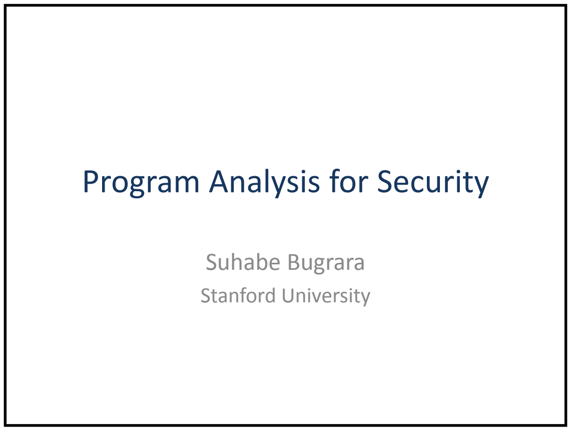
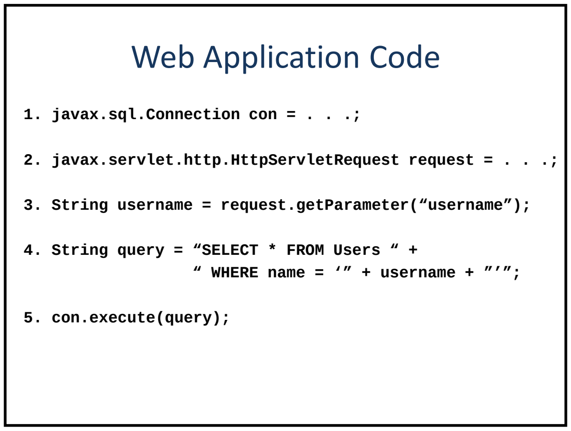


Program Analysis for Security

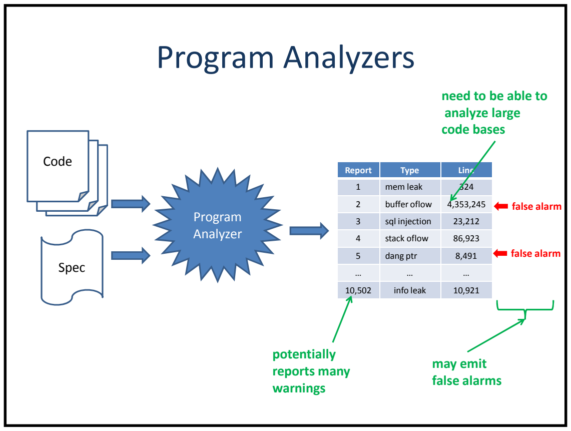
Suhabe Bugrara

Stanford University



Web Application Code

```
javax.sql.Connection con = . . .;

javax.servlet.http.HttpServletRequest request = . . .;

String username = request.getParameter(“username”);

String query = “SELECT * FROM Users “ +
               “ WHERE name = ‘” + username + ”’”;

con.execute(query);
```



Program Analyzers

Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
…	…	…
10,502	info leak	10,921

