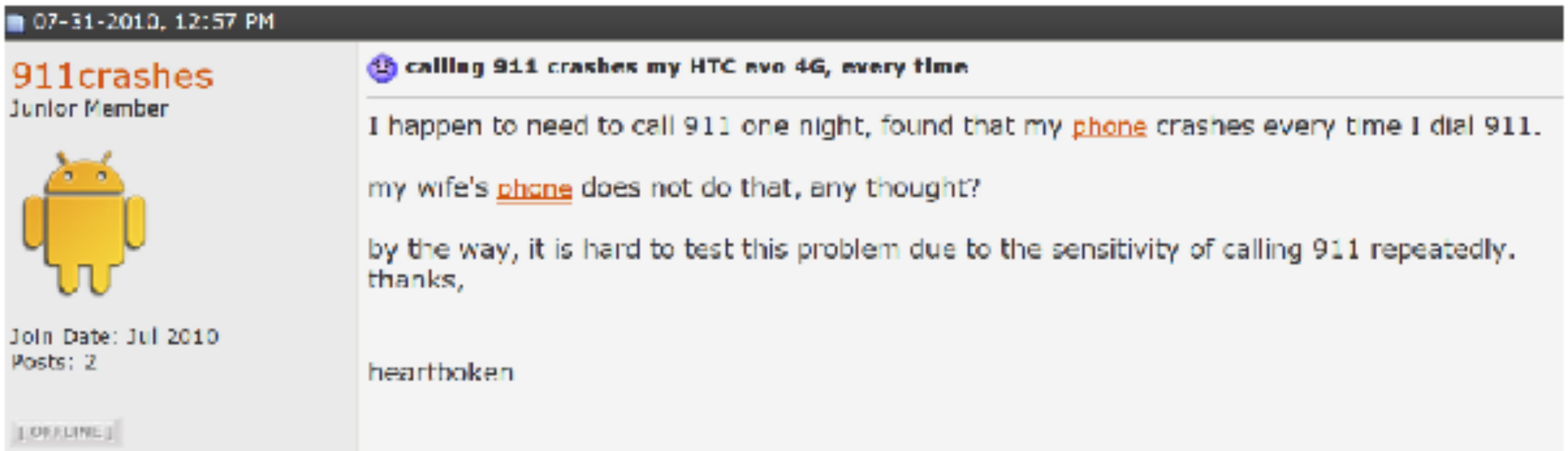
# Program Analysis for Security

John Mitchell

*Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh at Stanford University. When slides are obtained from other sources, a  a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.*

# MOTIVATION FOR PROGRAM ANALYZERS

# Software bugs are serious problems




Thanks: Isil and Thomas Dillig

# Facebook missed a single security check…

# App stores

## Apps for whatever you're up for.

Stay on top of the news. Stay on top of your finances. Or plan your dream vacation. No matter what you want to do with your iPhone, there's probably an app to help you do it.

### Business

iPhone is ready for work. Manage projects, track stocks, monitor finances, and more with these 9-to-5 apps.

View business apps in the App Store >

### Education

Keep up with your studies using intelligent education apps like King of Math and NatureTap.

View education apps in the App Store >

### Entertainment

Kick back and enjoy the show. Or find countless other ways to entertain yourself. These apps offer hours of viewing pleasure.

View entertainment apps in the App Store >

### Family & Kids

Turn every night into family night with interactive apps that are fun for the whole house.

View family and kids apps in the App Store >

### Finance

Create budgets, pay bills, and more with financial apps that take everything into account.

View finance apps in the App Store >

### Food & Drink

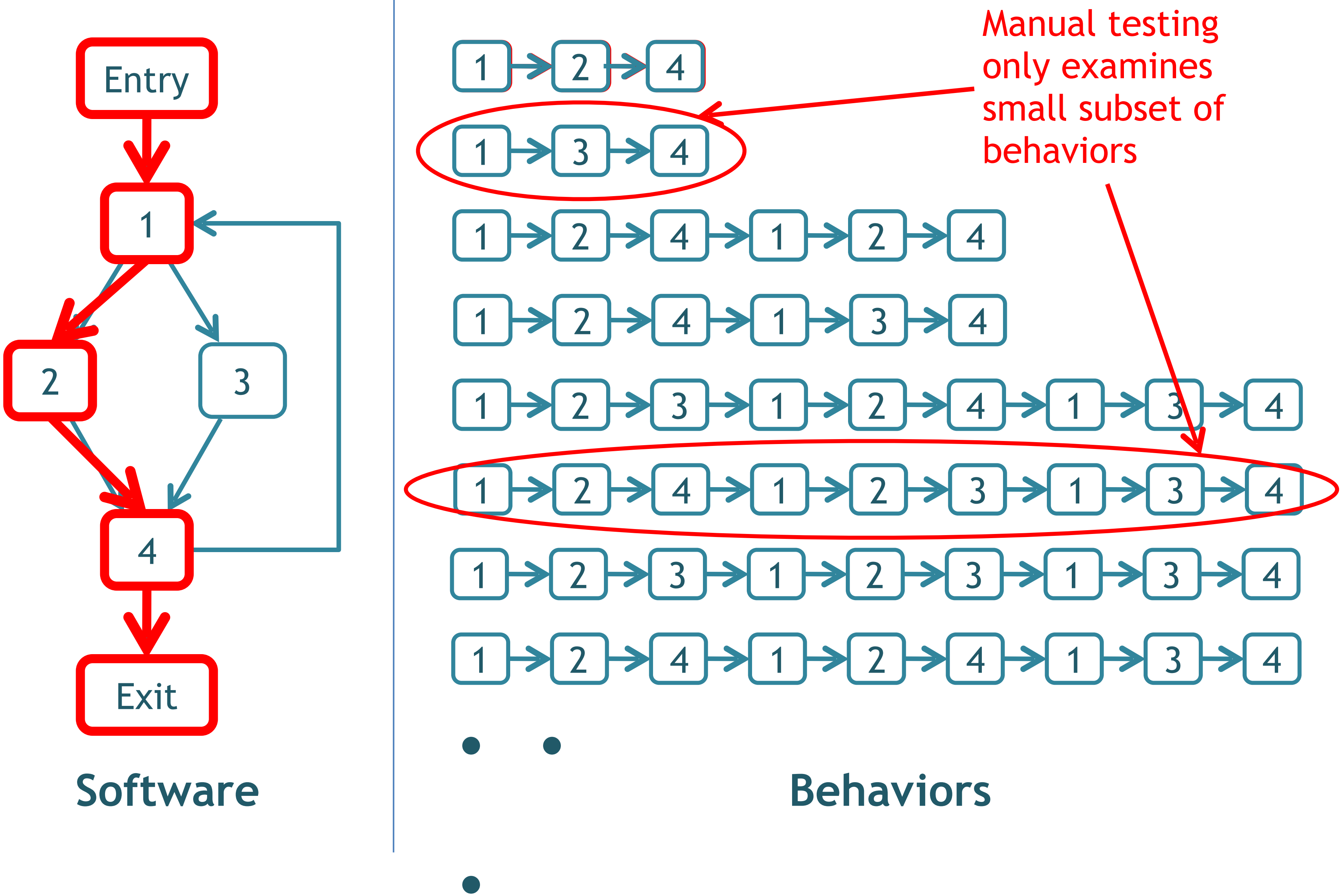
Hungry? Thirsty? A little of both? Learn new recipes, drinks, and the secrets behind what makes a great meal.

View food and drink apps in the App Store >

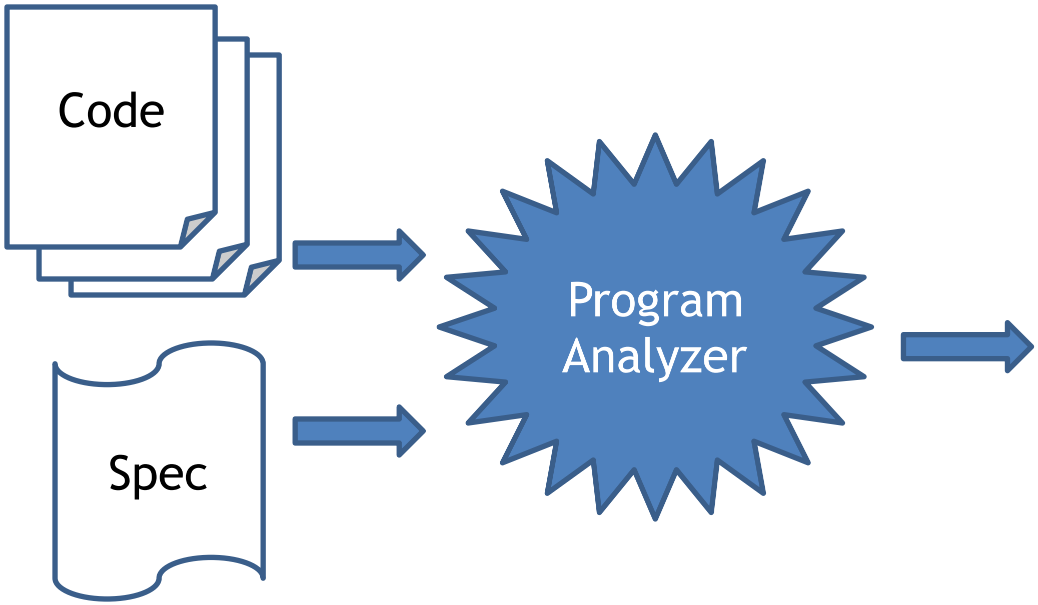
How can you tell whether software you

- Develop
- Buy

is safe to install and run?

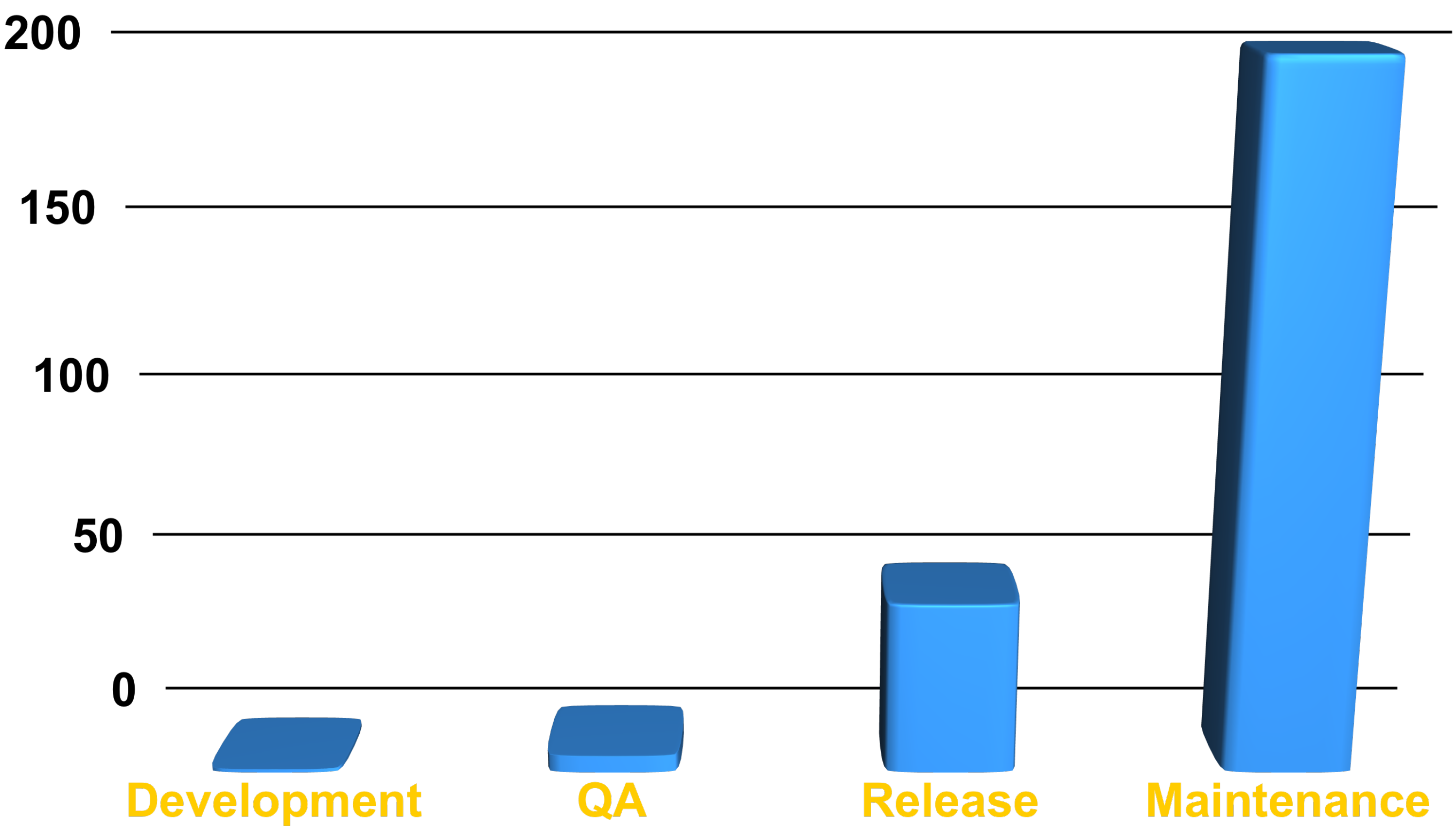

Entry
1
2
3
4
Exit
Software
Manual testing only examines small subset of behaviors
Behaviors

# 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 |

# Cost of Fixing a Defect




Credit: Andy Chou, Coverity

# Cost of security or data privacy vulnerability?

# Two options

- Static analysis
  - Inspect code or run automated method to find errors or gain confidence about their absence
- Dynamic analysis
  - Run code, possibly under instrumented conditions, to see if there are likely problems

# Static vs Dynamic Analysis

- Static
  - Consider all possible inputs (in summary form)
  - Find bugs and vulnerabilities
  - Can prove absence of bugs, in some cases
- Dynamic
  - Need to choose sample test input
  - Can find bugs/vulnerabilities
  - Cannot prove their absence

# Static Analysis

- Long research history
- Decade of commercial products
  - FindBugs, Fortify, Coverity, MS tools, …
- Main topic for this lecture

# Dynamic analysis

- Instrument code for testing
  - Heap memory: Purify
  - Perl tainting (information flow)
  - Java race condition checking
- Black-box testing
  - Fuzzing and penetration testing
  - Black-box web application security analysis
- Will come back to later in course

# Summary

- Program analyzers
  - Find problems in code before it is shipped to customers or before you install and run it
- Static analysis
  - Analyze code to determine behavior on all inputs
- Dynamic analysis
  - Choose some sample inputs and run code to see what happens

