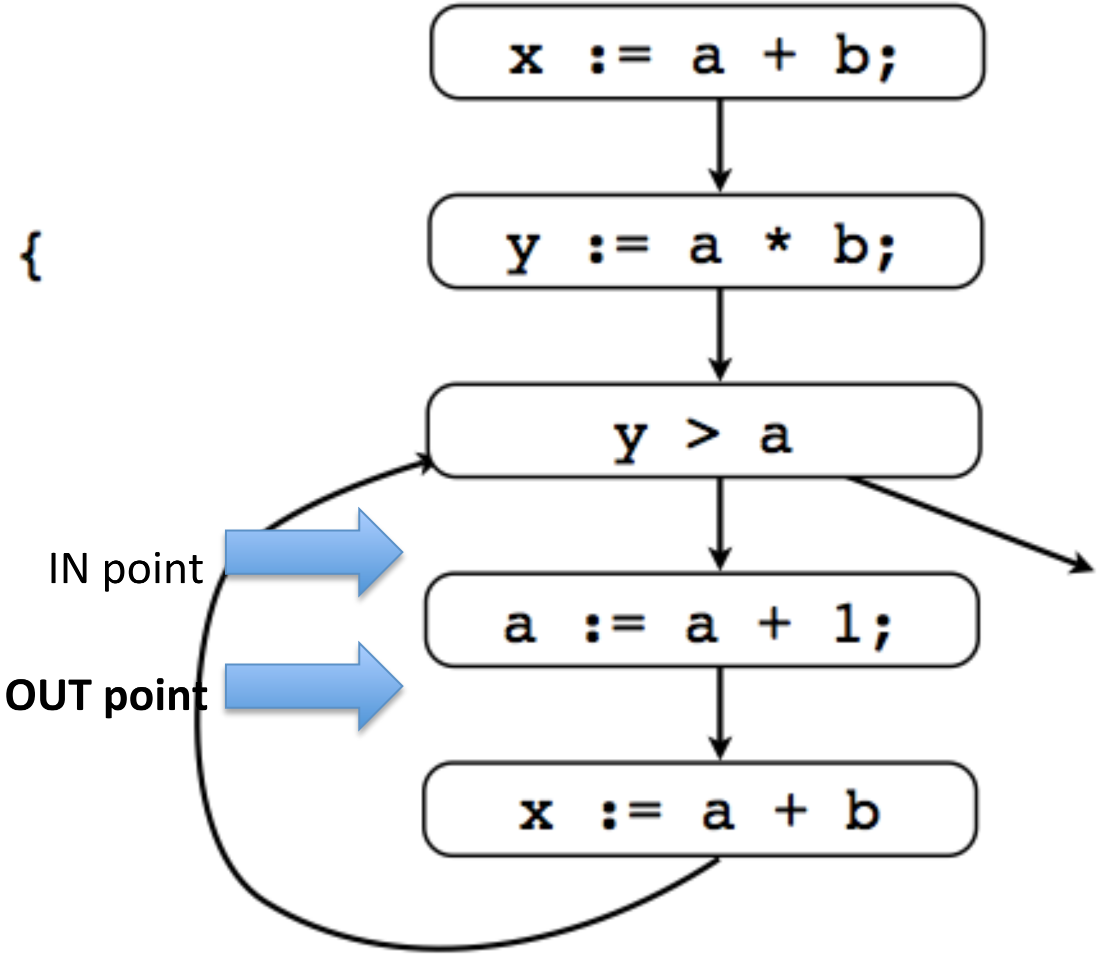



switch(machineId)
st1 = initState1
st3 = initState3
st2 = initState2
st5 = φ(st1,st2,st3,st4)
sym = getsym()
if(isAlpha(sym))
st4 = next[st5,sym]
elseif(sym == '\ n')
printf("%d\n", st5)
nextline()
printf("%d\n", st5)

# Control Flow Graph (CFG)

- A control flow graph is a representation of a program that makes certain analyses (including dataflow analyses) easier.
- Usually built on Intermediate representation:
  – Single static assignment (SSA) form.
- Statements may be
  – Assignments: x := y or x := y op z or x := op y
  – Branches: goto L or if b then goto L
- A directed graph where
  – Each node represents a statement
  – Edges represent control flow