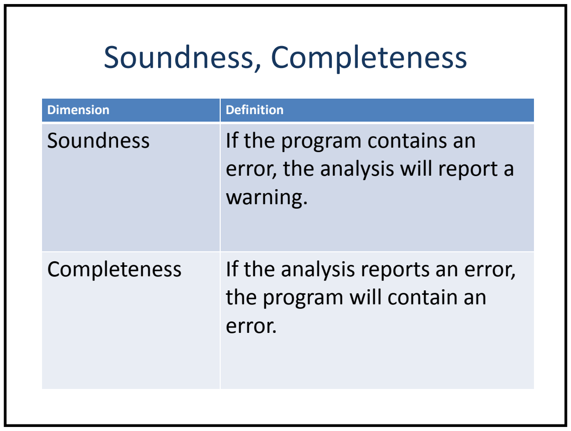
false alarm

false alarm

need to be able to analyze large code bases

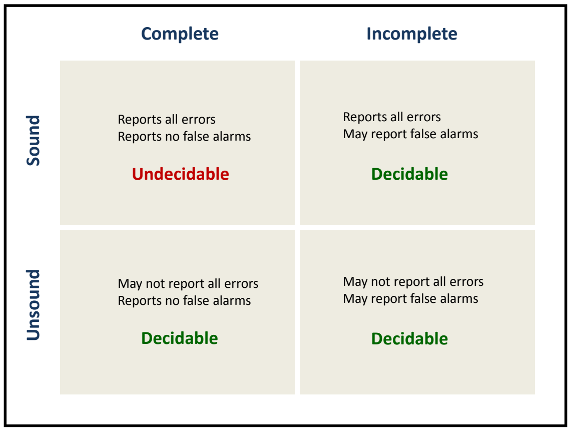
potentially reports many warnings

may emit false alarms

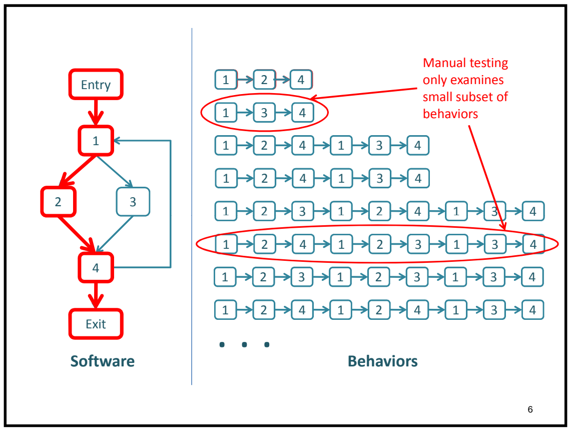


Soundness, Completeness

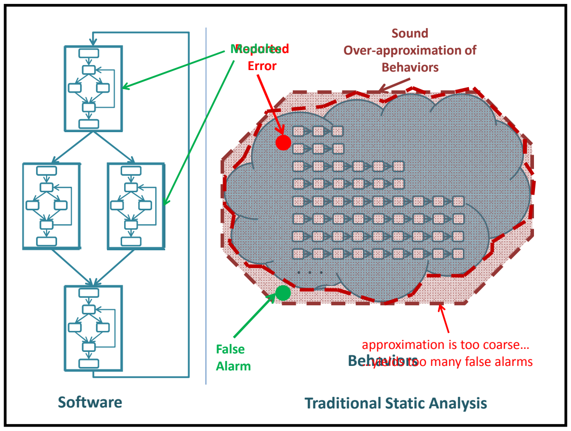
Dimension	Definition
Soundness	If the program contains an error, the analysis will report a warning.
Completeness	If the analysis reports an error, the program will contain an error.

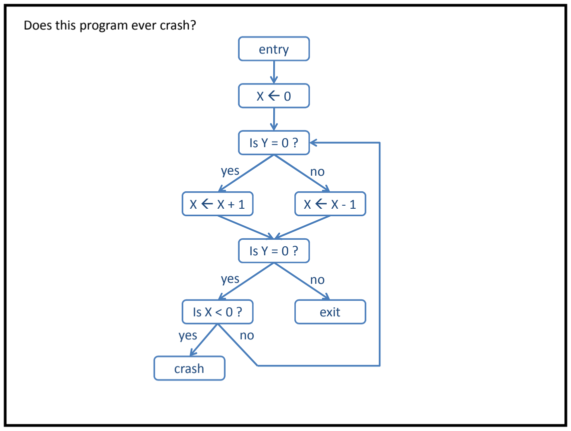


	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

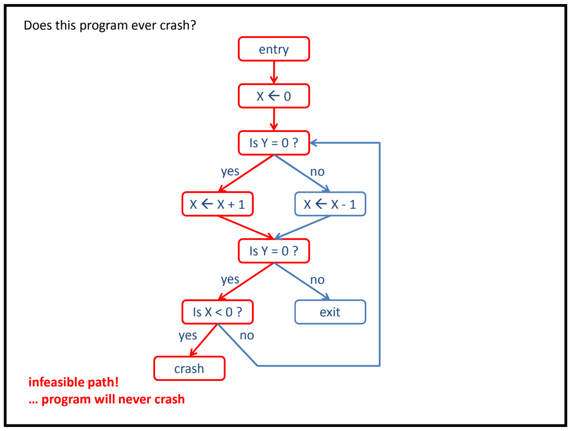


Manual testing only examines small subset of behaviors


Missed
Reported
Error
Sound
Over-approximation of
Behaviors
False
Alarm
approximation is too coarse…
…yields too many false alarms
Behaviors
Software
Traditional Static Analysis