# STATIC ANALYSIS

# Static Analysis: Outline

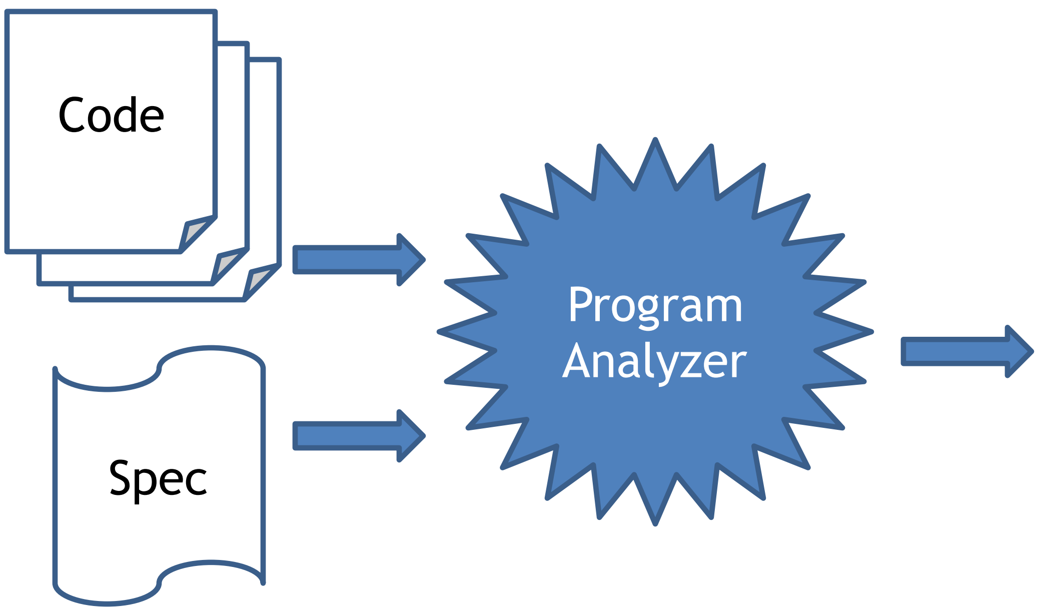
- General discussion of static analysis tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
  - Sample security checkers results
- Static analysis for of Android apps

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# Static analysis goals

- Bug finding
  - Identify code that the programmer wishes to modify or improve
- Correctness
  - Verify the absence of certain classes of errors

# Sound Program Analyzer




Analyze large code bases

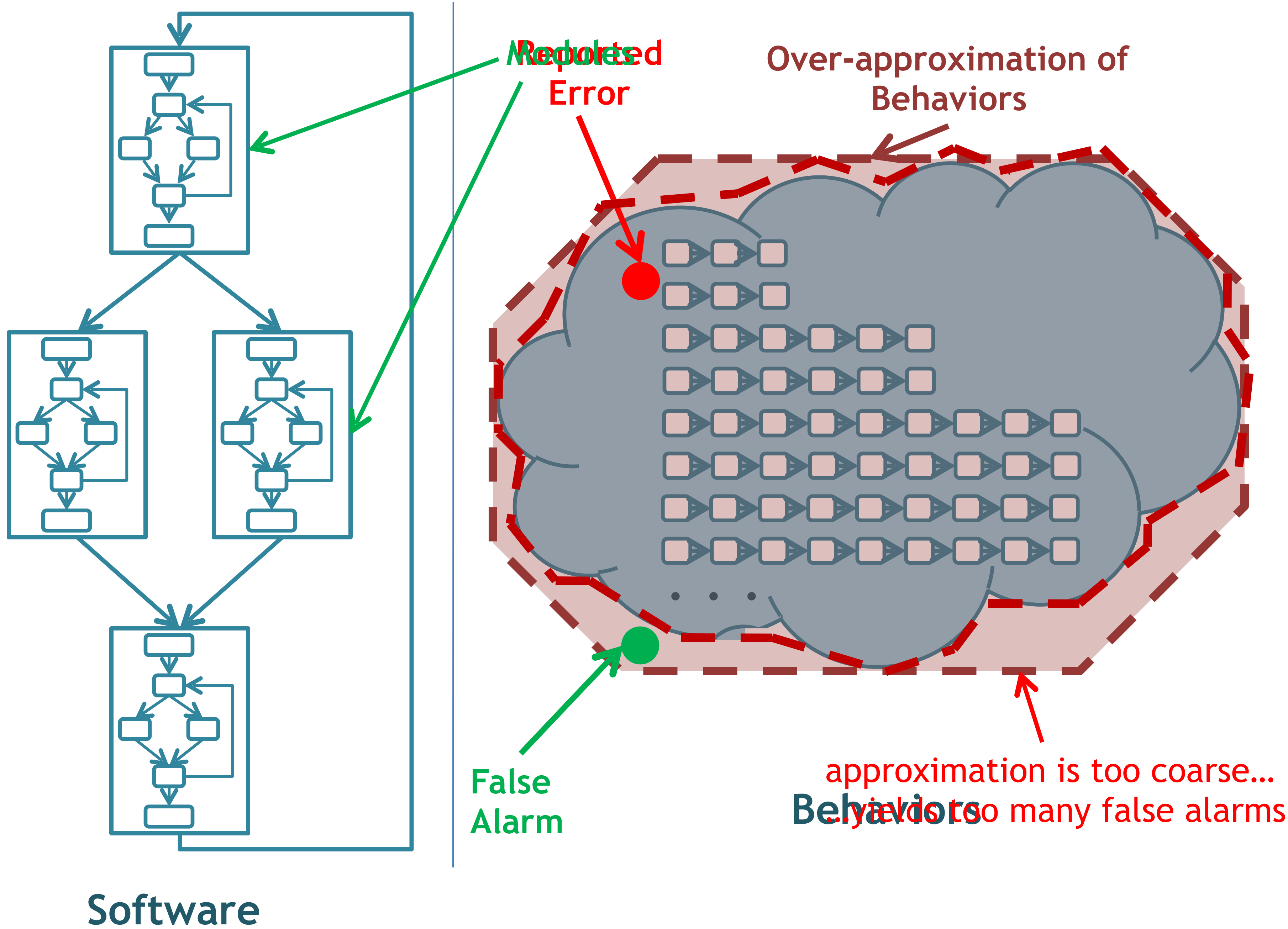
| Report | Type | Line |
|---|---|---|
| 1 | mem leak | 324 |
| 2 | buffer oflow | 4,353,245 |
| 3 | sql injection | 23,212 |
| 4 | stack oflow | 86,923 |
| 5 | dang ptr | 8,491 |
| … | … | … |
| 10,502 | info leak | 10,921 |

false alarm

false alarm

Sound: may report many warnings

May emit false alarms



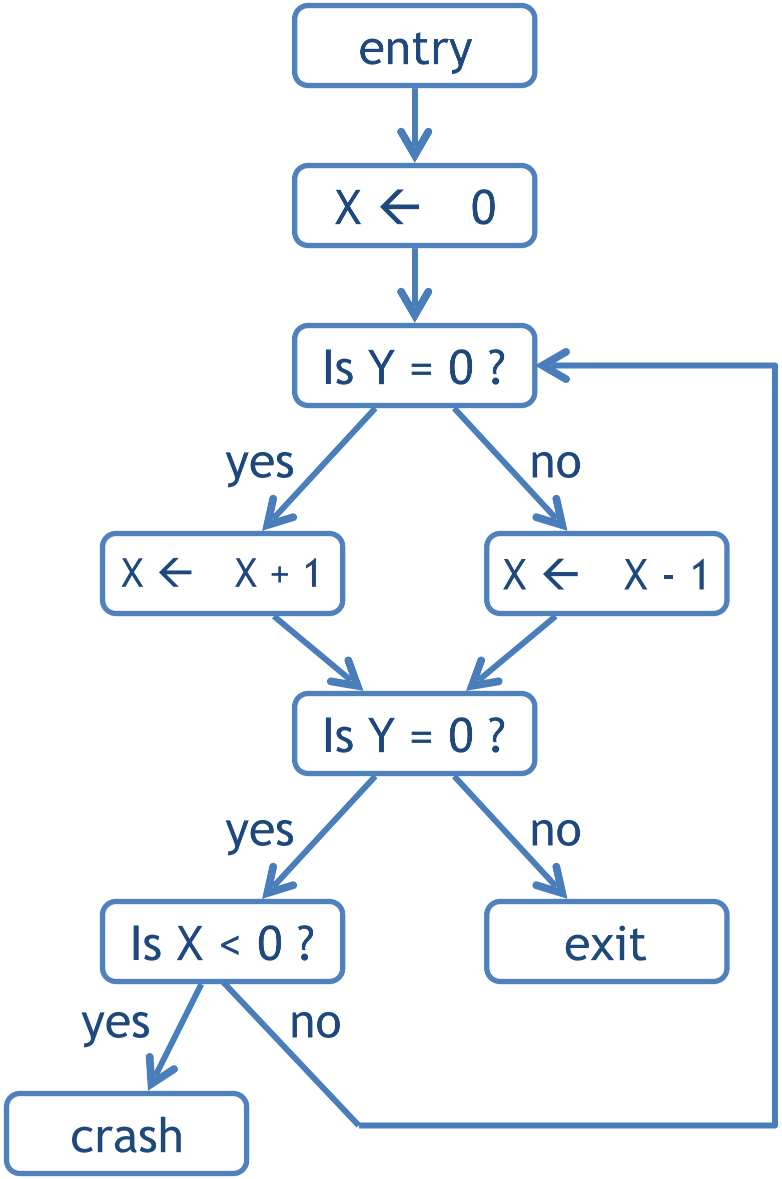
Modules
Reported
Error
Over-approximation of
Behaviors
False
Alarm
approximation is too coarse…
…yields too many false alarms
Behaviors
Software

# Outline

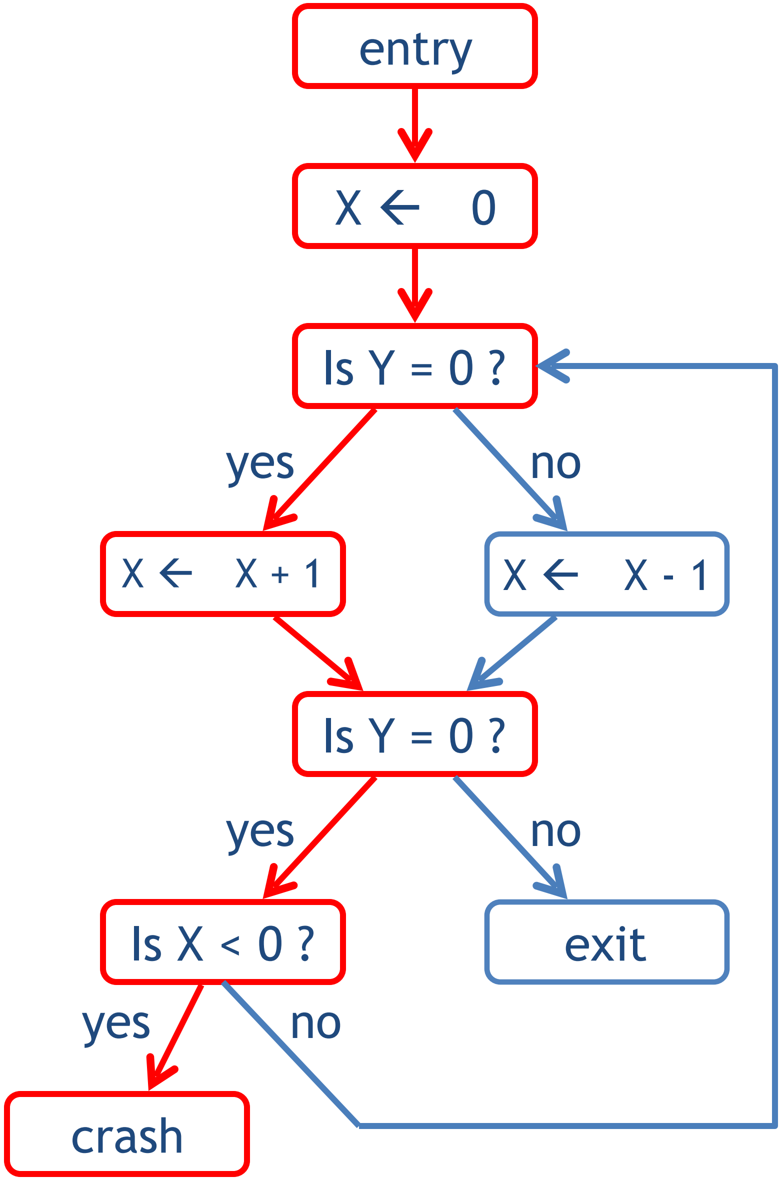
- General discussion of tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
  - Sample security-related results
- Static analysis for Android malware
  - …

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

Does this program ever crash?