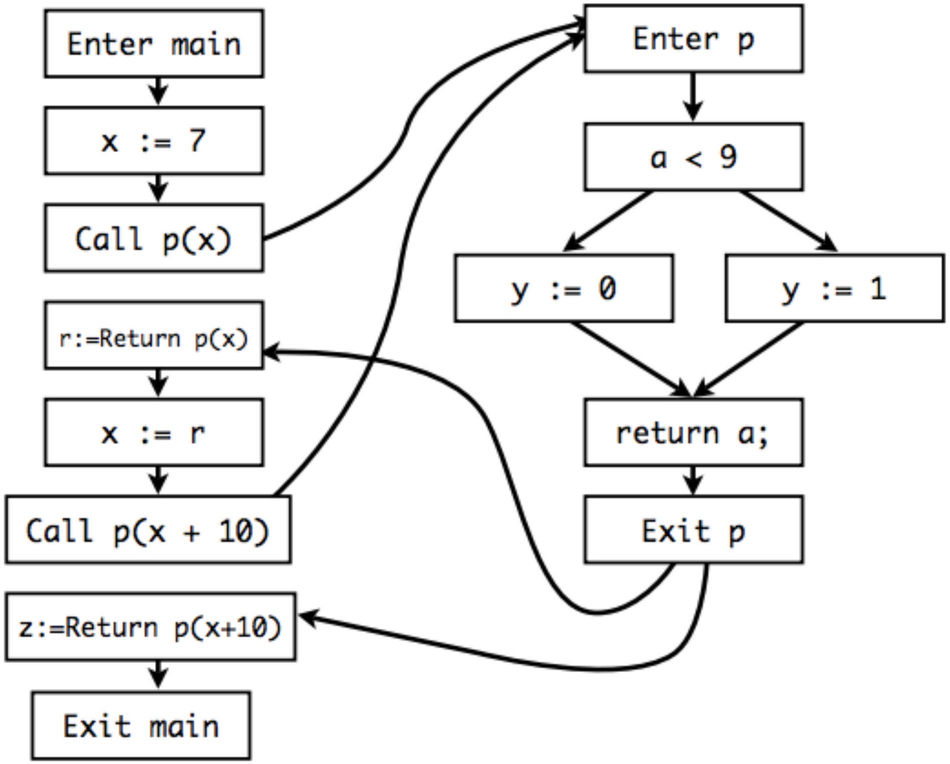
```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



# Inter-procedure Analysis

- How do we deal with procedure calls?
- Obvious idea: make one big CFG