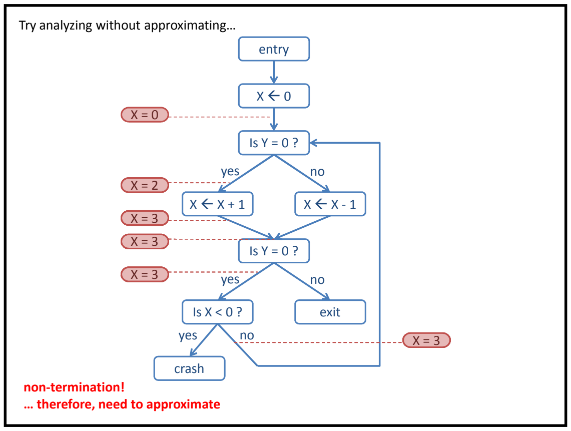


Does this program ever crash?



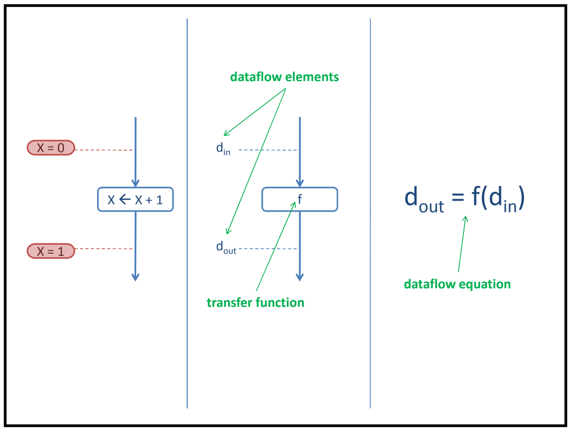
Does this program ever crash?