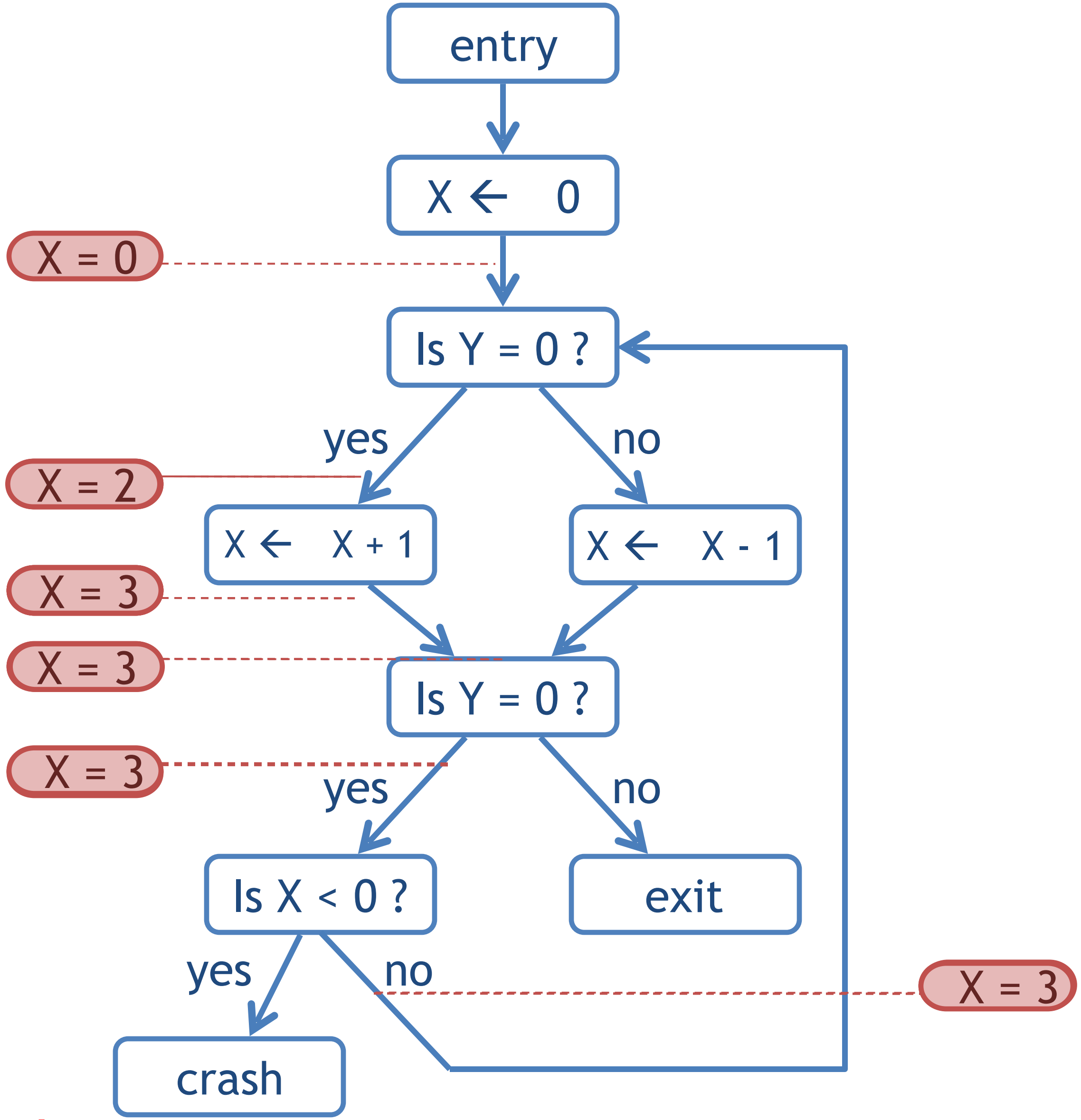
Does this program ever crash?



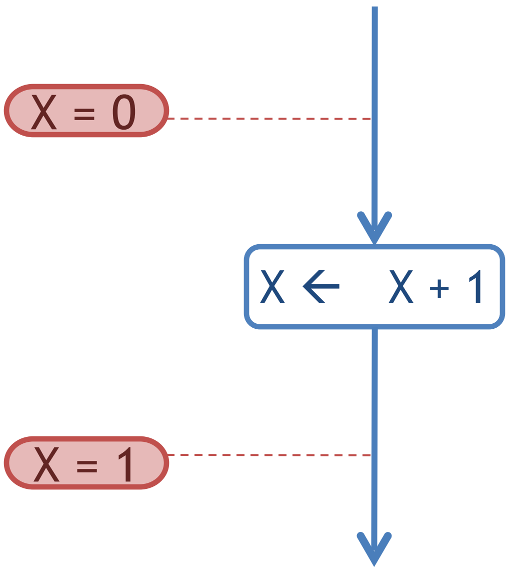

**infeasible path!**

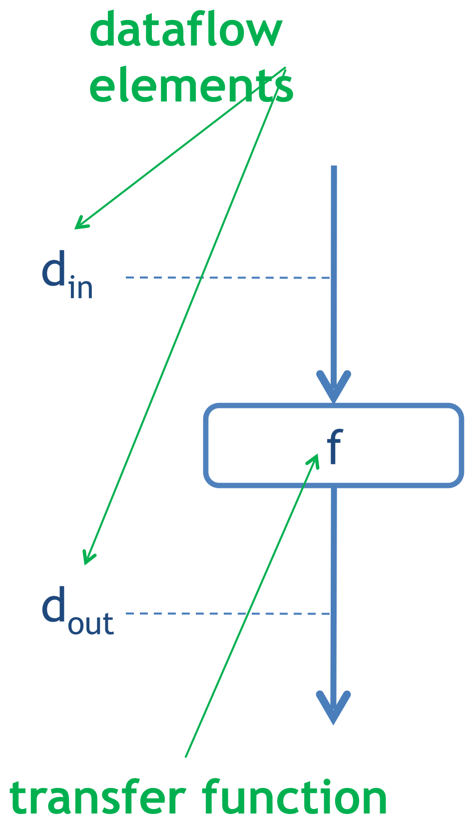
**… program will never crash**

Try analyzing without approximating…



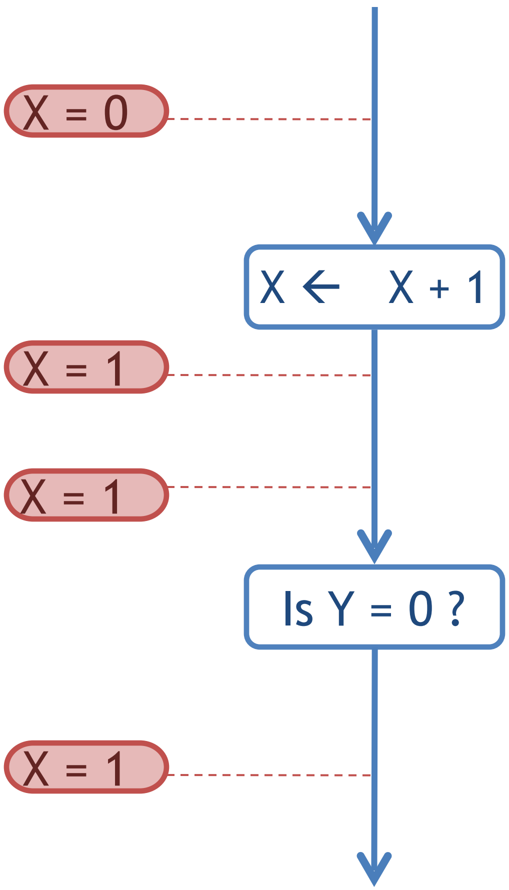

**non-termination!**
**… therefore, need to approximate**

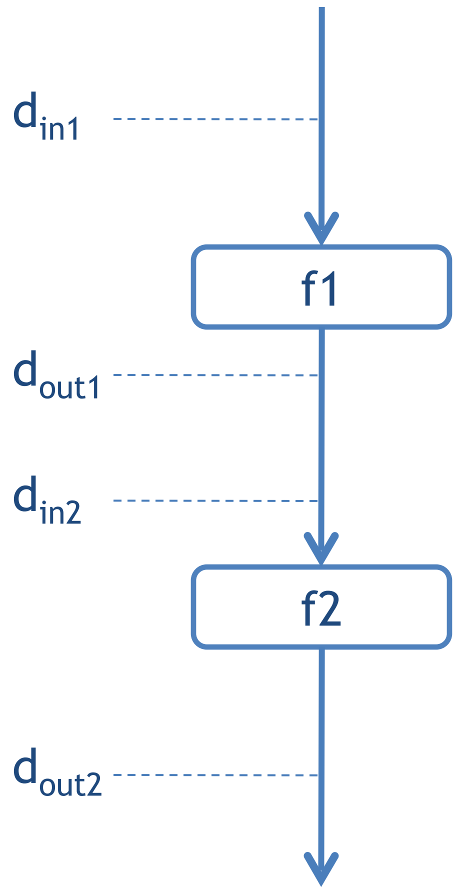



$$d_{out} = f(d_{in})$$

dataflow equation



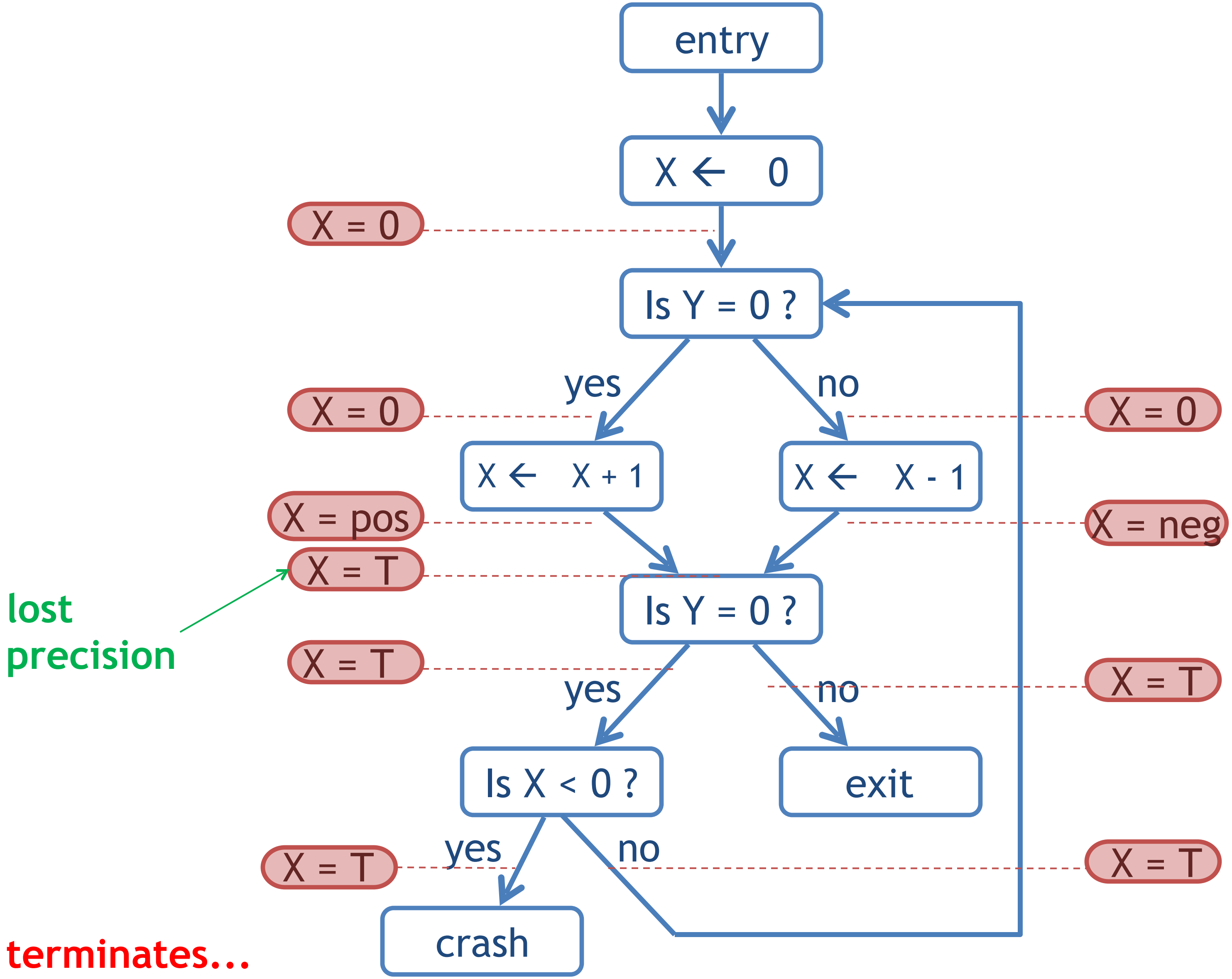




$$d_{out1} = f_1(d_{in1})$$

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

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

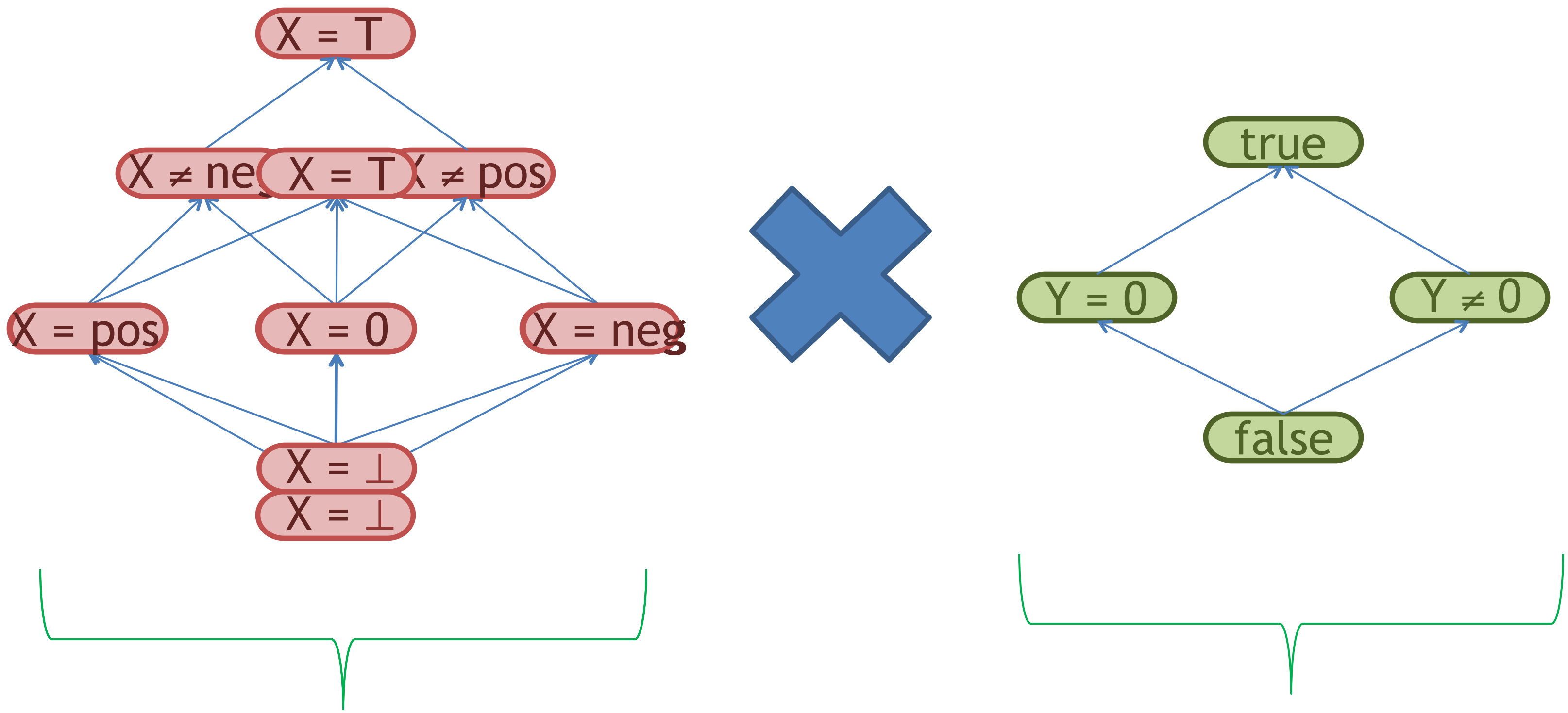
Try analyzing with "signs" approximation...




terminates...