```
main() {
  x := 7;
  r := p(x);
  x := r;
  z := p(x + 10);
}

p(int a) {
  if (a < 9)
    y := 0;
  else
    y := 1;
  return a;
}
```

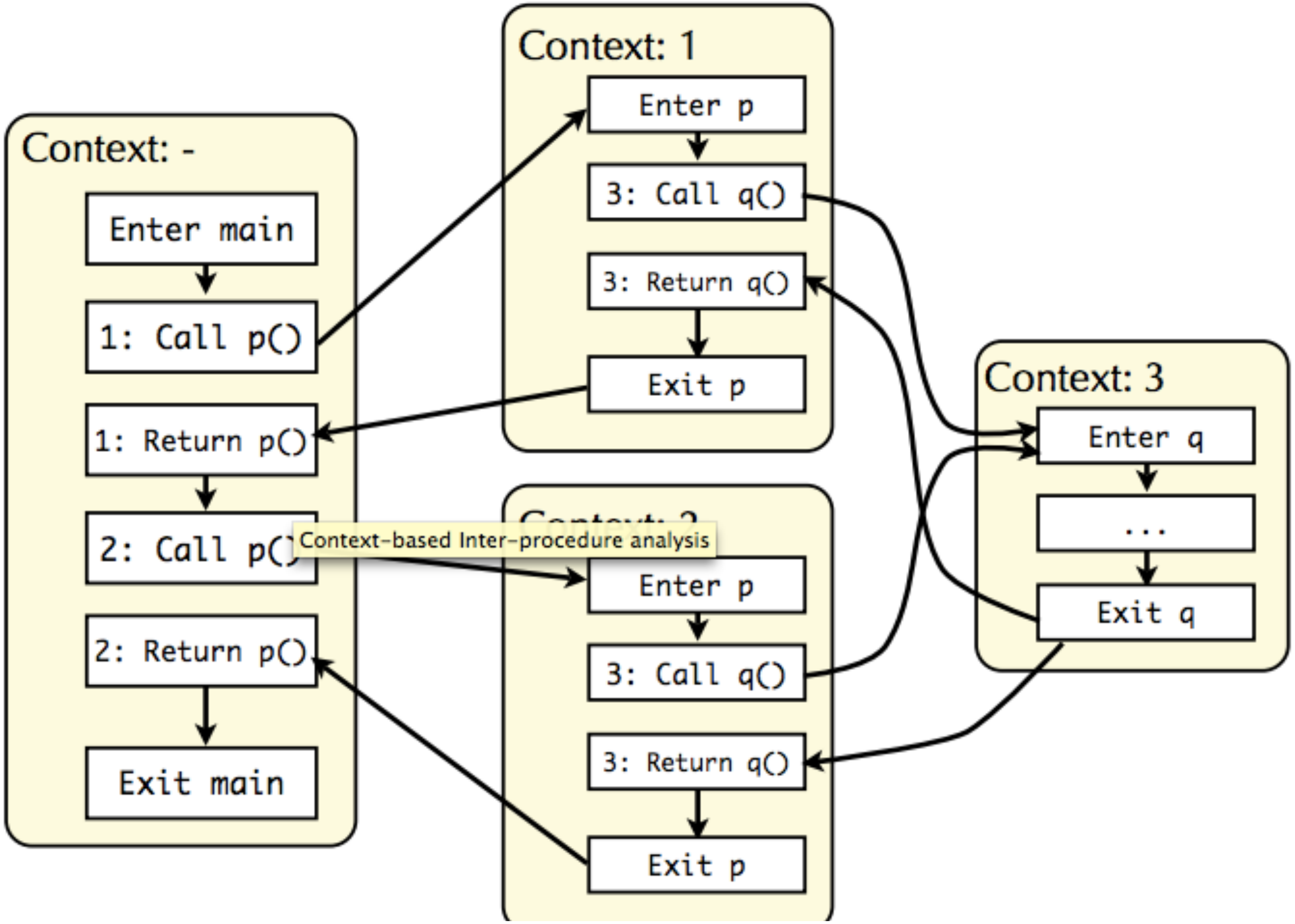


# Context-based Inter-procedure analysis

- Solution: make a finite number of copies
- Use context information to determine when to share a copy
- Choice of what to use for context will produce different tradeoffs between precision and scalability
- Common choice:
  - Call site
  - Parameter value

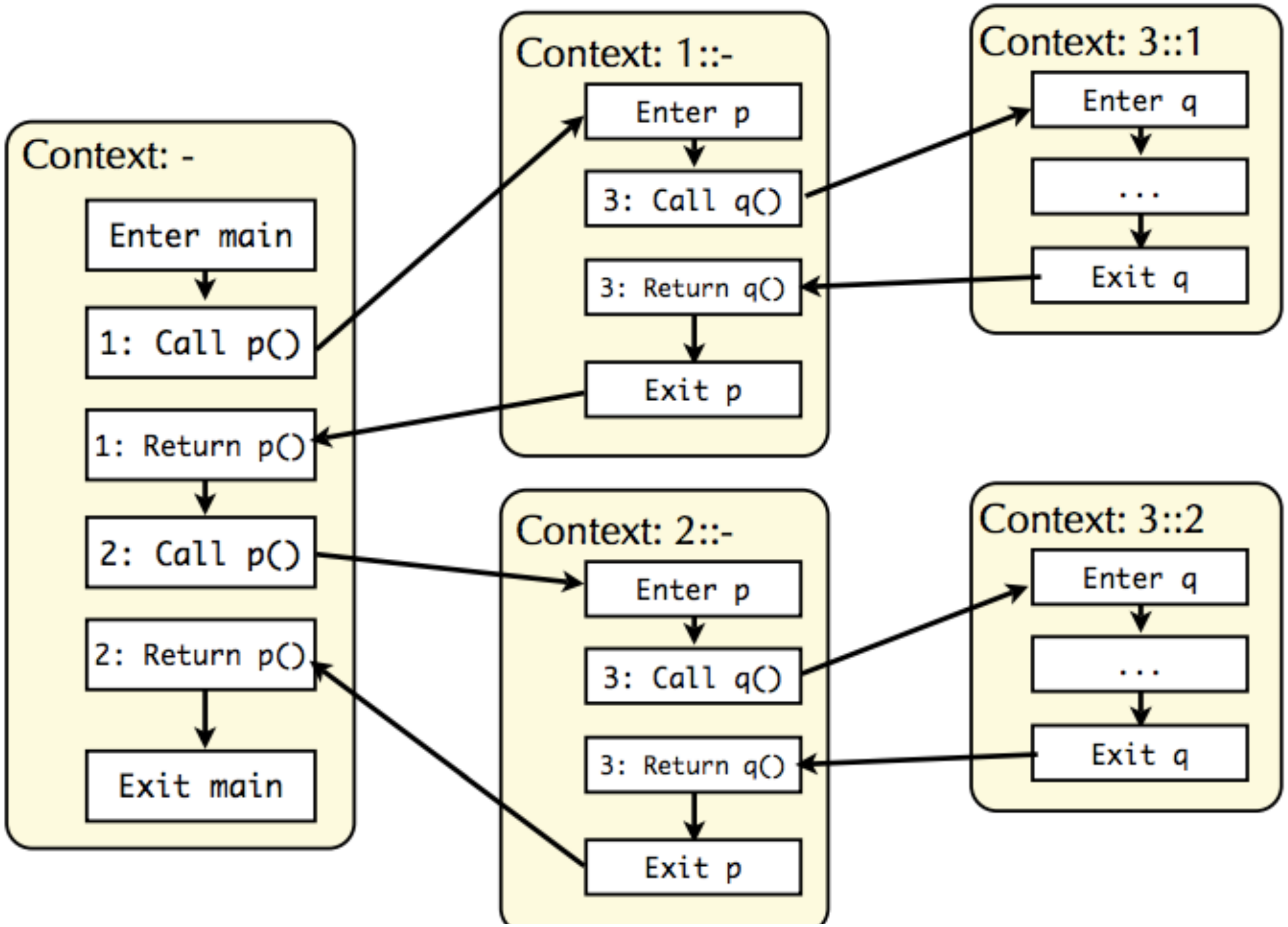
# Based on Call Stack Depth 1

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```

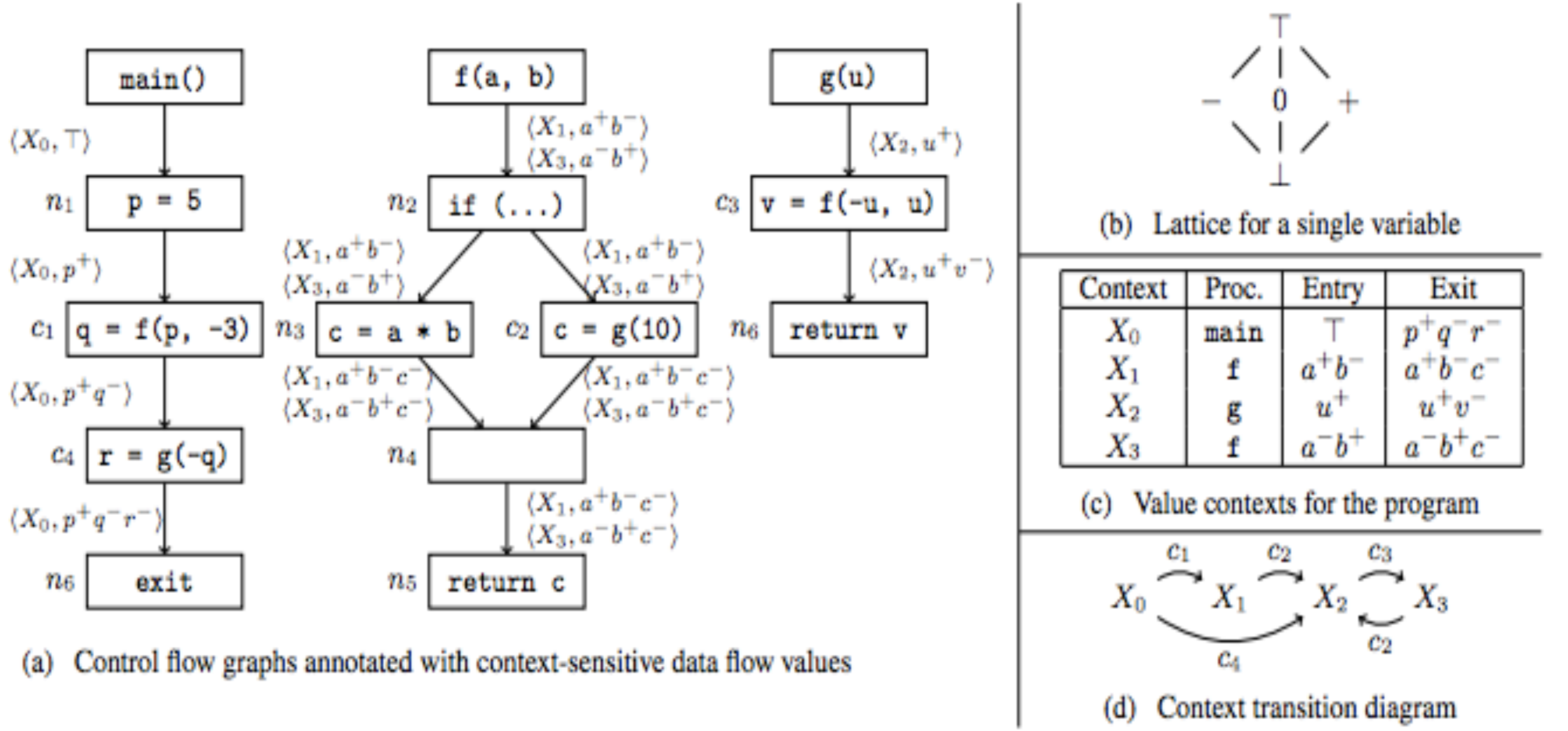


# Based on Call Stack Depth 2

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```



# Based on Parameter Value


main()
$\langle X_0, \top\rangle$
$n_1$
p = 5
$\langle X_0, p^+\rangle$
$c_1$
q = f(p, -3)
$\langle X_0, p^+q^-\rangle$
$c_4$
r = g(-q)
$\langle X_0, p^+q^-r^-\rangle$
$n_6$
exit
f(a, b)
$\langle X_1, a^+b^-\rangle$
$\langle X_3, a^-b^+\rangle$
$n_2$
if (...)
$\langle X_1, a^+b^-\rangle$