infeasible path!
… program will never crash

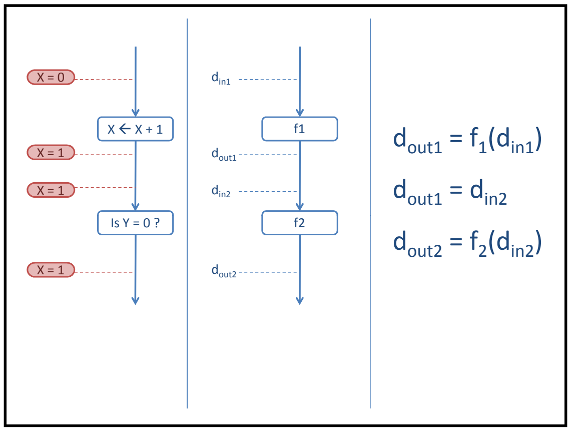


Try analyzing without approximating…

non-termination!
… therefore, need to approximate



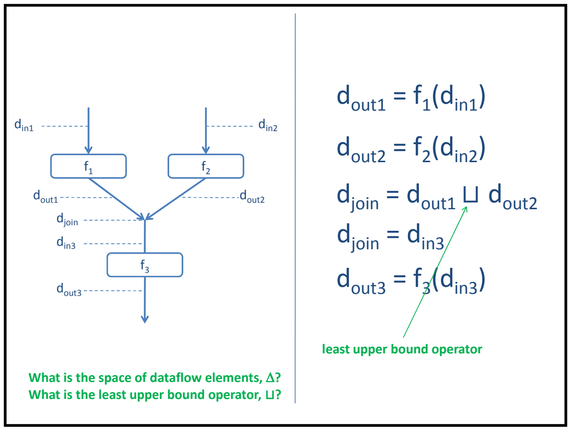
$$d_{out} = f(d_{in})$$



$$d_{out1} = f_1(d_{in1})$$

$$d_{out1} = d_{in2}$$

$$d_{out2} = f_2(d_{in2})$$



$$d_{out1} = f_1(d_{in1})$$

$$d_{out2} = f_2(d_{in2})$$

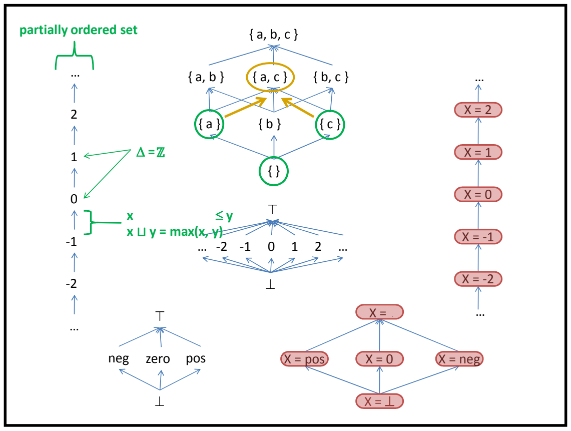
$$d_{join} = d_{out1} \sqcup d_{out2}$$

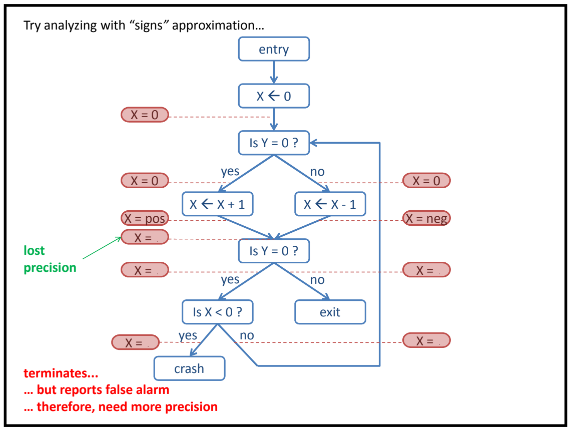
$$d_{join} = d_{in3}$$

$$d_{out3} = f_3(d_{in3})$$

least upper bound operator

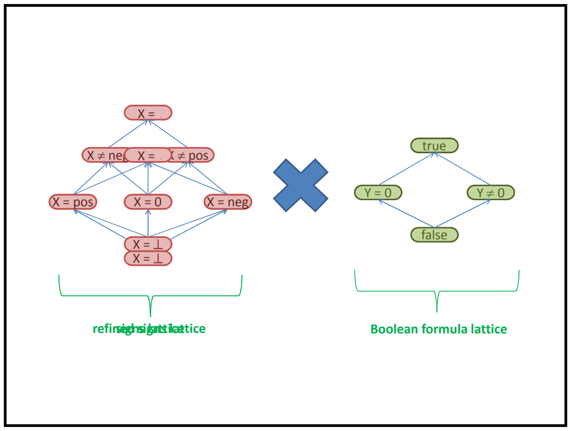
What is the space of dataflow elements, Δ?
What is the least upper bound operator, ⊔?

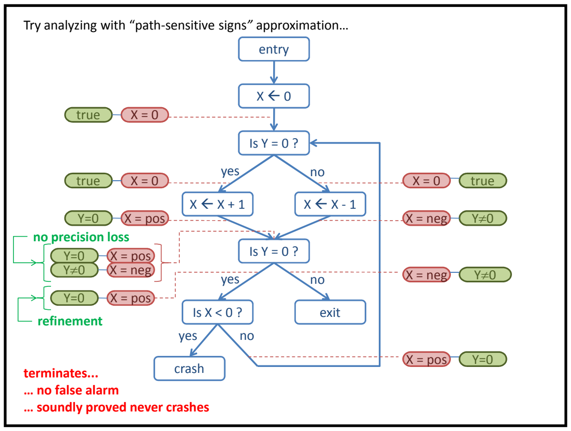




Try analyzing with "signs" approximation…

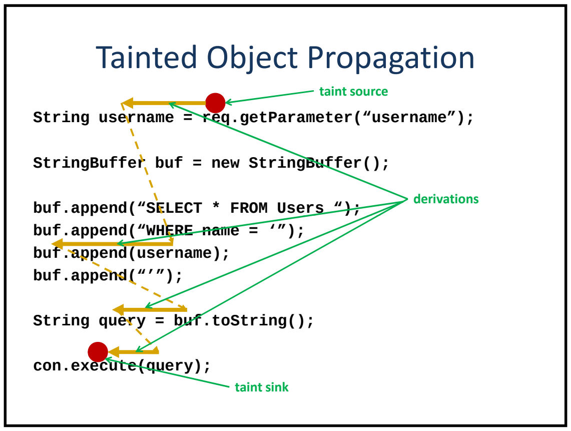
terminates...
… but reports false alarm
… therefore, need more precision