... but reports false alarm

... therefore, need more precision

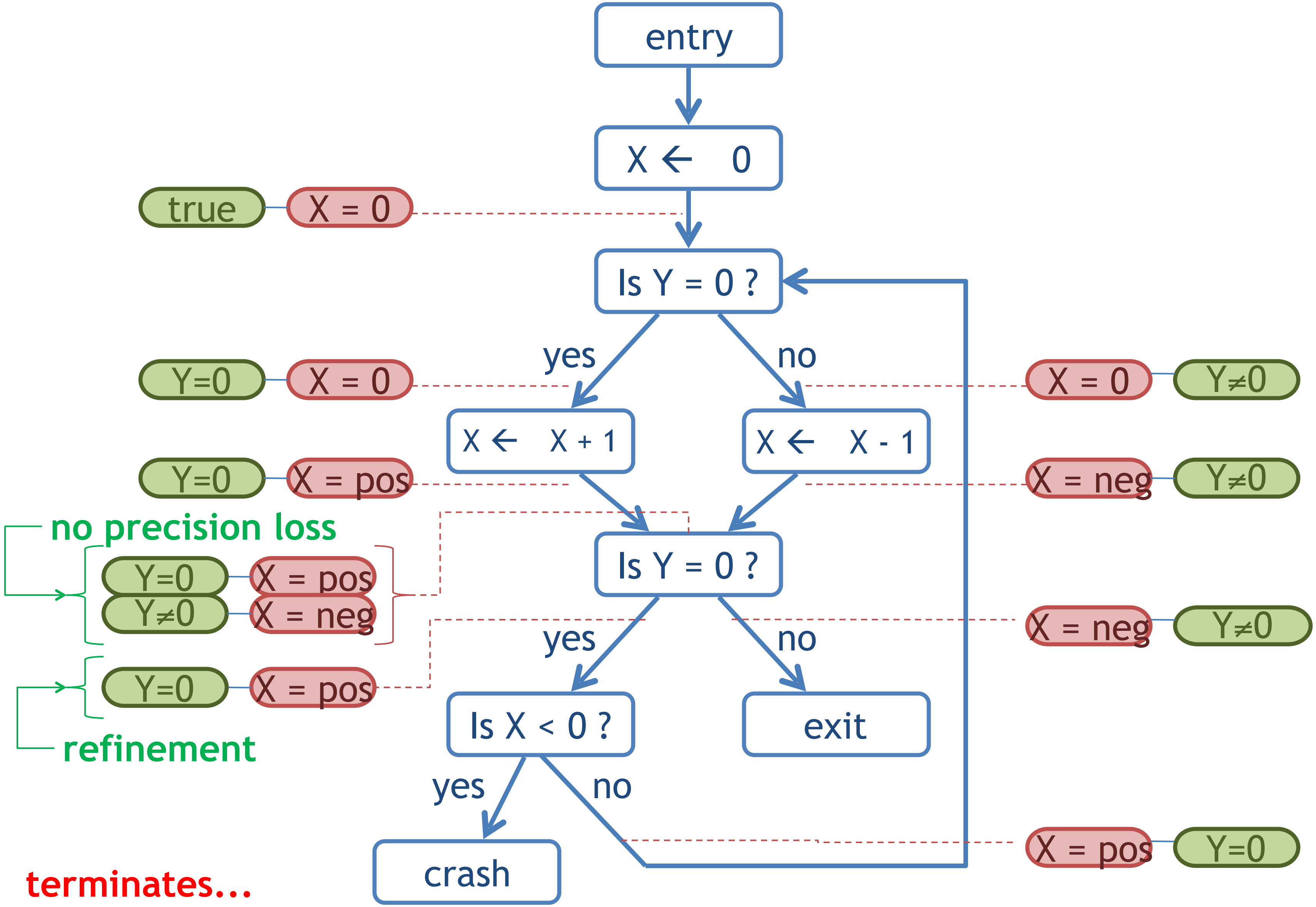



refined signs lattice

signs lattice

Boolean formula lattice

Try analyzing with "path-sensitive signs" approximation…




terminates…

… no false alarm

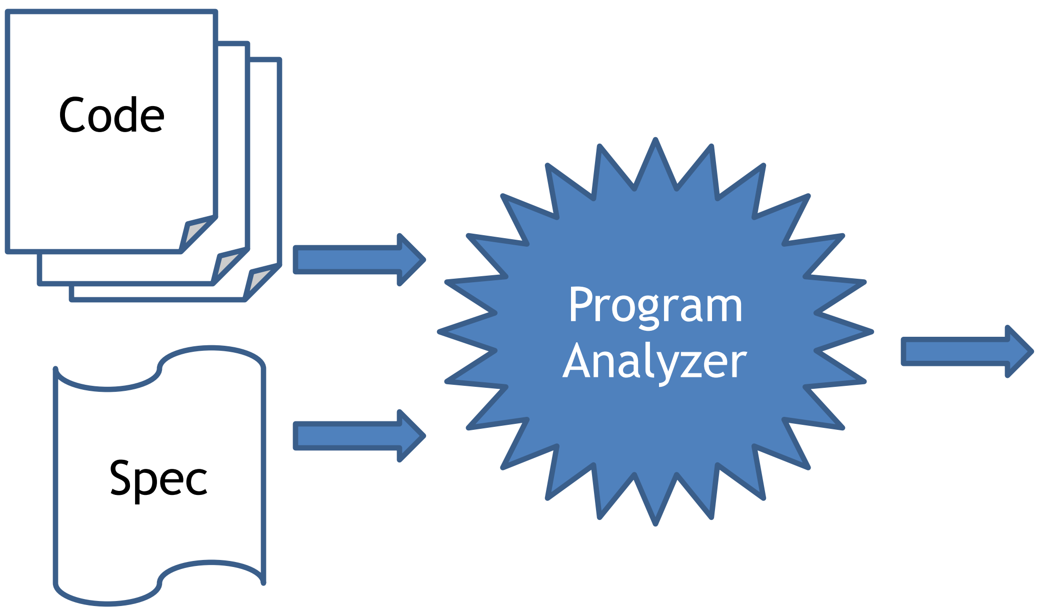
… soundly proved never crashes

# Outline

- General discussion of tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
  - Sample security-related results
- Static analysis for Android malware
  - …

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# Unsound Program Analyzer




analyze large
code bases

| 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 |
| … | … | … |

false alarm

false alarm

Not sound: may
miss some bugs

may emit
false alarms

# Demo

- Coverity video: http://youtu.be/_Vt4niZfNeA
- Observations
  - Code analysis integrated into development workflow
  - Program context important: analysis involves sequence of function calls, surrounding statements
  - This is a sales video: no discussion of false alarms

# Outline

- General discussion of tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
  - Sample security-related results
- Static analysis for Android malware
  - …

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# Bugs to Detect

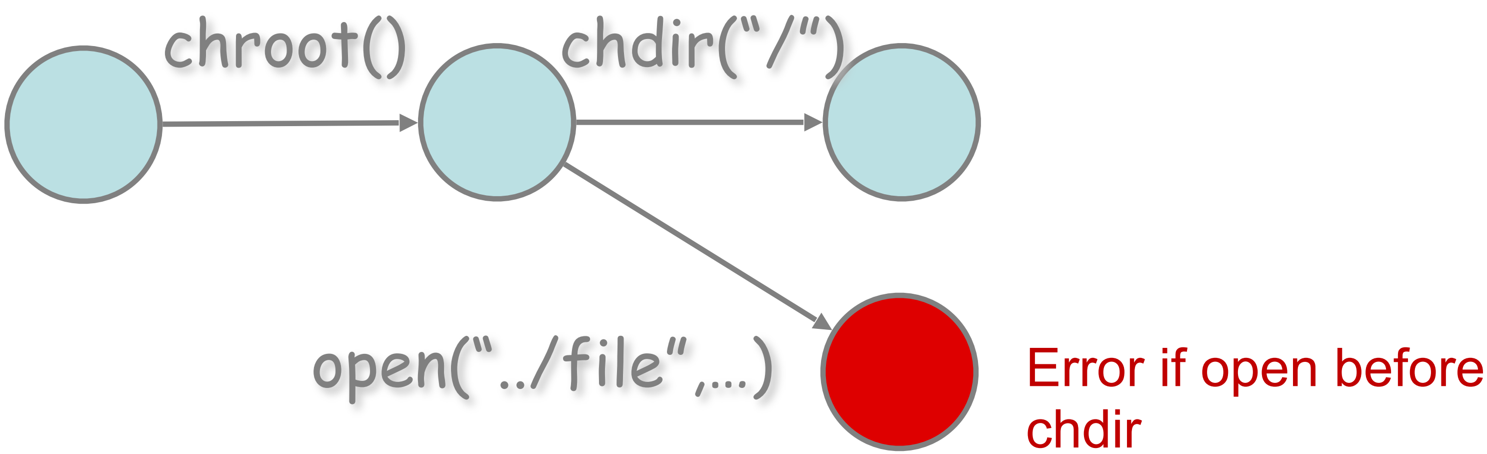
## Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

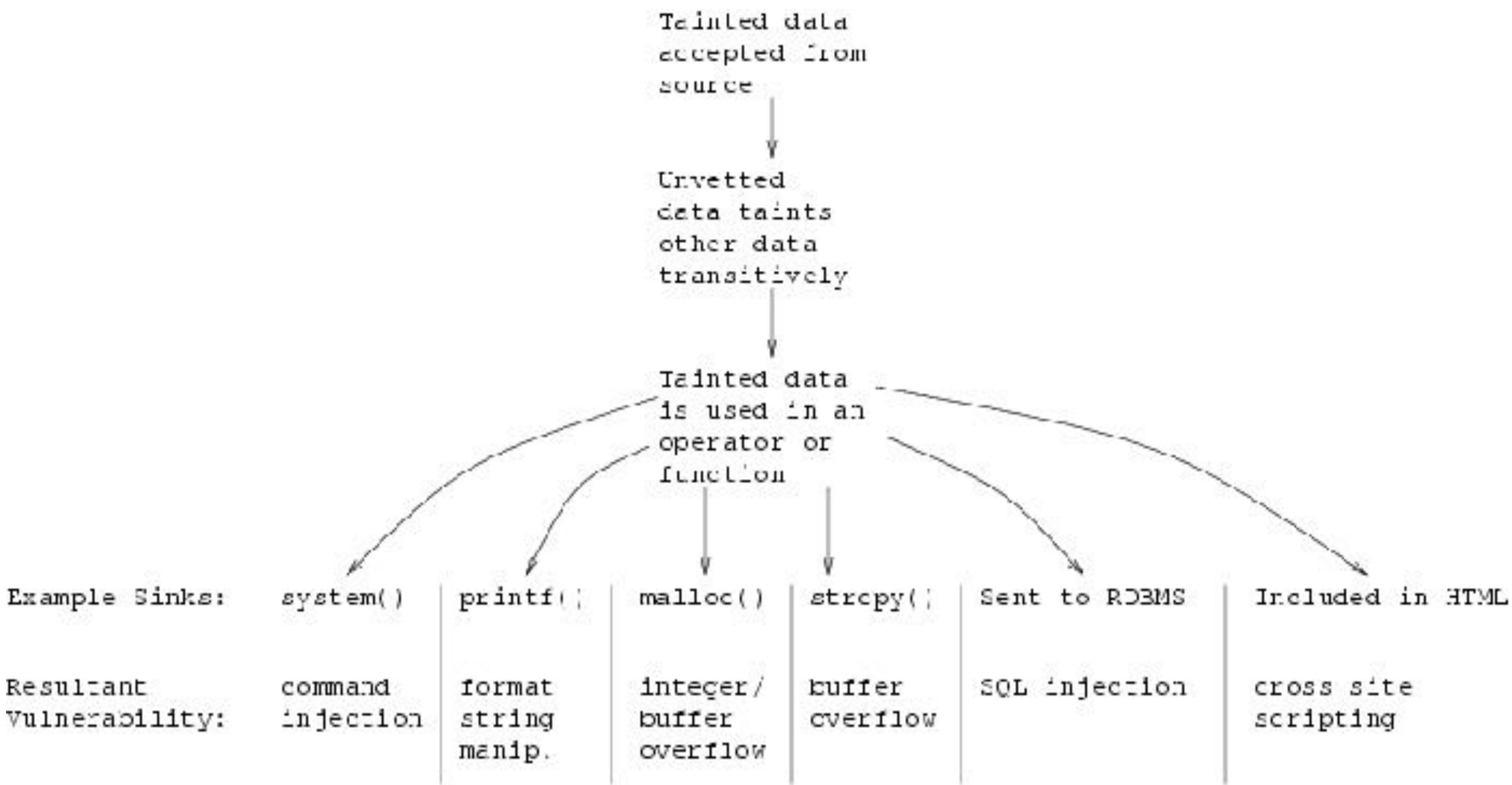
Slide credit: Andy Chou

# Example: Chroot protocol checker

- **Goal: confine process to a "jail" on the filesystem**
  - chroot() changes filesystem root for a process
- **Problem**
  - chroot() itself does not change current working directory