$\langle X_3, a^-b^+\rangle$
$\langle X_1, a^+b^-\rangle$
$\langle X_3, a^-b^+\rangle$
$n_3$
c = a * b
$c_2$
c = g(10)
$\langle X_1, a^+b^-c^-\rangle$
$\langle X_3, a^-b^+c^-\rangle$
$\langle X_1, a^+b^-c^-\rangle$
$\langle X_3, a^-b^+c^-\rangle$
$n_4$
$\langle X_1, a^+b^-c^-\rangle$
$\langle X_3, a^-b^+c^-\rangle$
$n_5$
return c
g(u)
$\langle X_2, u^+\rangle$
$c_3$
v = f(-u, u)
$\langle X_2, u^+v^-\rangle$
$n_6$
return v
$\top$
$-$
$0$
$+$
$\bot$
$X_0$
$X_1$
$X_2$
$X_3$
$c_1$
$c_2$
$c_3$
$c_2$
$c_4$

(a) Control flow graphs annotated with context-sensitive data flow values

(b) Lattice for a single variable

| Context | Proc. | Entry | Exit |
|---|---|---|---|
| $X_0$ | main | $\top$ | $p^+q^-r^-$ |
| $X_1$ | f | $a^+b^-$ | $a^+b^-c^-$ |
| $X_2$ | g | $u^+$ | $u^+v^-$ |
| $X_3$ | f | $a^-b^+$ | $a^-b^+c^-$ |

(c) Value contexts for the program

(d) Context transition diagram

**Figure 2.** A motivating example of a non-distributive sign-analysis performed on a program with mutually recursive procedures.

# Roadmap

- Overview**.**
- Example.