Try analyzing with "path-sensitive signs" approximation…

terminates...
… no false alarm
… soundly proved never crashes



Tainted Object Propagation

taint source

```
String username = req.getParameter("username");

StringBuffer buf = new StringBuffer();

buf.append("SELECT * FROM Users ");
buf.append("WHERE name = '");
buf.append(username);
buf.append("'");

String query = buf.toString();

con.execute(query);
```

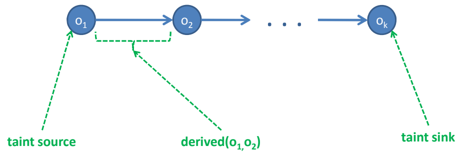
derivations

taint sink

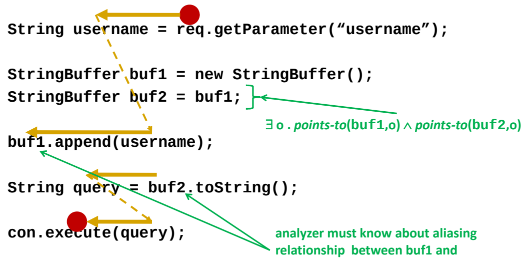
Tainted Object Propagation

Term	Descriptor	Example
Source Object	Method	HttpServletRequest.getParameter(String)
	Parameter	return
	Access path	ε

Security Violation



Complication of Aliasing



```
String username = req.getParameter(“username”);

StringBuffer buf1 = new StringBuffer();
StringBuffer buf2 = buf1;

buf1.append(username);

String query = buf2.toString();

con.execute(query);
```

∃ o . *points-to*(buf1,o) ∧ *points-to*(buf2,o)

analyzer must know about aliasing relationship between buf1 and buf2 to find vulnerability

No security violation found!

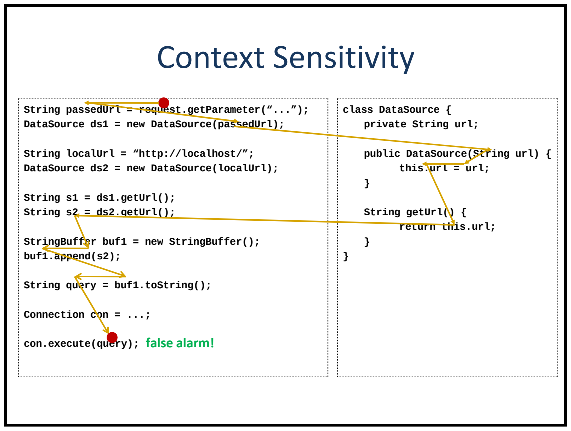
Statements

Statement	Code	Description
object creation	o_i: `T v = new T();`	Creates a new heap object o_i of type **T**, and makes variable **v** point to o_i

Pointer Analysis

Predicate	Description
points-to (**v**,o)	variable **v** can point to heap object o

Statement	sql-injection (o_{src}, o_{sink})
`1: user = req.getParam(“user”);`	
`2: buf.append(user);`	
`3: query = buf.toString();`	
`4: con.execute(query);`	