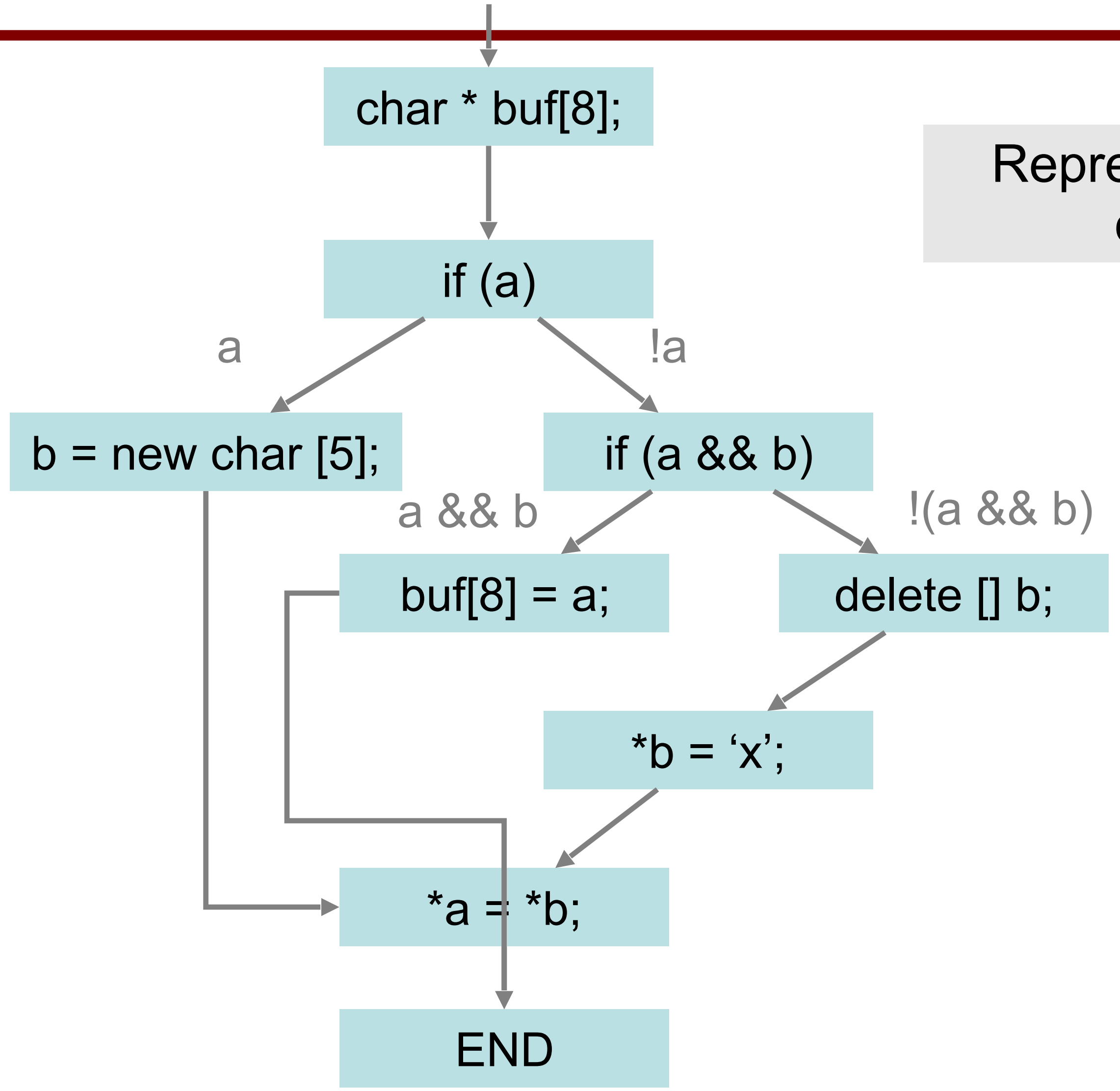
# Tainting checkers



# Finding Local Bugs

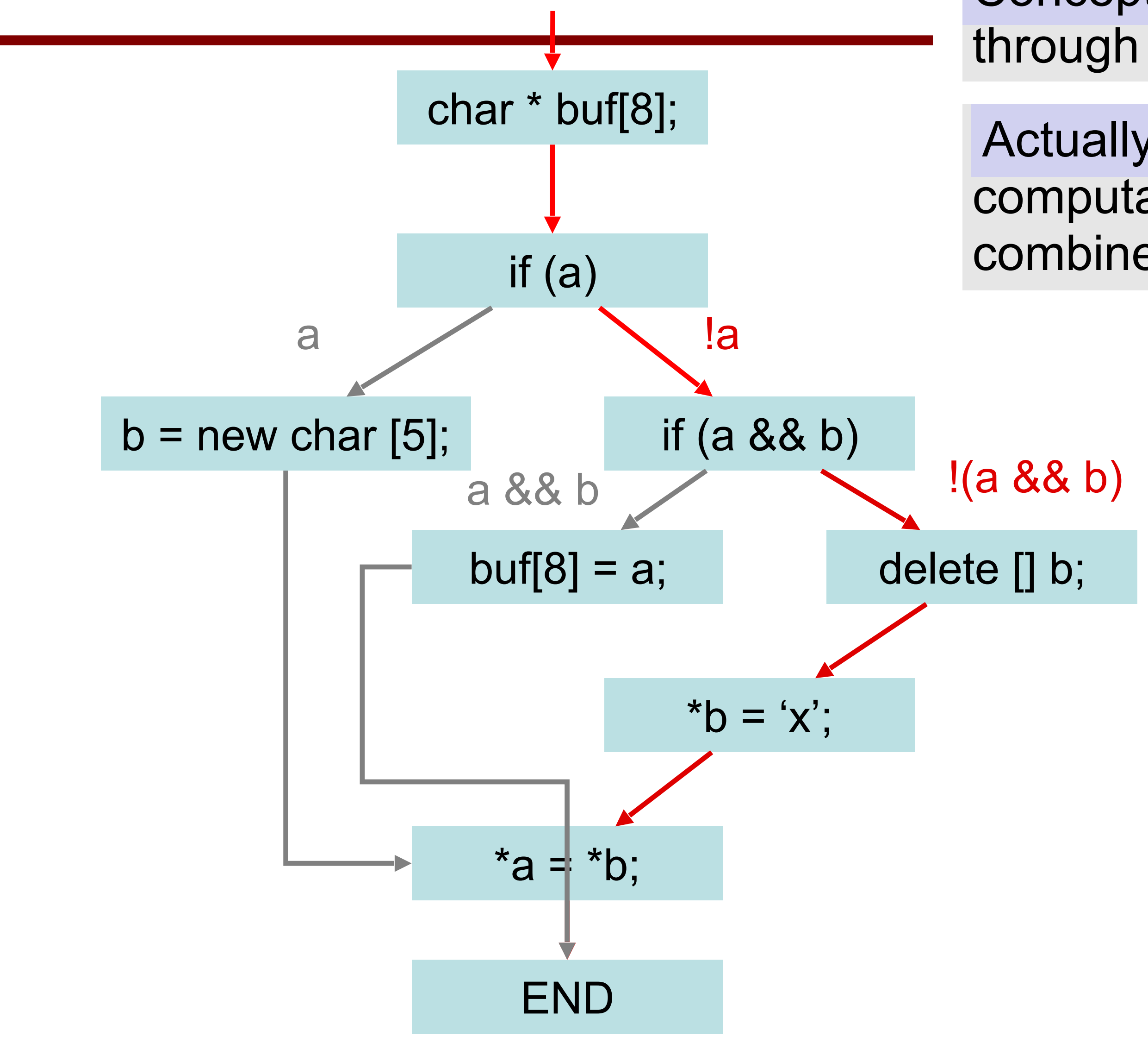
```
#define SIZE 8
void set_a_b(char * a, char * b) {
 char * buf[SIZE];
 if (a) {
     b = new char[5];
 } else {
     if (a && b) {
      buf[SIZE] = a;
      return;
     } else {
      delete [] b;
     }
     *b = 'x';
 }
 *a = *b;
}
```

# Control Flow Graph




Represent logical structure of code in graph form

# Path Traversal

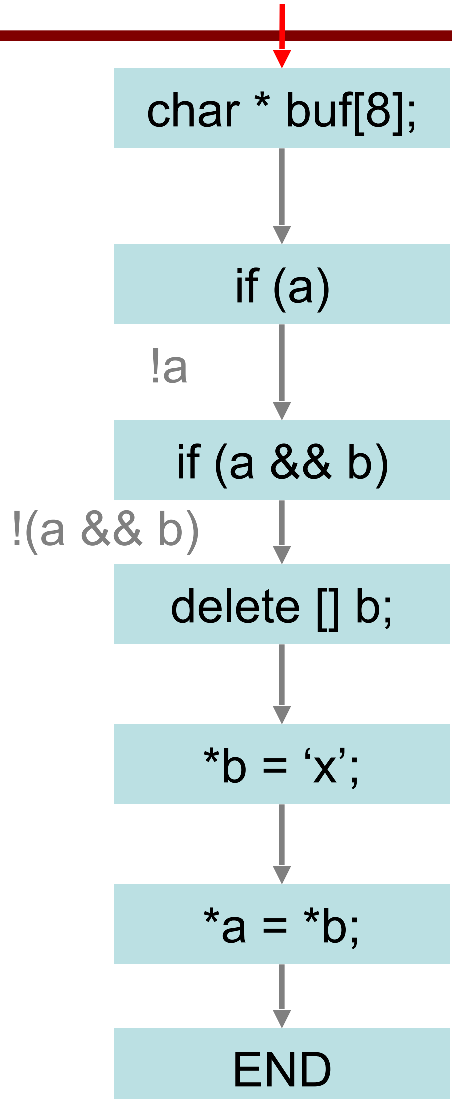



Conceptually Analyze each path through control graph separately

Actually Perform some checking computation once per node; combine paths at merge nodes

# Apply Checking

Null pointers  Use after free  Array overrun




See how three checkers are run for this path

**Checker**

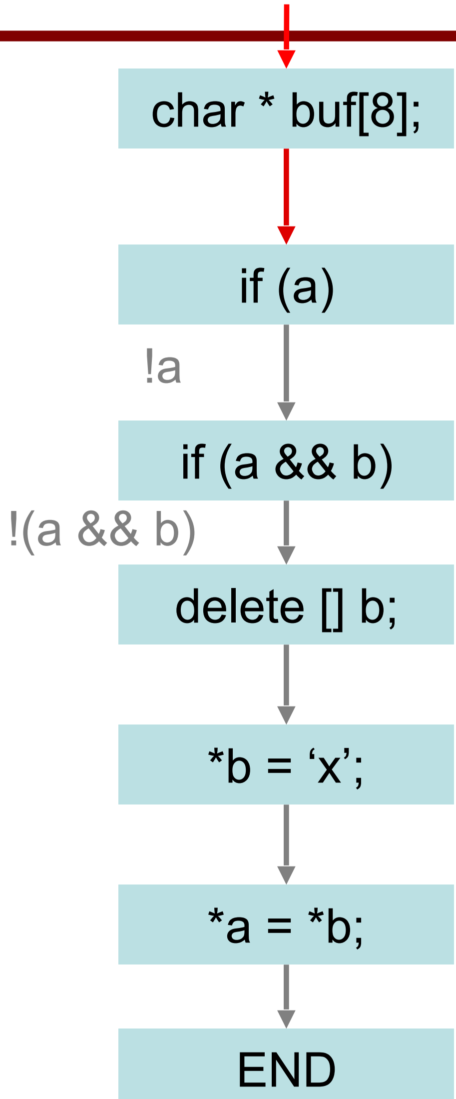
- Defined by a state diagram, with state transitions and error states

**Run Checker**

- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached

# Apply Checking

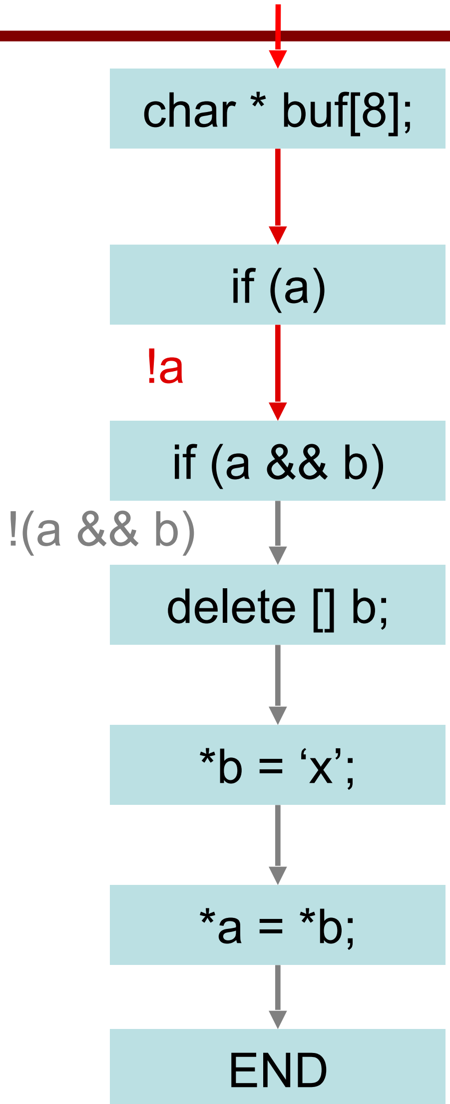
Null pointers  Use after free  Array overrun




“buf is 8 bytes”

# Apply Checking

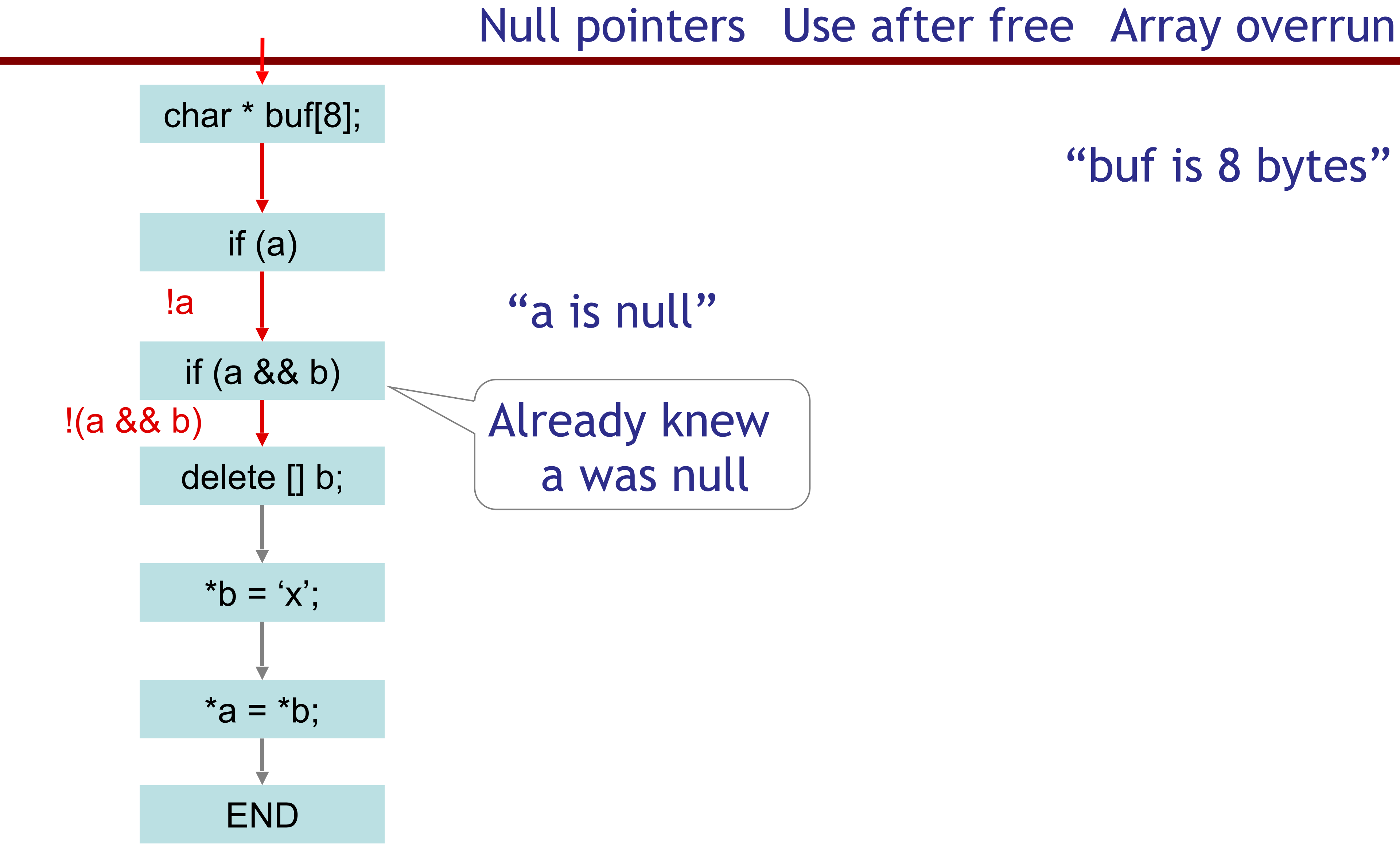
Null pointers   Use after free   Array overrun