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- **Taint Analysis**.
- Analysis Framework – Soot.

# Taint Analysis

- Follow any application inside a debugger and you will see that data information is being copied and modified all the time. In another words, information is always moving.
- Taint analysis can be seen as a form of Information Flow Analysis.
  - Insert some kind of tag or label for data we are interested in. (taint the data)
  - Track the influence of the tainted object along the execution of the program.
  - Taint relevant data.
  - Obverse if it flows to sensitive functions (sink).

# Taint Analysis

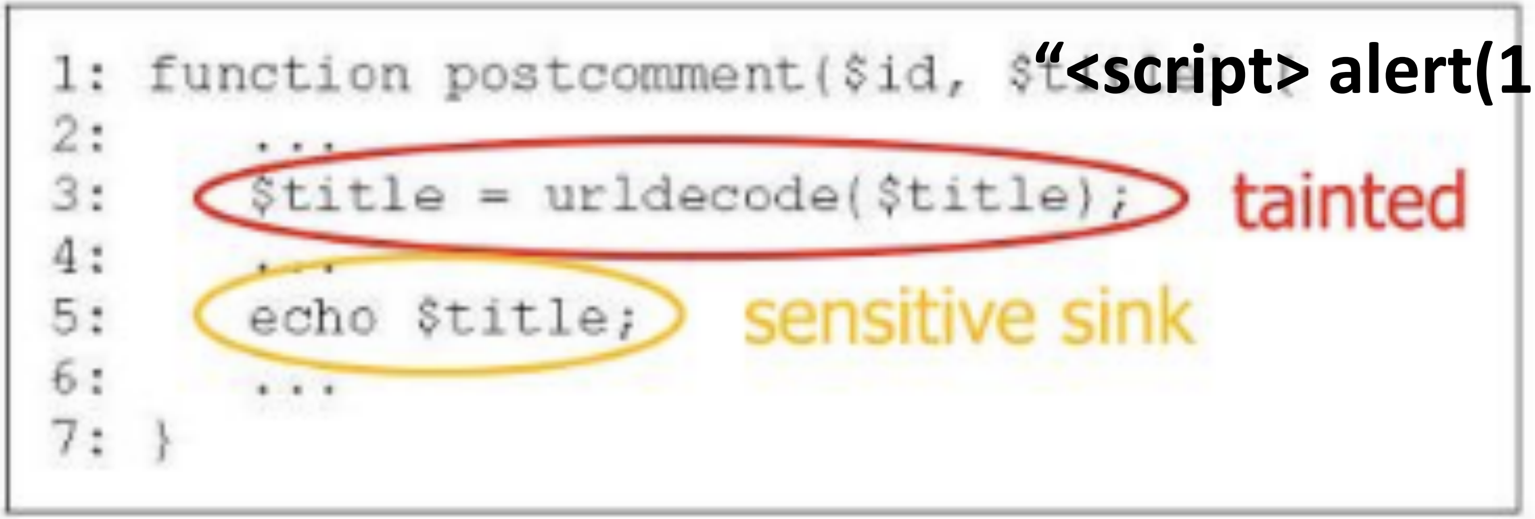
- Two usage in security:
  - **Finding information leakage.**
  - Finding program vulnerability.
- For information leakage:
  - If a data (variable) contains user secrets (e.g., location, contacts), we will taint such data.
  - Taint the variables whose data depend on tainted value. (e.g., a := b + x)
  - Observe if the tainted data will flow to functions that might send data to other places.

```
public class LeakageApp extends Activity{
    private User user = null;
    protected void onRestart(){
        EditText usernameText = (EditText)findViewById(R.id.username);
        EditText passwordText = (EditText)findViewById(R.id.password);
        String uname = usernameText.toString();
        String pwd = passwordText.toString();
        this.user = new User(uname, pwd);
    }
    //Callback method; name defined in Layout-XML
    public void sendMessage(View view){
        if(user != null){
            Password pwdObject = user.getPwdObject();
            String password = pwdObject.getPassword();
            String obfPwd = ""; //must track primitives
            for(char c : password.toCharArray())
                obfPwd += c + "_"; //must handle concat.

            String message = "User: " +
               user.getUsername() + " | Pwd: " + obfPwd;
            SmsManager sms = SmsManager.getDefault();
            sms.sendTextMessage("+44 020 7321 0905", null,
                message, null, null);
        }
    }
}
```

Listing 1.1: Motivating example

# Taint Analysis

- Two usage in security:
  - Finding information leakage.
  - **Finding program vulnerability (code injection).**
- Application vulnerability:
  - A lot of vulnerabilities are caused by unchecked input from user (attack) sent to sensitive functions.

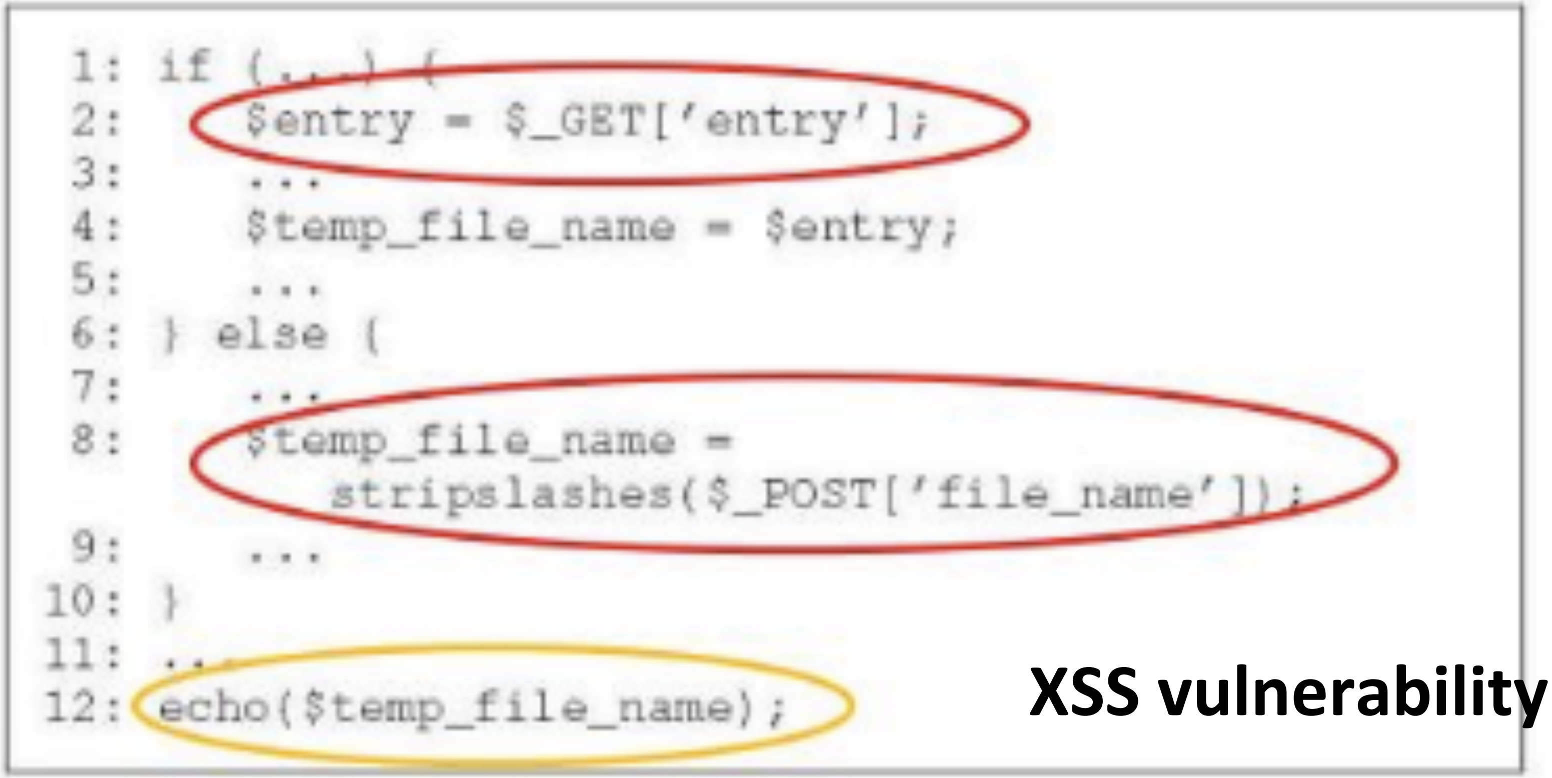


```
function postcomment($id, $title) {
    ...
    $title = urldecode($title);
    ...
    echo $title;
    ...
}
```

“<script> alert(1)</script>

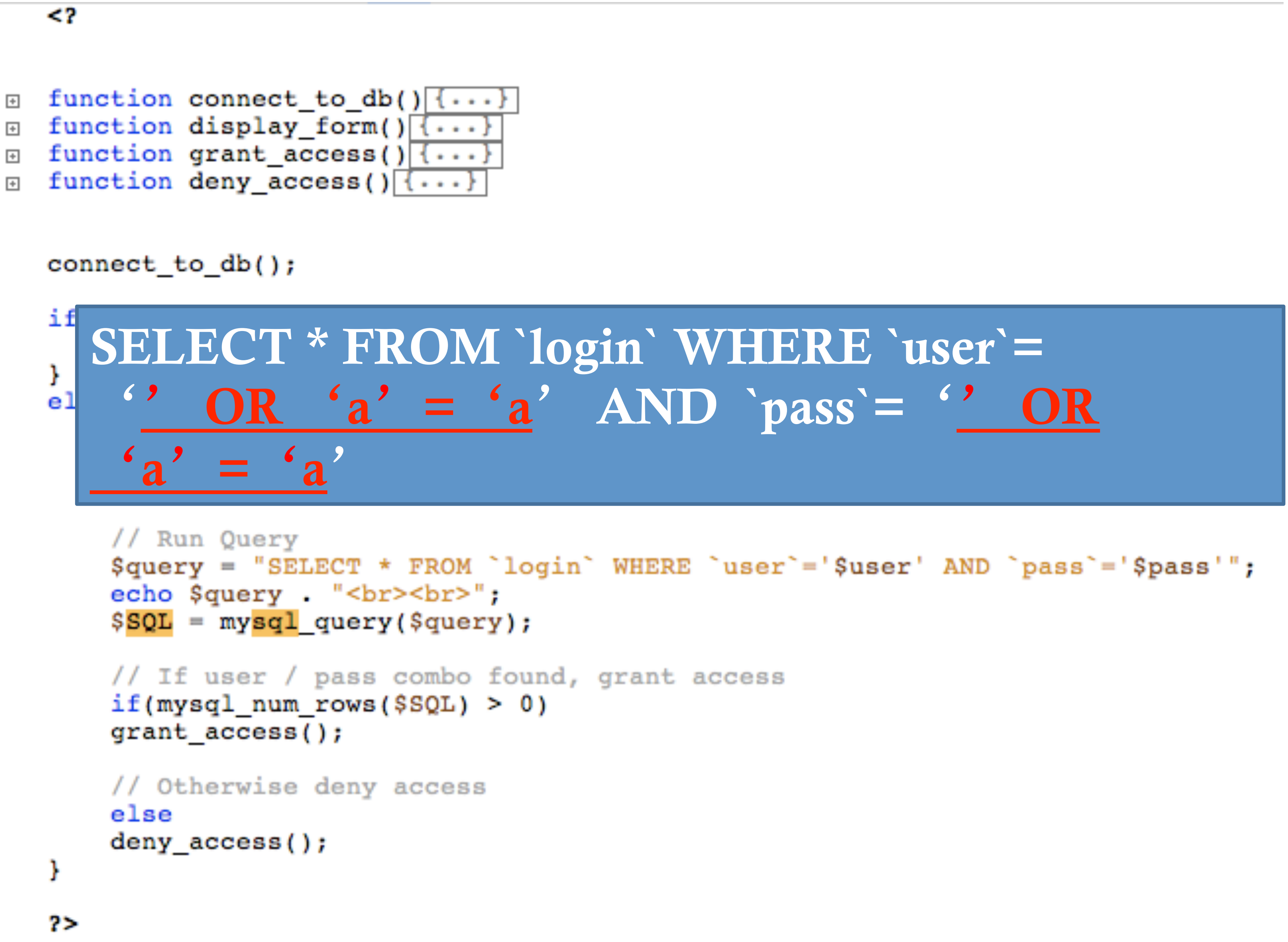
tainted

sensitive sink



```
if (...) {
    $entry = $_GET['entry'];
    ...
    $temp_file_name = $entry;
    ...
} else {
    ...
    $temp_file_name =
        stripslashes($_POST['file_name']);
    ...
}
...
echo($temp_file_name);
```

XSS vulnerability

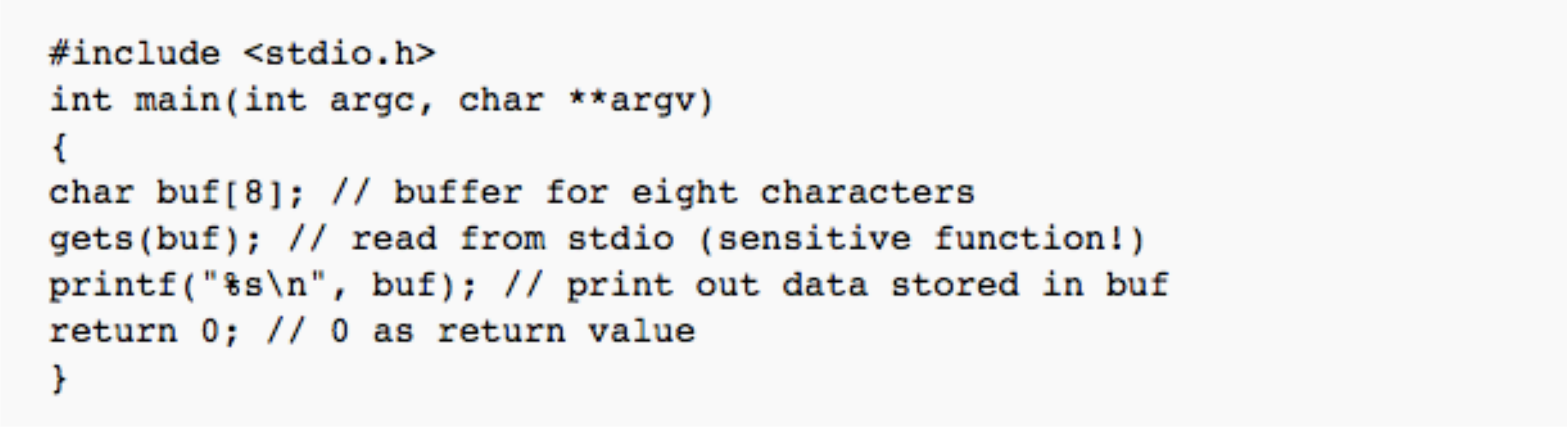


```
<?

function connect_to_db(){...}
function display_form(){...}
function grant_access(){...}
function deny_access(){...}

connect_to_db();

if

}
el
```

SELECT * FROM `login` WHERE `user`= ‘’ OR ‘a’ = ‘a’ AND `pass`= ‘’ OR ‘a’ = ‘a’

```
    // Run Query
    $query = "SELECT * FROM `login` WHERE `user`='$user' AND `pass`='$pass'";
    echo $query . "<br><br>";
    $SQL = mysql_query($query);

    // If user / pass combo found, grant access
    if(mysql_num_rows($SQL) > 0)
    grant_access();

    // Otherwise deny access
    else
    deny_access();
}

?>
```

# Buffer Overflow Vulnerability



```
#include <stdio.h>
int main(int argc, char **argv)
{
char buf[8]; // buffer for eight characters
gets(buf); // read from stdio (sensitive function!)
printf("%s\n", buf); // print out data stored in buf
return 0; // 0 as return value
}
```

# Taint Analysis

- Two usage in security:
  - Finding information leakage.
  - **Finding program vulnerability (code injection).**
- Application vulnerability:
  - A lot of vulnerabilities are caused by unchecked input from user (attack) sent to sensitive functions.
  - If the source of a object X's value is untrusted, we say X is tainted.
  - Taint the variables whose data depend on tainted value. (e.g., a := b + x)
  - Observe if the tainted data will flow to dangerous functions that might lead to execution its parameters.

# Taint Analysis Works

- Android App Information Leakage:
  - FlowDroid.
- JavaScript: Firefox Extension Vulnerability:
  - Bandhakavi, Sruthi, et al. "VEX: Vetting browser extensions for security vulnerabilities." Usenix Security. 2010.
- Php: Web Application Vulnerability:
  - Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities." Oakland, 2006

# References

- Soot Tutorial: https://github.com/Sable/soot/wiki/Tutorials
- Interprocedural Data Flow Analysis in Soot using Value Contexts: https://arxiv.org/pdf/1304.6274.pdf
- Harvard Advanced Programming Language: http://www.seas.harvard.edu/courses/cs252/2011sp/
- Textbool: Data Flow Analysis: Theory and Practice: https://www.amazon.com/Data-Flow-Analysis-Theory-Practice/dp/0849328802
- Course: Professor Finddler's programming Languages seminar.
- Course: Professor Campanoni's code analysis and transformation.