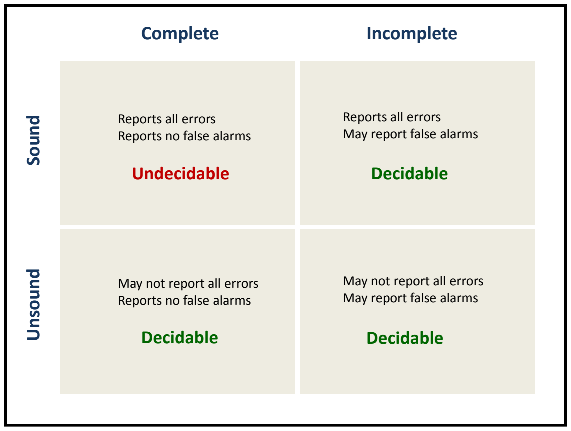


Context Sensitivity

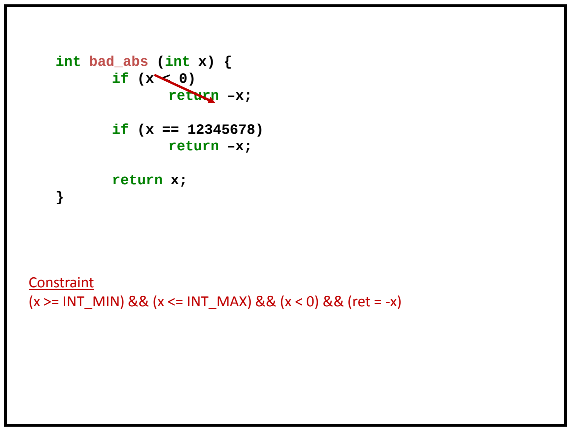
```
String passedUrl = request.getParameter("…");
DataSource ds1 = new DataSource(passedUrl);

String localUrl = "http://localhost/";
DataSource ds2 = new DataSource(localUrl);

String s1 = ds1.getUrl();
String s2 = ds2.getUrl();

StringBuffer buf1 = new StringBuffer();
buf1.append(s2);

String query = buf1.toString();

Connection con = ...;

con.execute(query);
```

false alarm!

```
class DataSource {
   private String url;

   public DataSource(String url) {
         this.url = url;
   }

   String getUrl() {
         return this.url;
   }
}
```

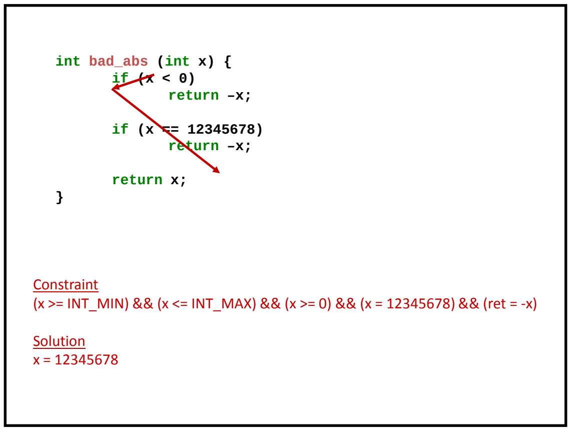


	Complete	Incomplete
Sound	Reports all errors Reports no false alarms Undecidable	Reports all errors May report false alarms Decidable
Unsound	May not report all errors Reports no false alarms Decidable	May not report all errors May report false alarms Decidable



```
int bad_abs (int x) {
      if (x < 0)
            return -x;

      if (x == 12345678)
            return -x;

      return x;
}
```

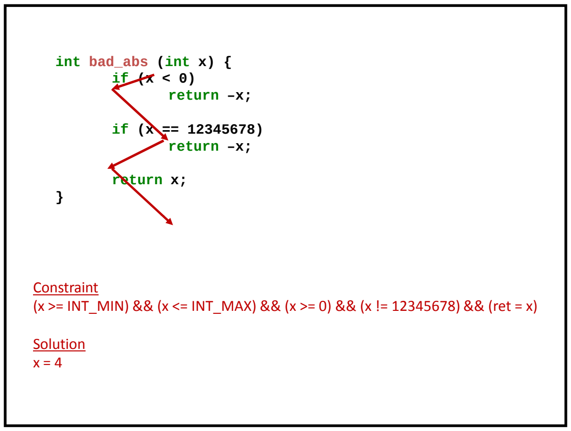
Constraint
(x >= INT_MIN) && (x <= INT_MAX) && (x < 0) && (ret = -x)



```
int bad_abs (int x) {
      if (x < 0)
            return -x;

      if (x == 12345678)
            return -x;

      return x;
}
```

Constraint
(x >= INT_MIN) && (x <= INT_MAX) && (x >= 0) && (x = 12345678) && (ret = -x)

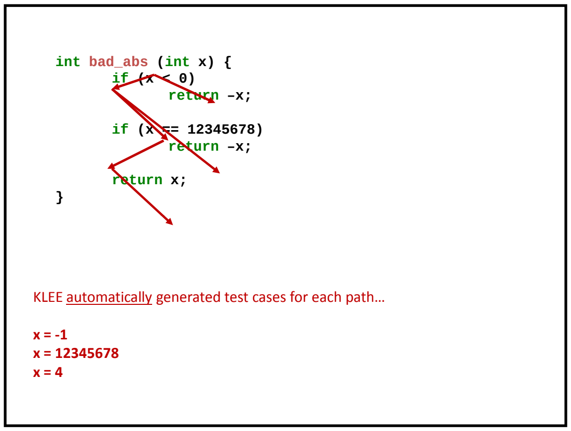
Solution
x = 12345678



```
int bad_abs (int x) {
      if (x < 0)
            return -x;

      if (x == 12345678)
            return -x;

      return x;
}
```

Constraint
(x >= INT_MIN) && (x <= INT_MAX) && (x >= 0) && (x != 12345678) && (ret = x)

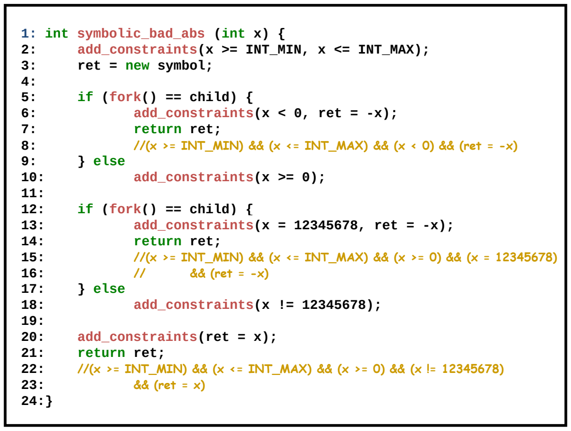
Solution
x = 4



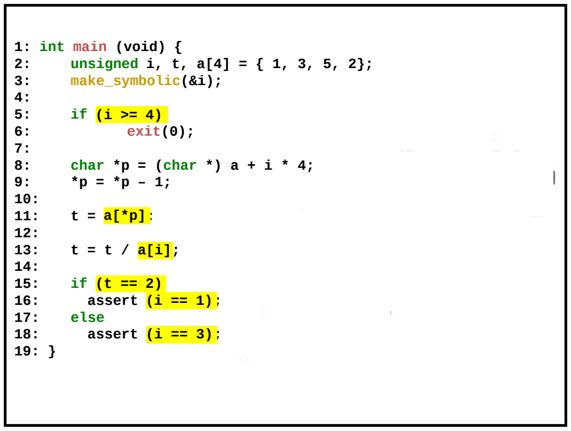
```
int bad_abs (int x) {
      if (x < 0)
            return -x;

      if (x == 12345678)
            return -x;

      return x;
}
```

KLEE automatically generated test cases for each path…

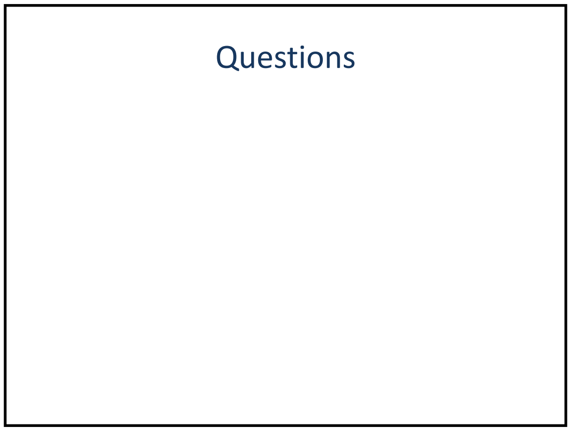
x = -1
x = 12345678
x = 4



```
int symbolic_bad_abs (int x) {
    add_constraints(x >= INT_MIN, x <= INT_MAX);
    ret = new symbol;

    if (fork() == child) {
        add_constraints(x < 0, ret = -x);
        return ret;
        //(x >= INT_MIN) && (x <= INT_MAX) && (x < 0) && (ret = -x)
    } else
        add_constraints(x >= 0);

    if (fork() == child) {
        add_constraints(x = 12345678, ret = -x);
        return ret;
        //(x >= INT_MIN) && (x <= INT_MAX) && (x >= 0) && (x = 12345678)
        //      && (ret = -x)
    } else
        add_constraints(x != 12345678);

    add_constraints(ret = x);
    return ret;
    //(x >= INT_MIN) && (x <= INT_MAX) && (x >= 0) && (x != 12345678)
        && (ret = x)
}
```



```
int main (void) {
    unsigned i, t, a[4] = { 1, 3, 5, 2};
    make_symbolic(&i);

    if (i >= 4)
        exit(0);

    char *p = (char *) a + i * 4;
    *p = *p – 1;

    t = a[*p];

    t = t / a[i];

    if (t == 2)
      assert (i == 1);
    else
      assert (i == 3);
}
```



Questions