“buf is 8 bytes”

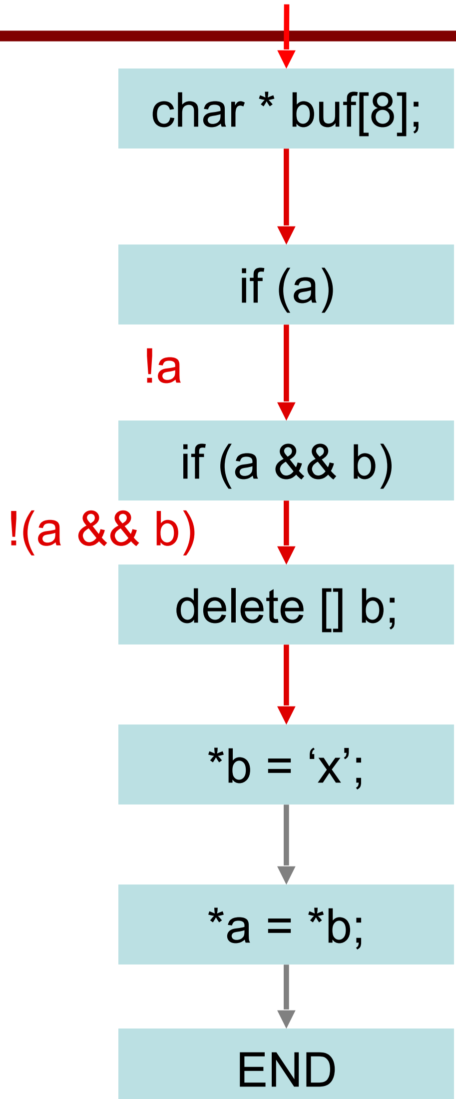
“a is null”

# Apply Checking



# Apply Checking

Null pointers | Use after free | Array overrun



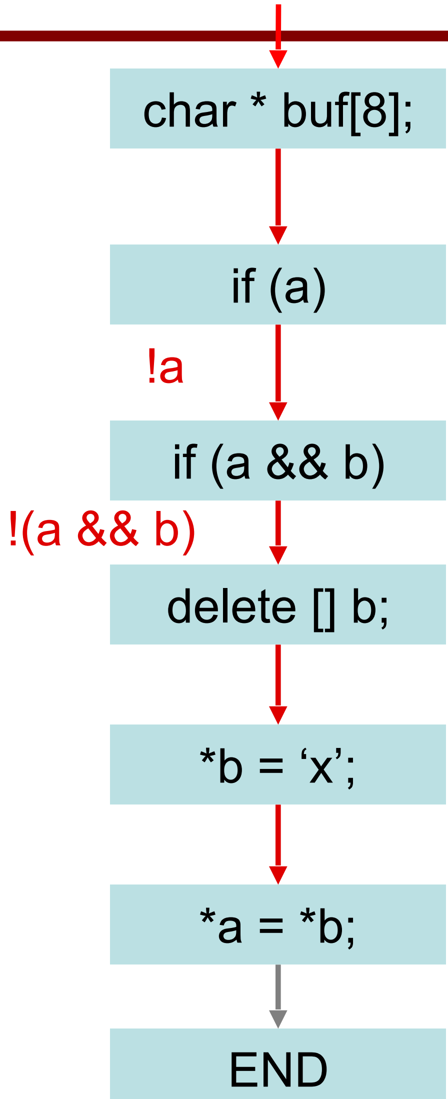

“buf is 8 bytes”

“a is null”

“b is deleted”

# Apply Checking

Null pointers Use after free Array overrun




“buf is 8 bytes”

“a is null”

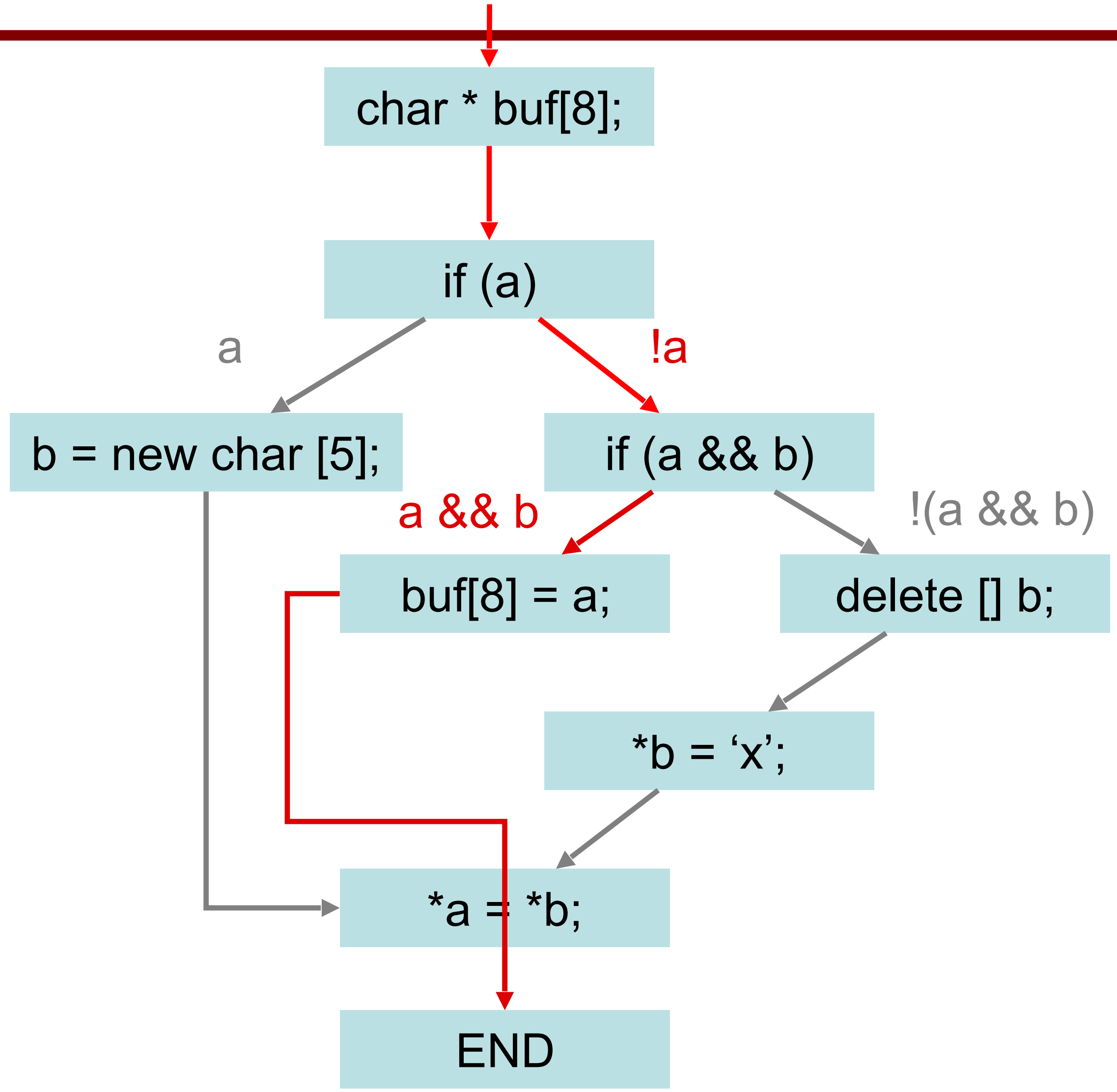
“b is deleted”

**“b dereferenced!”**

# False Positives

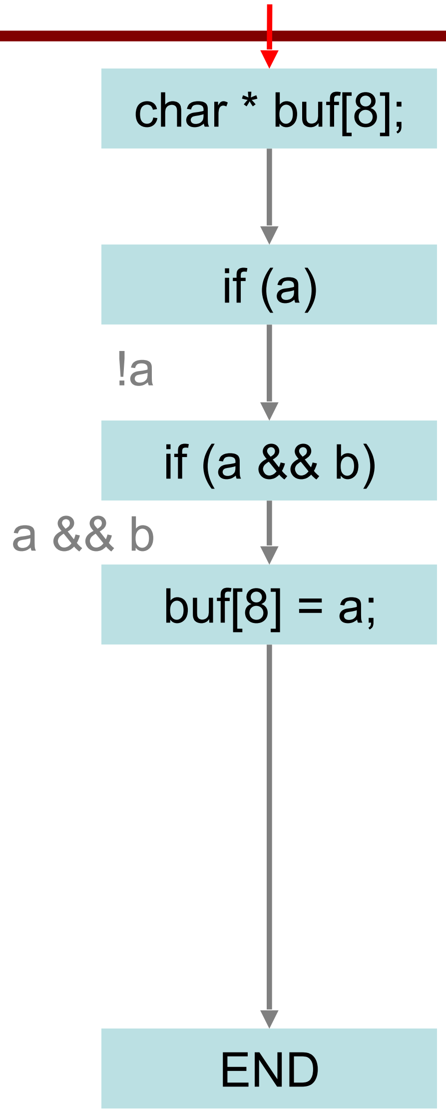
- **What is a bug? Something the user will fix.**
- **Many sources of false positives**
  - False paths
  - Idioms
  - Execution environment assumptions
  - Killpaths
  - Conditional compilation
  - “third party code”
  - Analysis imprecision
  - …

# A False Path



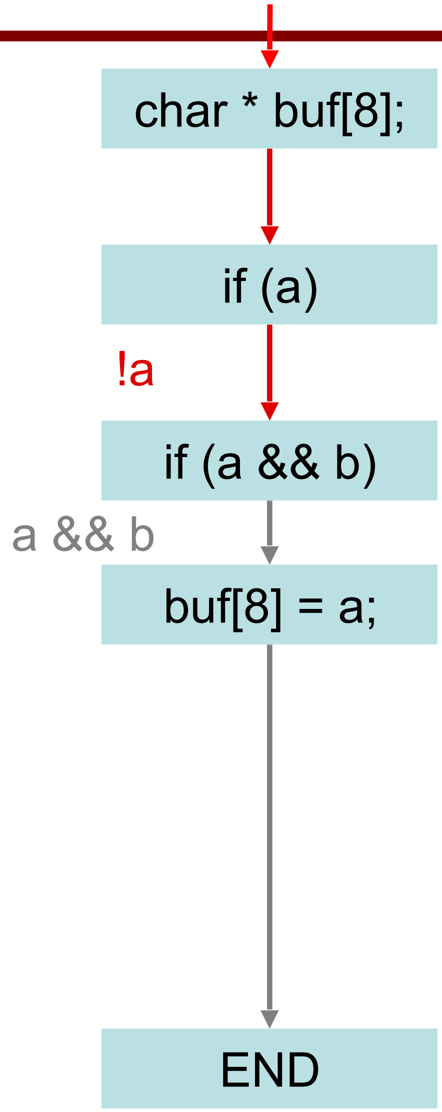
# False Path Pruning

Integer Range Disequality Branch



# False Path Pruning

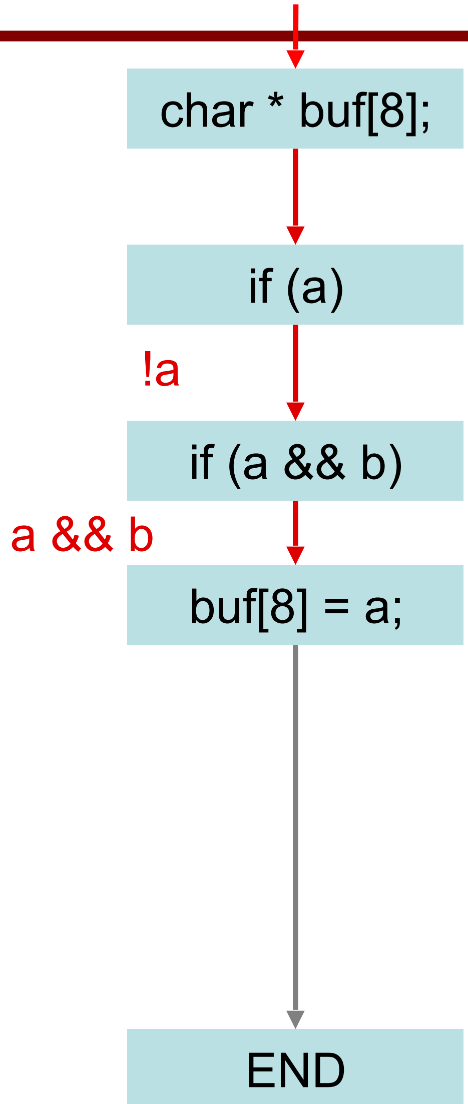
Integer Range | Disequality | Branch




"a in [0,0]"

"a == 0 is true"

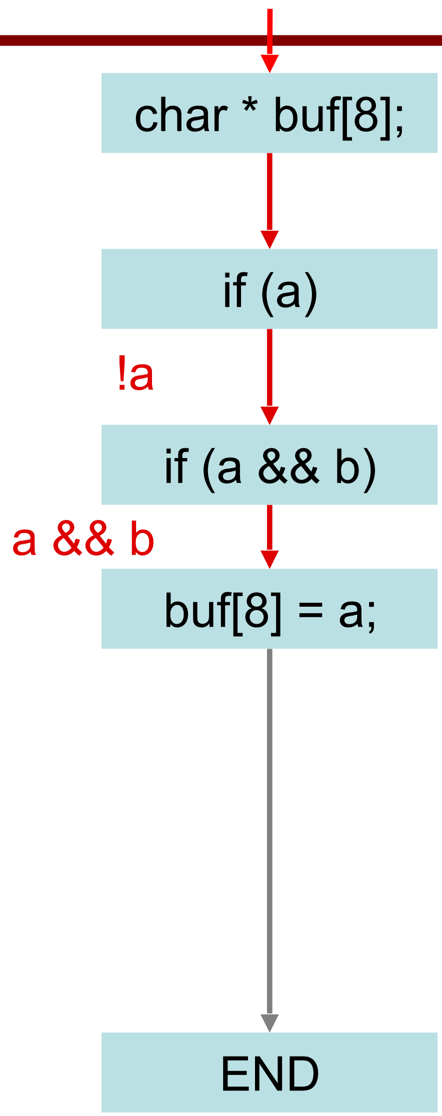
# False Path Pruning




| Integer Range | Disequality | Branch |
| --- | --- | --- |
| “a in [0,0]” | | “a == 0 is true” |
| | “a != 0” | |

# False Path Pruning

Integer Range | Disequality | Branch




Impossible

“a in [0,0]”

“a != 0”

“a == 0 is true”

# Outline

- General discussion of tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
    - Reducing false positive using circumstantial evidence
  - Sample security-related results
- Static analysis for Android malware
  - …

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# Environment Assumptions

- **Should the return value of malloc() be checked?**

```
int *p = malloc(sizeof(int));
*p = 42;
```

OS Kernel:
Crash machine.

File server:
Pause filesystem.

Web application:
200ms downtime

Spreadsheet:
Lose unsaved changes.

Game:
Annoy user.

IP Phone:
Annoy user.

Library:
?

Medical device:
malloc?!

# Statistical Analysis

- **Assume the code is usually right**

| 3/4 deref | | | 1/4 deref |
|---|---|---|---|
| | `int *p = malloc(sizeof(int));`<br>`*p = 42;` | `int *p = malloc(sizeof(int));`<br>`if(p) *p = 42;` | |
| | `int *p = malloc(sizeof(int));`<br>`*p = 42;` | `int *p = malloc(sizeof(int));`<br>`if(p) *p = 42;` | |
| | `int *p = malloc(sizeof(int));`<br>`*p = 42;` | `int *p = malloc(sizeof(int));`<br>`if(p) *p = 42;` | |
| | `int *p = malloc(sizeof(int));`<br>`if(p)  *p = 42;` | `int *p = malloc(sizeof(int));`<br>`*p = 42;` | |

# Outline

- General discussion of tools
  – Goals and limitations
  – Approach based on abstract states
- More about one specific approach
  – Property checkers from Engler et al., Coverity
  – Sample security-related results
- Static analysis for Android malware
  – …

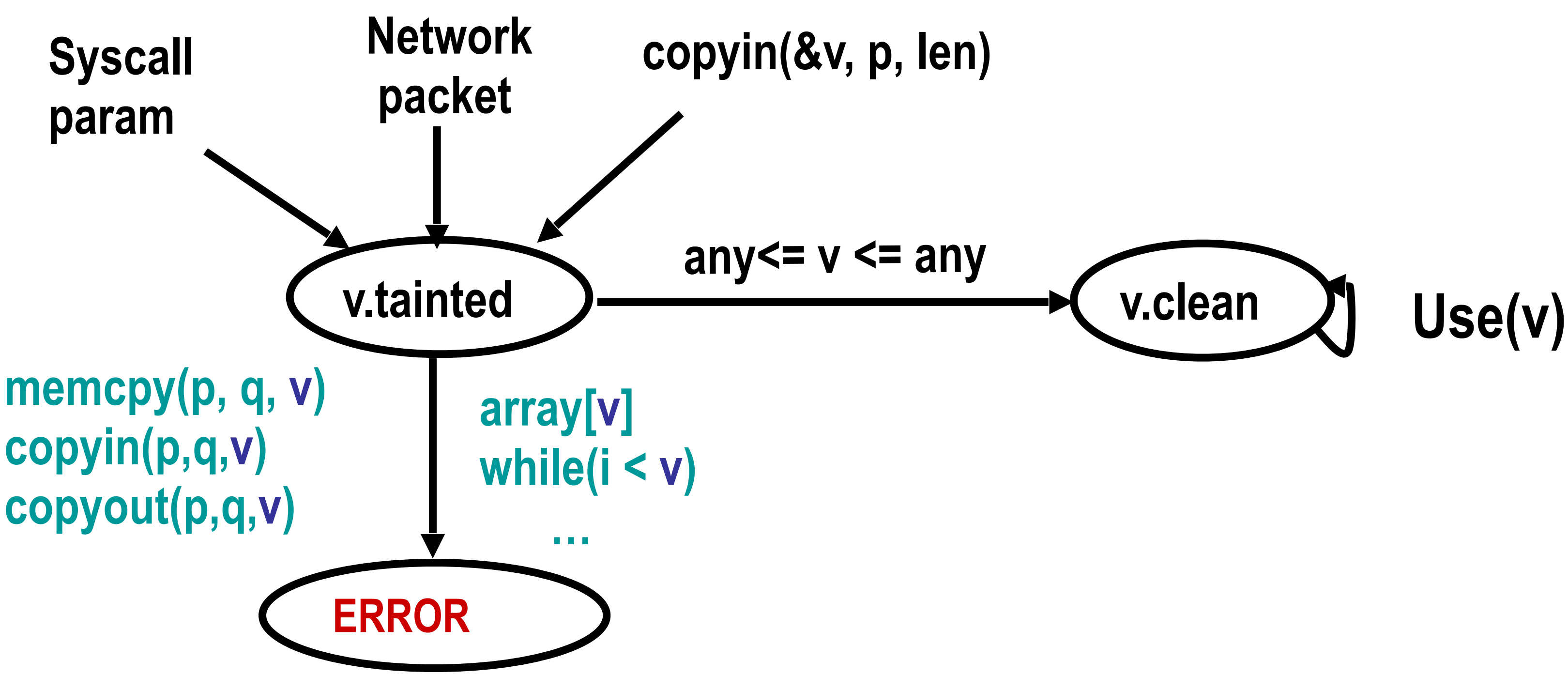
Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# Application to Security Bugs

- **Stanford research project**
  - Ken Ashcraft and Dawson Engler, Using Programmer-Written Compiler Extensions to Catch Security Holes, IEEE Security and Privacy 2002
  - Used modified compiler to find over 100 security holes in Linux and BSD
  - http://www.stanford.edu/~engler/
- **Benefit**
  - Capture recommended practices, known to experts, in tool available to all

# Sanitize integers before use

Warn when unchecked integers from untrusted sources reach trusting sinks




Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

# Example security holes

- **Remote exploit, no checks**

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
   msg = skb->data;
   ...
   memcpy(cmd.parm.setup.phone,
          msg->msg.connect_ind.addr.num,
          msg->msg.connect_ind.addr.len - 1);
```

# Example security holes

- **Missed lower-bound check:**

```
/* 2.4.5/drivers/char/drm/i810_dma.c */

if(copy_from_user(&d, arg, sizeof(arg)))
   return -EFAULT;
if(d.idx > dma->buf_count)
   return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

# Results for BSD and Linux

- **All bugs released to implementers; most serious fixed**

| | Linux | | BSD | |
|---|---|---|---|---|
| Violation | Bug | Fixed | Bug | Fixed |
| Gain control of system | 18 | 15 | 3 | 3 |
| Corrupt memory | 43 | 17 | 2 | 2 |
| Read arbitrary memory | 19 | 14 | 7 | 7 |
| Denial of service | 17 | 5 | 0 | 0 |
| Minor | 28 | 1 | 0 | 0 |
| Total | 125 | 52 | 12 | 12 |

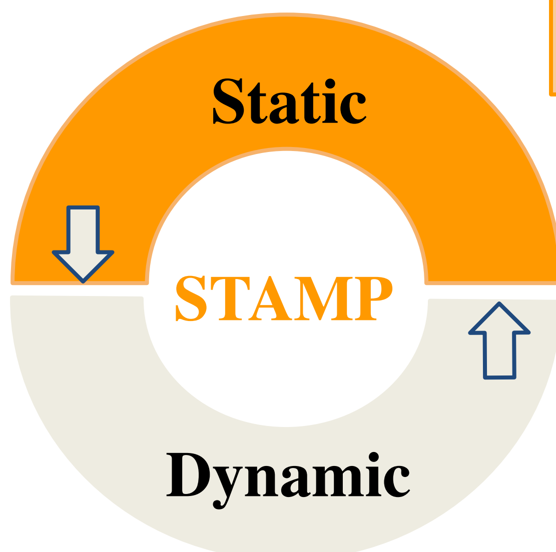
# Outline

- General discussion of tools
  - Goals and limitations
  - Approach based on abstract states
- More about one specific approach
  - Property checkers from Engler et al., Coverity
  - Sample security-related results
- Static analysis for Android malware
  - …

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,

# STAMP Admission System

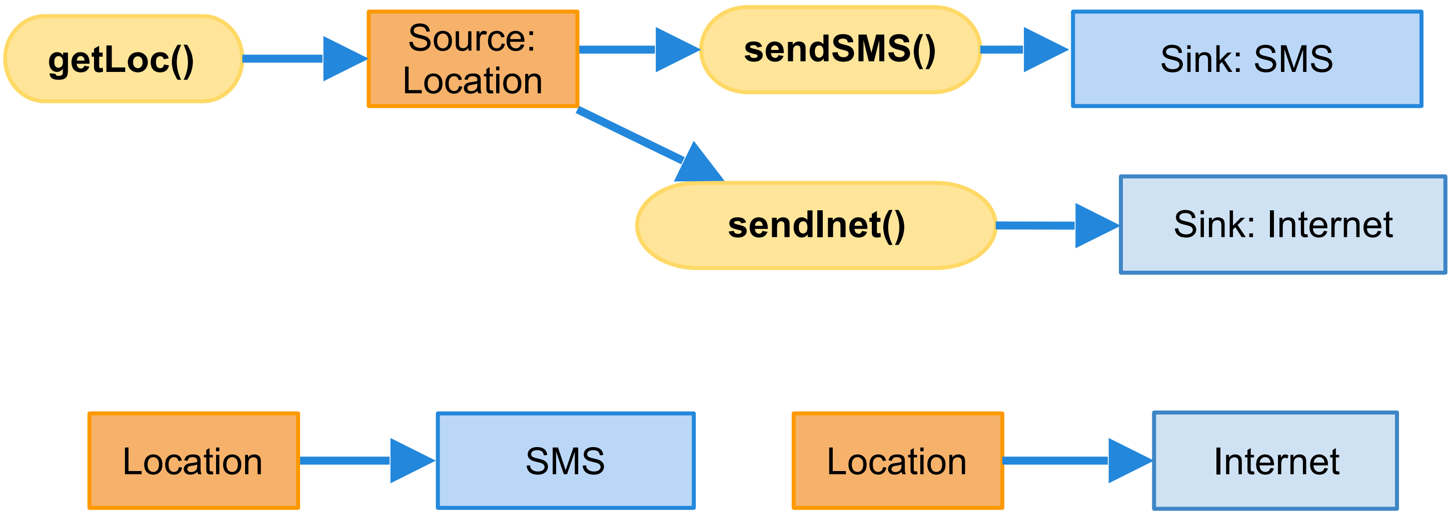
**Static Analysis**
More behaviors,
fewer details




**Dynamic Analysis**
Fewer behaviors,
more details


Alex Aiken,
John Mitchell,
Saswat Anand,
Jason Franklin
Osbert Bastani,
Lazaro Clapp,
Patrick Mutchler,
Manolis Papadakis

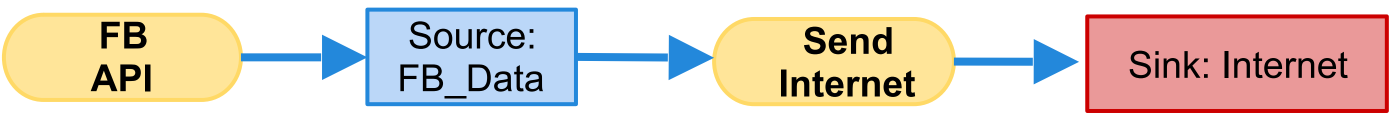
# Data Flow Analysis




- Source-to-sink flows
  - Sources: Location, Calendar, Contacts, Device ID etc.
  - Sinks: Internet, SMS, Disk, etc.

# Applications of Data Flow Analysis

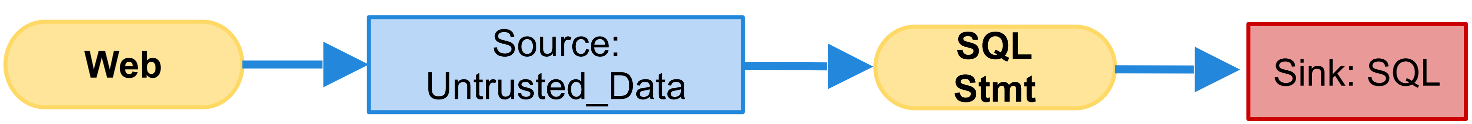
- Malware/Greyware Analysis
  - Data flow summaries enable enterprise-specific policies
- API Misuse and Data Theft Detection




- Automatic Generation of App Privacy Policies
  - Avoid liability, protect consumer privacy

**Privacy Policy**
This app collects your:
Contacts
Phone Number
Address

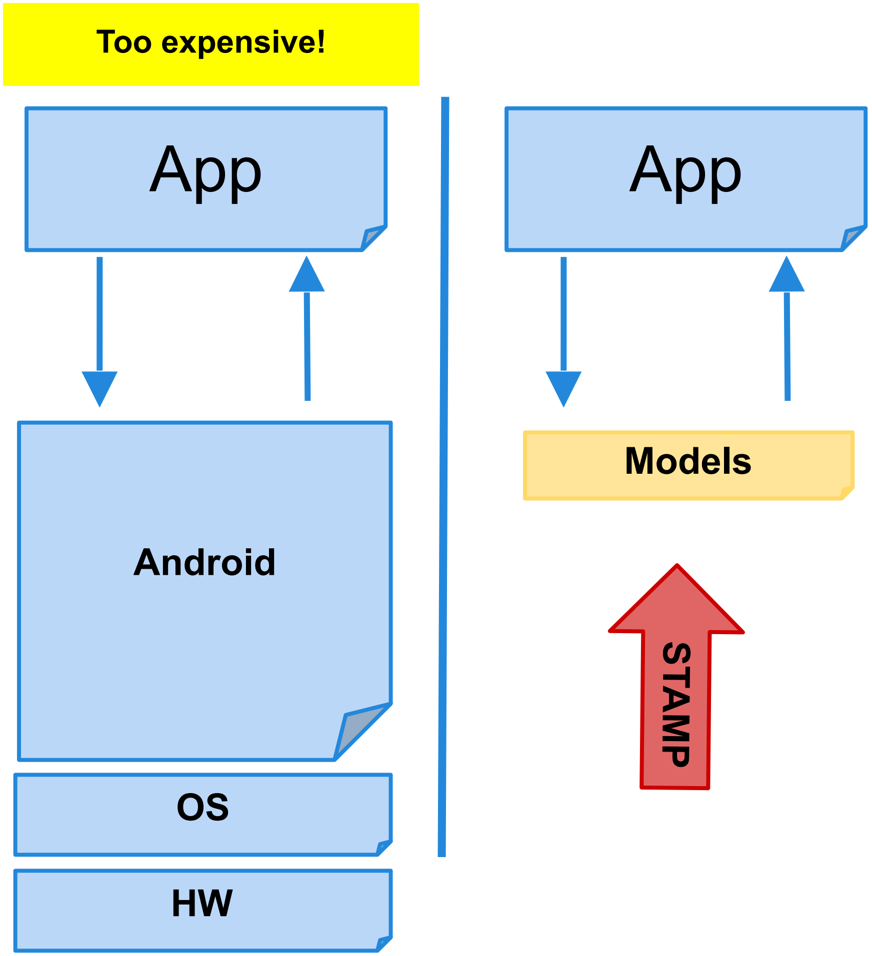
- Vulnerability Discovery



# Challenges

- Android is 3.4M+ lines of complex code
  - Uses reflection, callbacks, native code
- **Scalability:** Whole system analysis impractical
- **Soundness:** Avoid missing flows
- **Precision:** Minimize false positives

# STAMP Approach




- Model Android/Java
  - Sources and sinks
  - Data structures
  - Callbacks
  - 500+ models
- Whole-program analysis
  - Context sensitive

# Data We Track (Sources)

- Account data
- Audio
- Calendar
- Call log
- Camera
- Contacts
- Device Id
- Location
- Photos (Geotags)
- SD card data
- SMS

30+ types of sensitive data

# Data Destinations (Sinks)

- `Internet (socket)`
- `SMS`
- `Email`
- `System Logs`
- `Webview/Browser`
- `File System`
- `Broadcast Message`

10+ types of exit points

# Currently Detectable Flow Types

396 Flow Types

Unique Flow Types = Sources x Sink

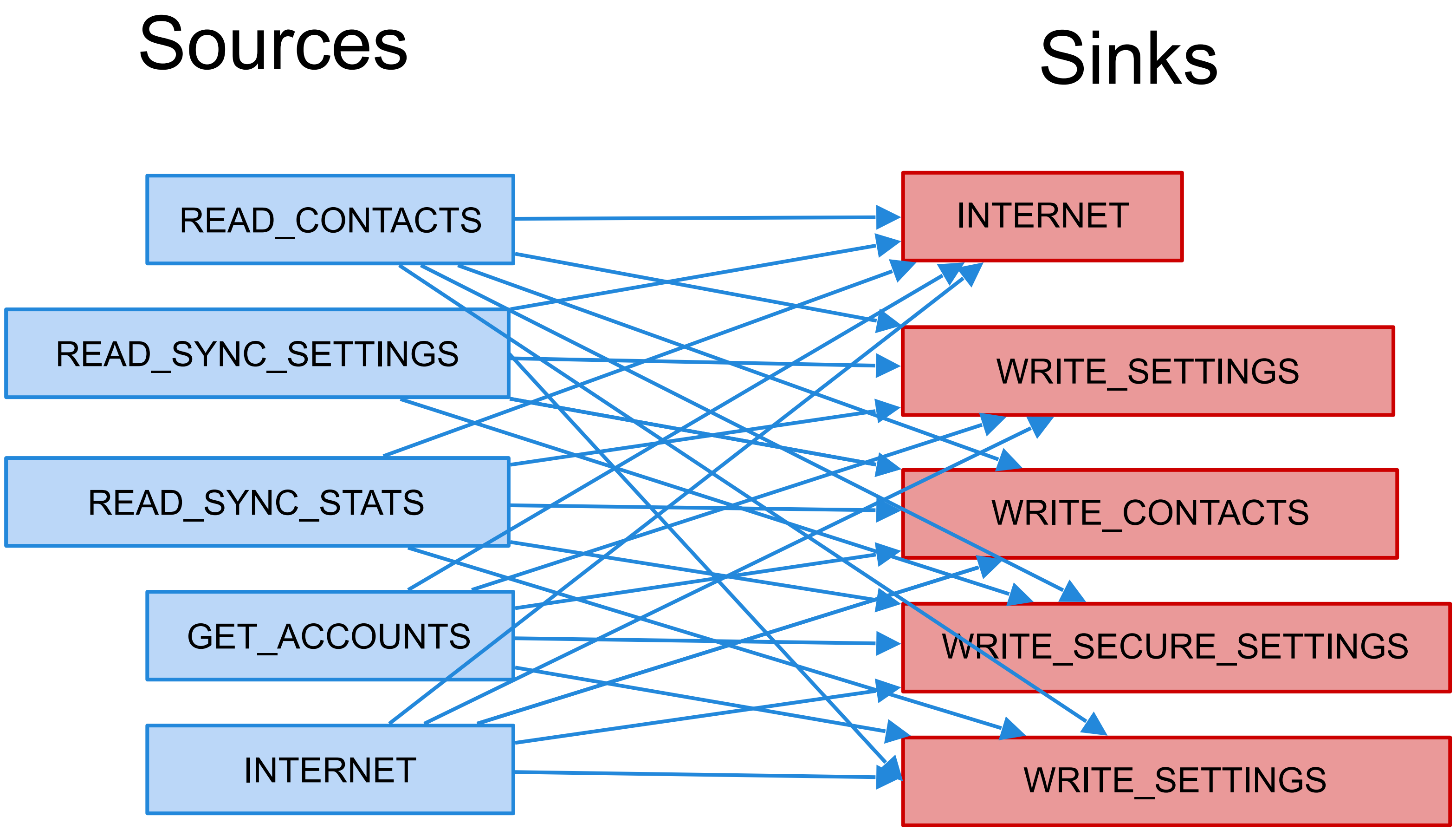
# Example Analysis

## Contact Sync for Facebook (unofficial)

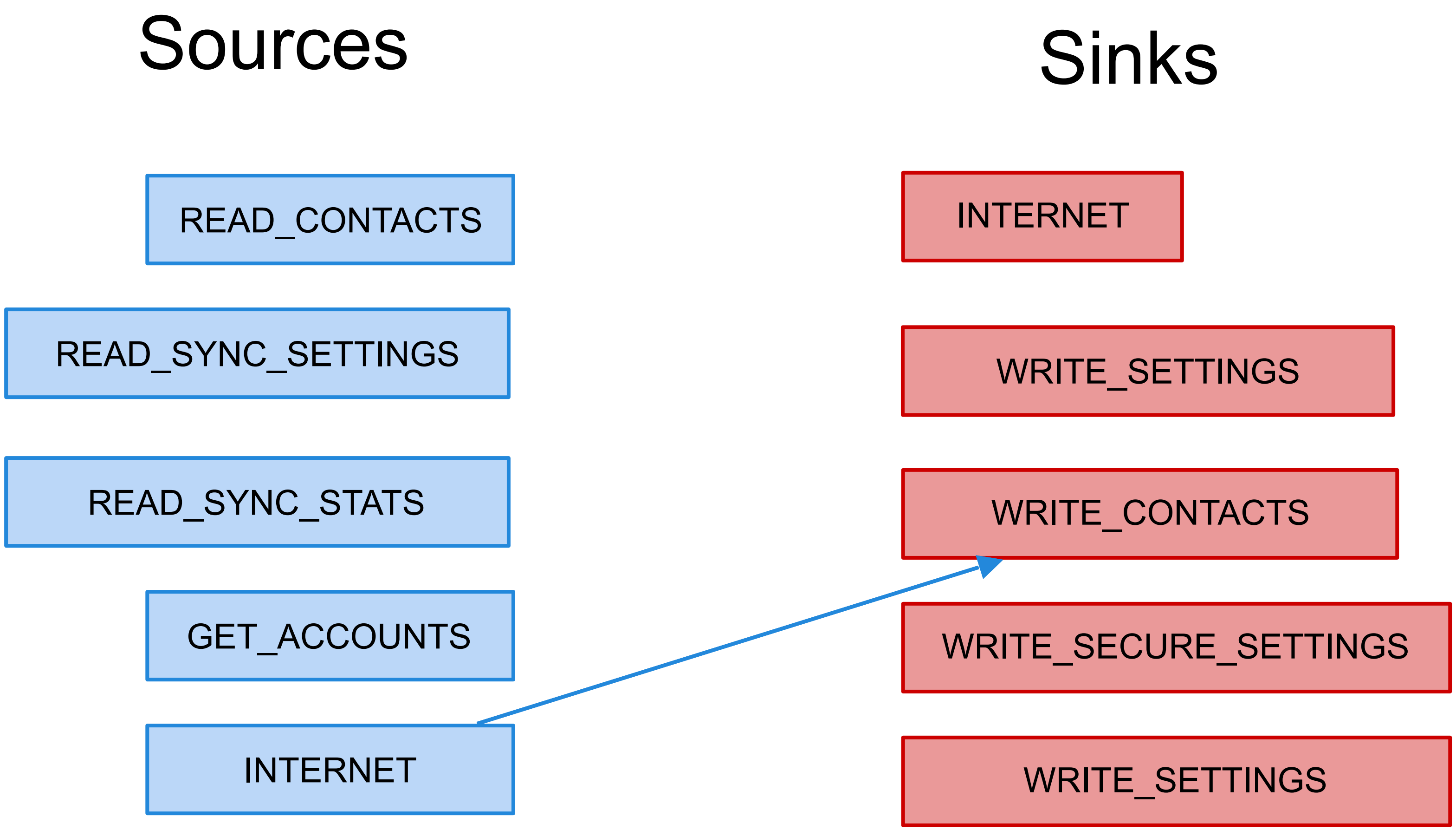
# Contact Sync Permissions

| Category | Permission | Description |
| --- | --- | --- |
| Your Accounts | AUTHENTICATE_ACCOUNTS | Act as an account authenticator |
| | MANAGE_ACCOUNTS | Manage accounts list |
| | USE_CREDENTIALS | Use authentication credentials |
| **Network Communication** | **INTERNET** | **Full Internet access** |
| | ACCESS_NETWORK_STATE | View network state |
| **Your Personal Information** | **READ_CONTACTS** | **Read contact data** |
| | WRITE_CONTACTS | Write contact data |
| System Tools | WRITE_SETTINGS | Modify global system settings |
| | WRITE_SYNC_SETTINGS | Write sync settings (e.g. Contact sync) |
| | READ_SYNC_SETTINGS | Read whether sync is enabled |
| | READ_SYNC_STATS | Read history of syncs |
| **Your Accounts** | **GET_ACCOUNTS** | **Discover known accounts** |
| Extra/Custom | WRITE_SECURE_SETTINGS | Modify secure system settings |

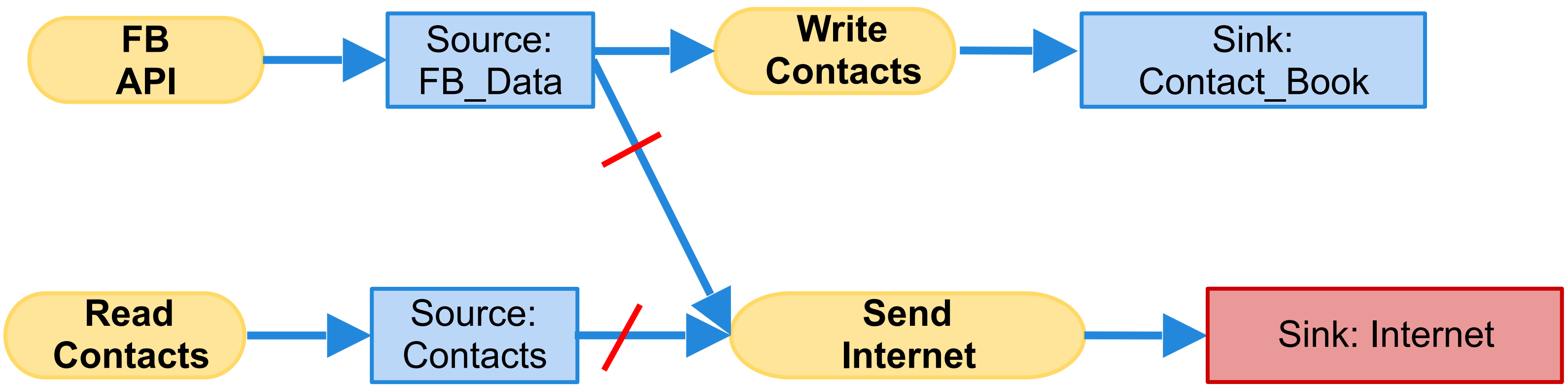
# Possible Flows from Permissions



# Expected Flows



# Observed Flows



# Example Study: Mobile Web Apps

- Goal

  Identify security concerns and vulnerabilities specific to mobile apps that access the web using an embedded browser

- Technical summary
  - WebView object renders web content
  - methods loadUrl, loadData, loadDataWithBaseUrl, postUrl
  - addJavascriptInterface(obj, name) allows JavaScript code in the web content to call Java object method name.foo()

# Sample results

Analyze 998,286 free web apps from June 2014

| Mobile Web App Feature | % Apps |
|---|---|
| JavaScript Enabled | 97 |
| JavaScript Bridge | 36 |
| shouldOverrideUrlLoading | 94 |
| shouldInterceptRequest | 47 |
| onReceivedSslError | 27 |
| postUrl | 2 |
| Custom URL Patterns | 10 |

| Vuln | % Relevant | % Vulnerable |
|---|---|---|
| Unsafe Navigation | 15 | 34 |
| Unsafe Retrieval | 40 | 56 |
| Unsafe SSL | 27 | 29 |
| Exposed POST | 2 | 7 |
| Leaky URL | 10 | 16 |

# Summary

- Static vs dynamic analyzers
- General properties of static analyzers
  - Fundamental limitations
  - Basic method based on abstract states
- More details on one specific method
  - Property checkers from Engler et al., Coverity
  - Sample security-related results
- Static analysis for Android malware
  - STAMP method, sample studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken,