Security Analyses
For The Lazy Superhero

One-Slide Summary

- We can ***statically detect*** buffer overruns in programs by modeling the space allocated for a buffer and the space used for a buffer. We ***cannot be right*** all the time.
- ***SQL injection*** and ***cross-site scripting*** attacks occur when evil user input is used (parsed) as part of another important ***language*** (e.g., HTML or SQL).
- Program analyses are expensive; recent research can ***randomize*** them to save time.

Lecture Outline

- Static Analyses to Detect Buffer Overruns
  - Strings
  - Alloc, Used
  - Constraints
- SQL Injection Attacks
  - Untrusted User Strings
  - Interpreted as valid SQL
- Randomized Dataflow Analysis
  - Random Join

Static Analysis to Detect Buffer Overruns

- Detecting buffer overruns *before* distributing code would be better

- Idea: Build a tool similar to a type checker to detect buffer overruns

- This is a popular research area; we'll present one idea at random [Wagner, Aiken, …]
  - You'll see more in later lectures

Focus on Strings

- Most important buffer overrun exploits are through **string** buffers
  - Reading an untrusted string from the network, keyboard, etc.
- Focus the tool only on arrays of characters

Idea 1: Strings as an Abstract Data Type

- A problem: Pointer operations and array dereferences are very difficult to analyze statically
  - Where does *ptr point?
  - What does buf[j] refer to?

- Idea: Model effect of string library functions directly
  - Hard code effect of strcpy, strcat, etc.

Idea 2: The Abstraction

- Model buffers as pairs of integer ranges
  - *Alloc* min allocated size of the buffer in bytes
  - *Used* max number of bytes actually in use

- Use integer ranges
  - $[x,y] = \{ x, x+1, \ldots, y-1, y \}$
  - Alloc and used cannot be computed exactly

The Strategy

- For each program expression, write **constraints** capturing the alloc and used of its string subexpressions

- Solve the constraints for the entire program

- Check for each string variable s

$$\mathrm{used}(s) \leq \mathrm{alloc}(s)$$

The Constraints

char s[n];	$n = \text{alloc}(s)$
strcpy(dst,src)	$\text{used}(src) \leq \text{used}(dst)$
p = strdup(s)	$\text{used}(s) \leq \text{used}(p)$ & $\text{alloc}(s) \leq \text{alloc}(p)$
p[n] = ‘\0’	$\min(\text{used}(p),n+1)) \leq \text{used}(p)$

Constraint Solving

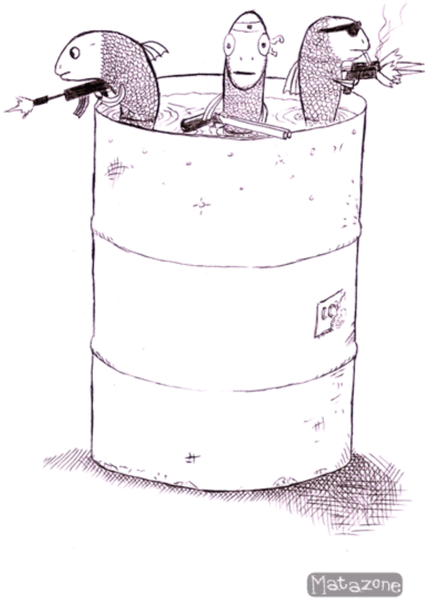
- Solving the constraints is akin to solving dataflow equations
  - Remember liveness? Constant prop?
- Build a graph
  - Nodes are len(s), alloc(s)
  - Edges are constraints len(s) $\leq$ len(t)

- Propagate information forward through the graph
  - Special handling of loops in the graph

Results

- This technique found new buffer overruns in *sendmail*
  - Which is like shooting fish in a barrel …
- Found new exploitable overruns in Linux *nettools* package

- Both widely used
- Previously hand-audited packages



Limitations

- Tool produces many **false positives** (*why?*)
  - 1 out of 10 warnings is a real bug

- Tool has false negatives (*why?*)
  - Unsound: may miss some overruns

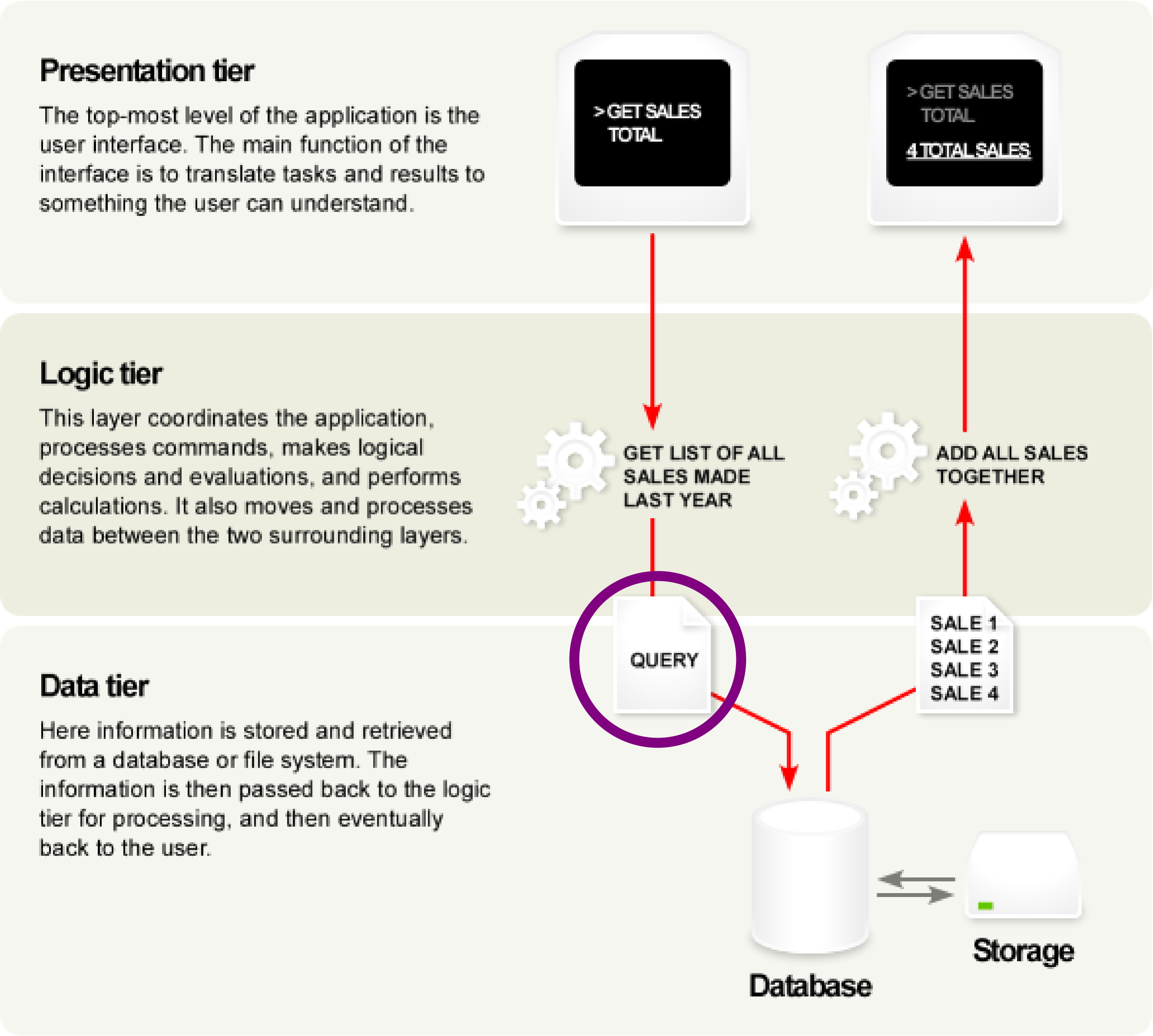
- But still productive to use
- So let's pretend we used it ...

Cat and Mouse

- Suppose I have a server (e.g., Amazon.com)
- Let's imagine that I have solved ...
  - Viruses: no malicious code on machine
  - Buffer overruns: no injection of evil assembly code
  - Buffer overruns: no non-control data attacks
  - Privileges: no running at root
  - Spam: as long as I'm dreaming, I'd like a pony ...
- I can still convince the server to do the wrong thing with the resources it legitimately has access to ...

Three-Tier Web Application

- This is how Amazon is structured
- Query is a SQL database command generated by program logic



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

The Problem In The Logic Tier

```
$userid = read_from_network();

if (!eregi('[0-9]+', $userid)) {
  unp_msg('You entered an invalid user ID.');
  exit;
}

$user = $DB->query(“SELECT * FROM `unp_user`”.
                   “WHERE userid='$userid'”);

if (!DB->is_single_row($user)) {
   unp_msg('You entered an invalid user ID.');
   exit;
}
```

The Problem

```
$userid = read_from_network();

if (!eregi('[0-9]+', $userid)) {
  unp_msg('You  tered an invalid user ID.');
  exit;
}

$user = $D                                  _user`".
                                             id'");

if (!DB->is_single_row($user)) {
   unp_msg('You entered an invalid user ID.');
   exit;
}
```

Matches any string that contains a sequence of digits...

The Bad Place

```
// $userid == “1'; DROP TABLE unp_user; --”

if (!eregi('[0-9]+', $userid)) {
  unp_msg('You entered an invalid user ID.');
  exit;
}

$user = $DB->query(“SELECT * FROM `unp_user`”.
                   “WHERE userid='$userid'”);

if (!DB->is_single_row($user)) {
   unp_msg('You entered an invalid user ID.');
   exit;
}
```

The Bad Place: Destroying Data

```
// $userid == "1'; DROP TABLE unp_user; --"

if
  '                                                      ;
  '
}

$user = $DB->q            "SELECT * FROM `unp_user`".
                          "WHERE userid='$userid'");

if (!DB->is_single_row($user)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}
```

```
SELECT * FROM `unp_user`
          WHERE userid='1';
DROP TABLE unp_user;
--'
```

Also A Bad Place: Viewing Data

```
// $userid == “1' OR 1 = 1 --”

if
                                                  ;

}

$user = $DB->q              “SELECT * FROM `unp_user`”.
                           “WHERE userid='$userid'”);

if (!DB->is_single_row($user)) {
   unp_msg('You entered an invalid user ID.');
   exit;
}
```

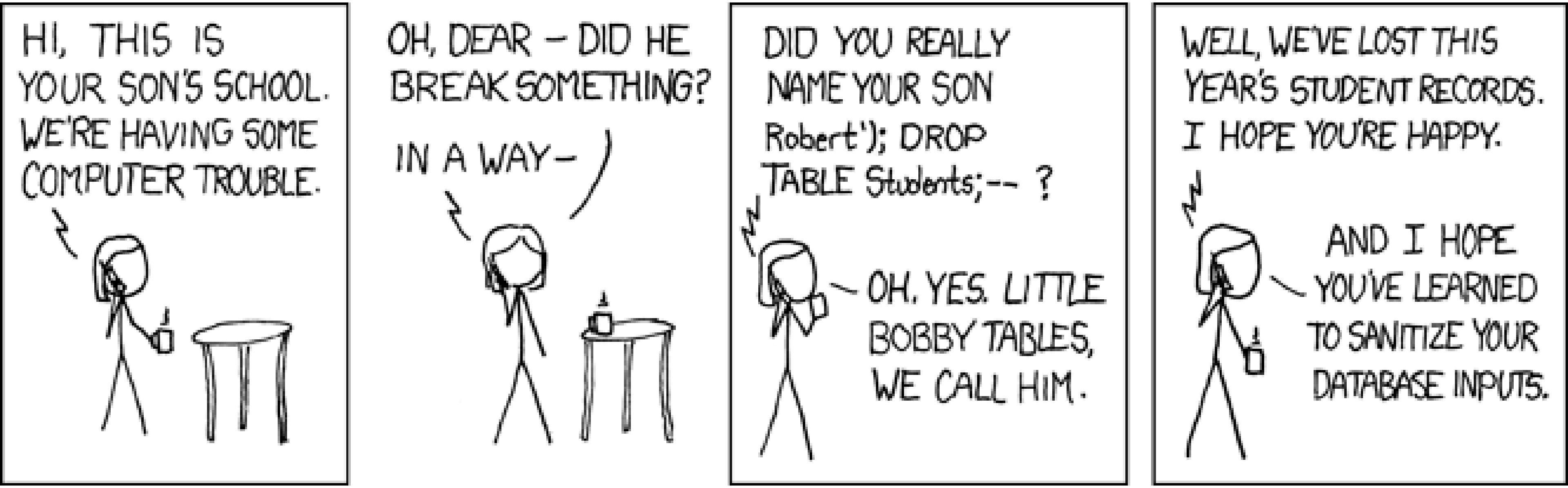
```
SELECT * FROM `unp_user`
             WHERE userid='1'
             OR 1 = 1
--'
```

SQL Code-Injection Vulnerabilities

- A *SQL injection* attack exploits a vulnerability in the database layer of an application whereby user input is incorrectly filtered for string literal escape characters or otherwise unexpected executed.
- Most common types of vulnerability in 2006:
  - **25.1% Cross-Site Scripting**
  - **14% SQL Command Injection**
  - 7.9% Buffer Overruns
- Attacks are easy and expose valuable data

Exploits Of A Mom

- The essence of SQL injection:



SQL Injection

- Note that it's basically a parsing problem
- We have a string constant in PHP plus a string constant from the user, and when combined they must make a valid SQL program
- One Solution: Dynamic Taint Analysis
  - Propagate a “taint” bit with every string
- One Solution: Dynamic Grammar Analysis
  - Partially parse PHP string fragment
  - If PHP string fragment + user string fragment parses to something with a different top-level structure, bail!

Parse Trees To The Rescue!

- Do the user input strings contribute to something "too high" on the parse tree?

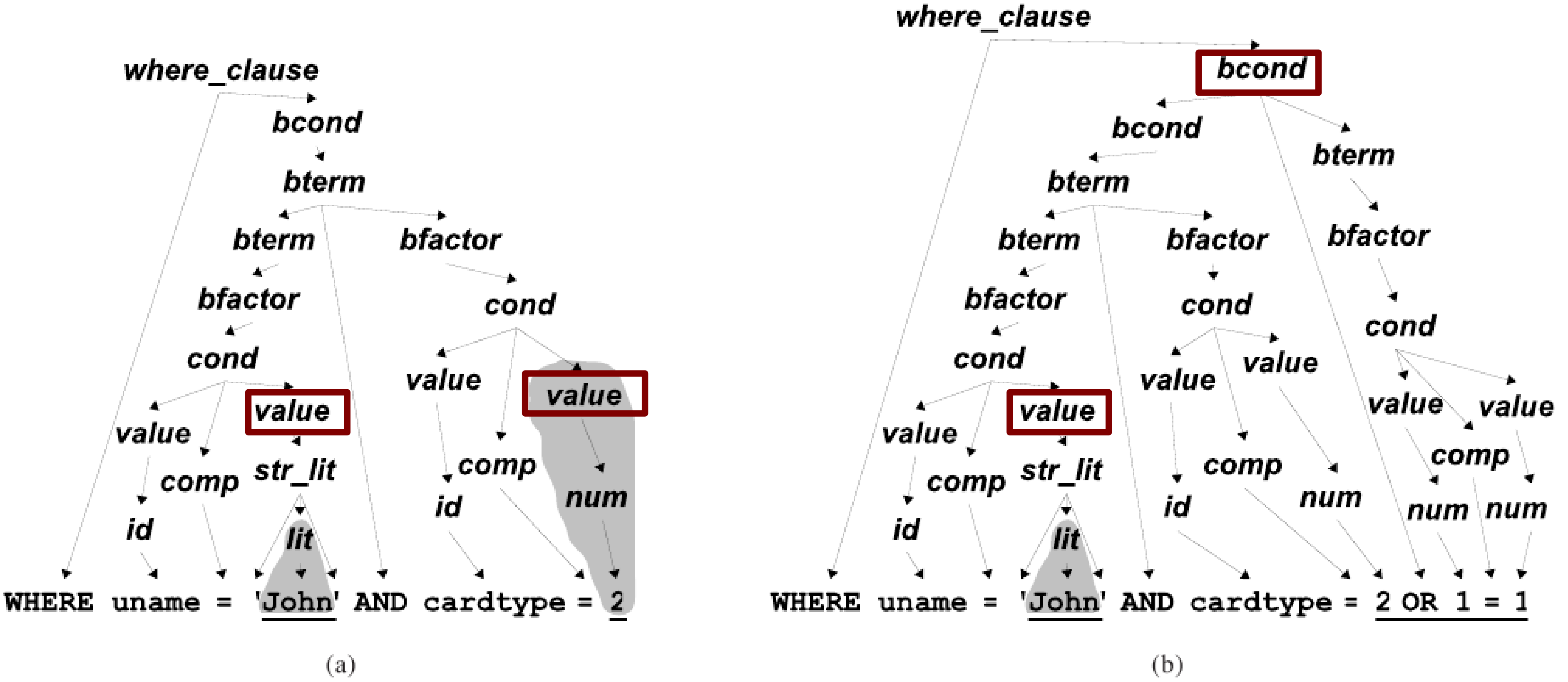


Figure 4. Parse trees for `WHERE` clauses of generated queries. Substrings from user input are underlined.

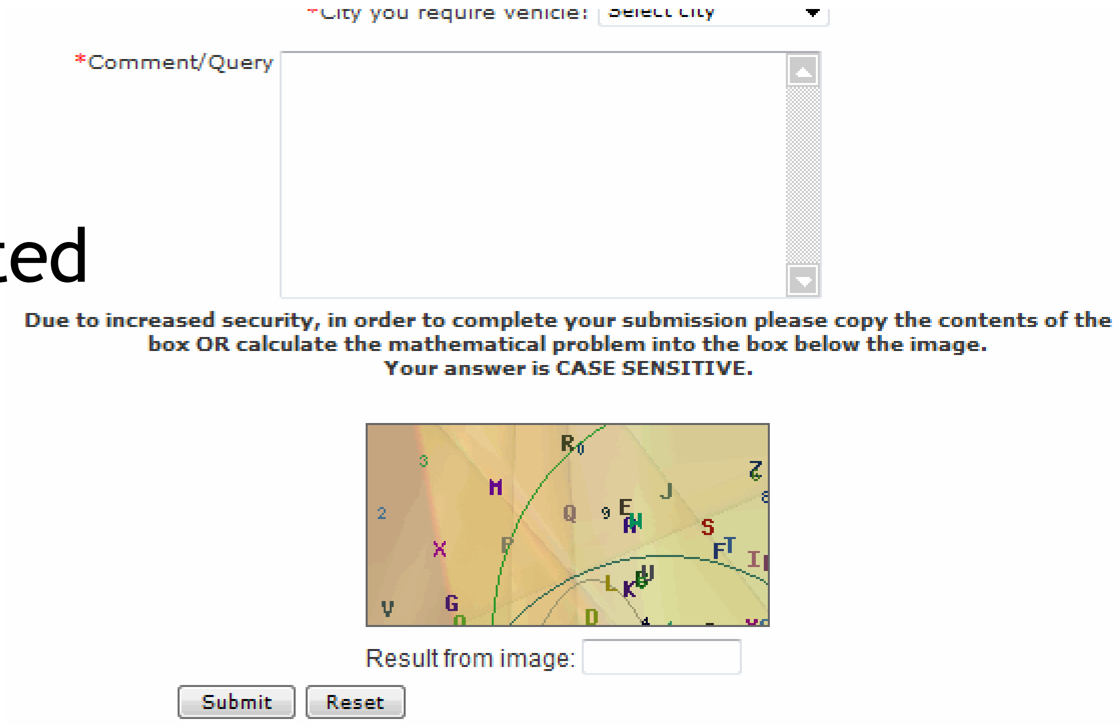
Su & Wassermann, *POPL '06*

Cross-Site Scripting

- ***Cross-Site Scripting*** (XSS) has the same flavor
- Evil User X posts a message with JavaScript in it (e.g., send passwords to me) to Blog B
  - Blog B can also be a forum, etc.
- Later, Luser browses Blog B
- Blog B sends over page data, including Evil X's Message
- Luser thinks it is from Blog B (misplaced trust)
- Luser renders and interprets it

Stopping Evil Posts

- Evil network-crawling robots try to post evil JavaScript to every forum they can find
- Let's require a real human when posting
- Increases cost
- *CAPTCHA*
  - Complete Automated
  - Public Turing test
  - to tell Computers
  - and Humans Apart



*City you require vehicle: Select City

*Comment/Query

Due to increased security, in order to complete your submission please copy the contents of the box OR calculate the mathematical problem into the box below the image.
Your answer is CASE SENSITIVE.

Result from image:

Have We Won Yet?

- CAPTCHAs fail in theory and in practice
- The overarching problem is exactly the same:
  - The server takes input from an untrusted user
  - That input may be interpreted by another parser later
    - In SQL-CIVs, by the database's SQL parser
    - In XSS, by a user's JavaScript parser
  - So all of the same techniques apply for XSS

Random Interpretation

Sumit Gulwani & George Necula

Probabilistically Sound Program Analysis!

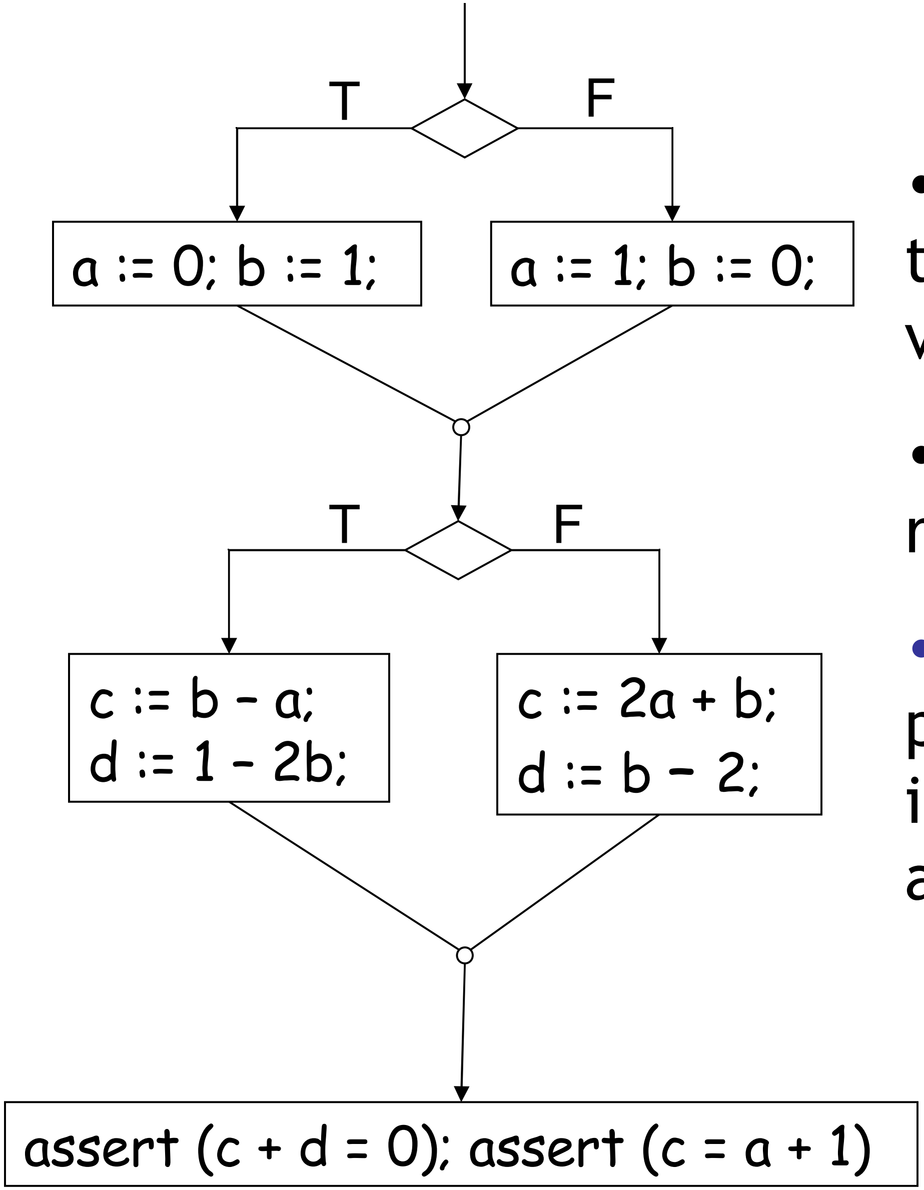
- Sound program analysis is hard (Rice's Theorem)
- PL researchers usually pay in terms of
  - Loss of completeness or precision
  - Complicated algorithms
  - Long running times

- Can we pay in terms of **soundness** instead?
  - Basically, *soundness* = *correctness*
  - Judgments are unsound with low probability
  - We can predict and control the probability of error
  - Can gain simplicity and efficiency

Discovering Affine Equalities

- Given a program (control-flow graph) …
- Discover equalities of the form $2y + 3z = 7$
  - Compiler Optimizations
  - Loop Invariants
  - Translation Validation
- There exist polynomial time deterministic algorithms [Karr 76]
  - involving expensive operations - $O(n^4)$
- We present a randomized algorithm
  - as complete as the deterministic algorithms
  - but faster - $O(n^2)$
  - and simpler (almost as simple as an interpreter)

Example 1



a := 0; b := 1;

a := 1; b := 0;

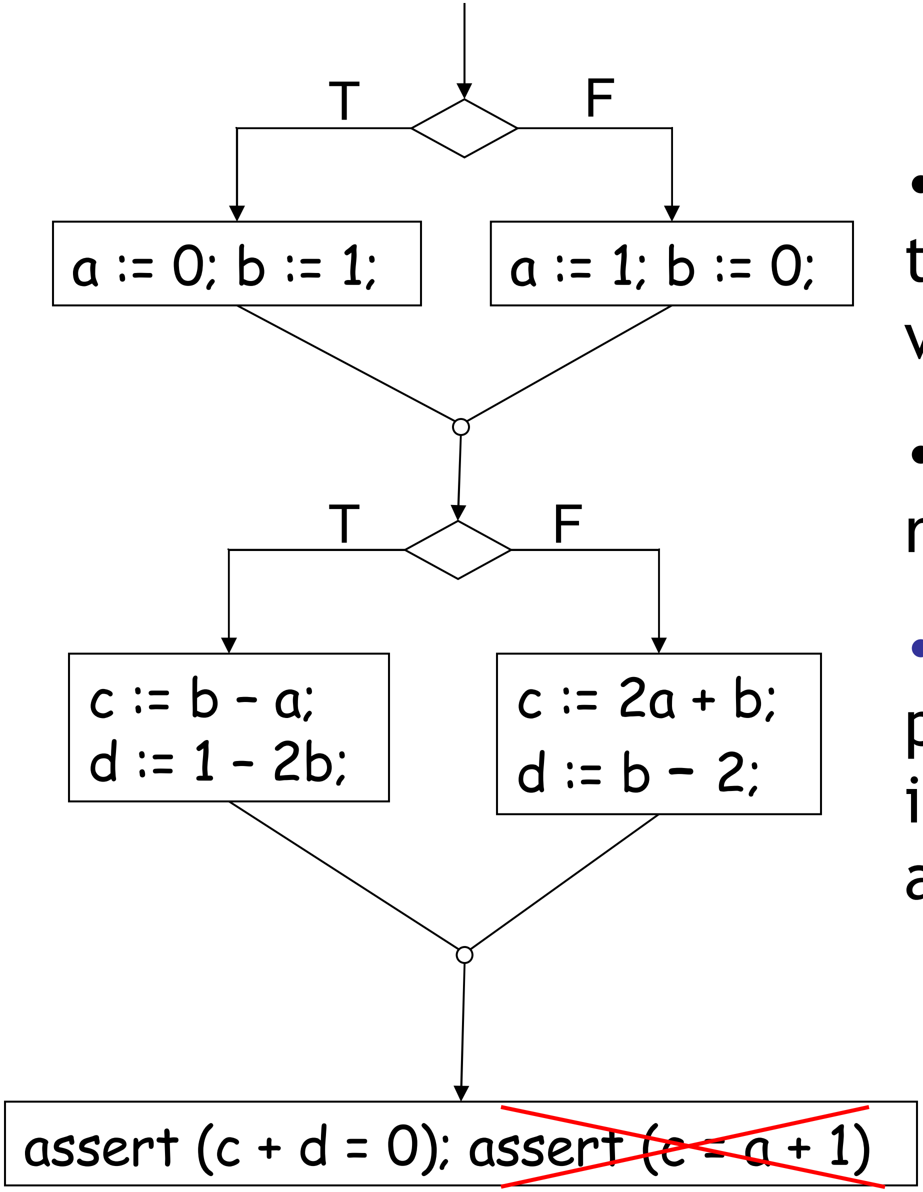
c := b – a;
d := 1 – 2b;

c := 2a + b;
d := b − 2;

assert (c + d = 0); assert (c = a + 1)

- Random testing will have to exercise all the 4 paths to verify the assertions
- Our algorithm is similar to random testing
- However, we execute the program once, in a way that it captures the “effect” of all the paths

Example 1



T F

a := 0; b := 1;

a := 1; b := 0;

T F

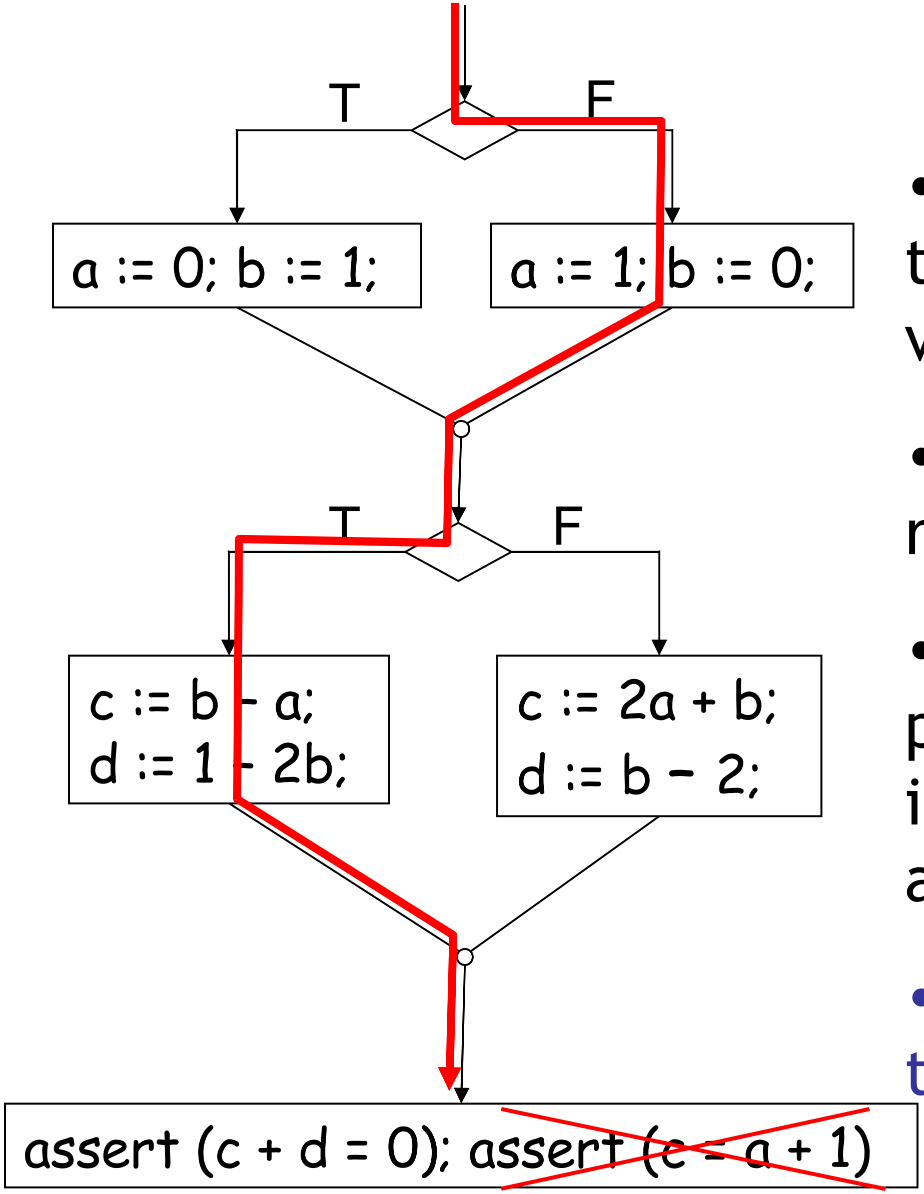
c := b – a;
d := 1 – 2b;

c := 2a + b;
d := b − 2;

assert (c + d = 0); ~~assert (c = a + 1)~~

- Random testing will have to exercise all the 4 paths to verify the assertions
- Our algorithm is similar to random testing
- However, we execute the program once, in a way that it captures the “effect” of all the paths

Example 1



T F

a := 0; b := 1;

a := 1; b := 0;

T F

c := b - a;
d := 1 - 2b;

c := 2a + b;
d := b - 2;

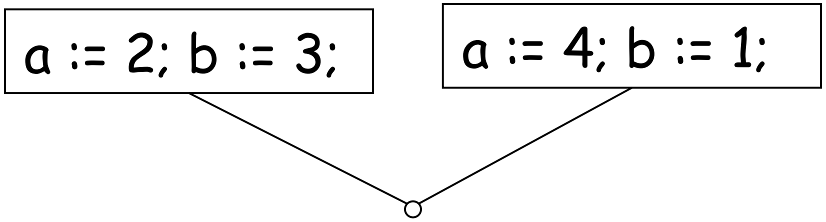
assert (c + d = 0); ~~assert (c = a + 1)~~

- Random testing will have to exercise all the 4 paths to verify the assertions
- Our algorithm is similar to random testing
- However, we execute the program once, in a way that it captures the "effect" of all the paths
- Exponential work, linear time! *(P=NP?)*

Idea #1: Affine Join Operation

- Execute both the branches
- Combine the values of the variables at joins using the affine join operation $\oplus_w$ for some randomly chosen w

$$v_1 \oplus_w v_2 = w \times v_1 + (1-w) \times v_2$$



a := 2; b := 3;

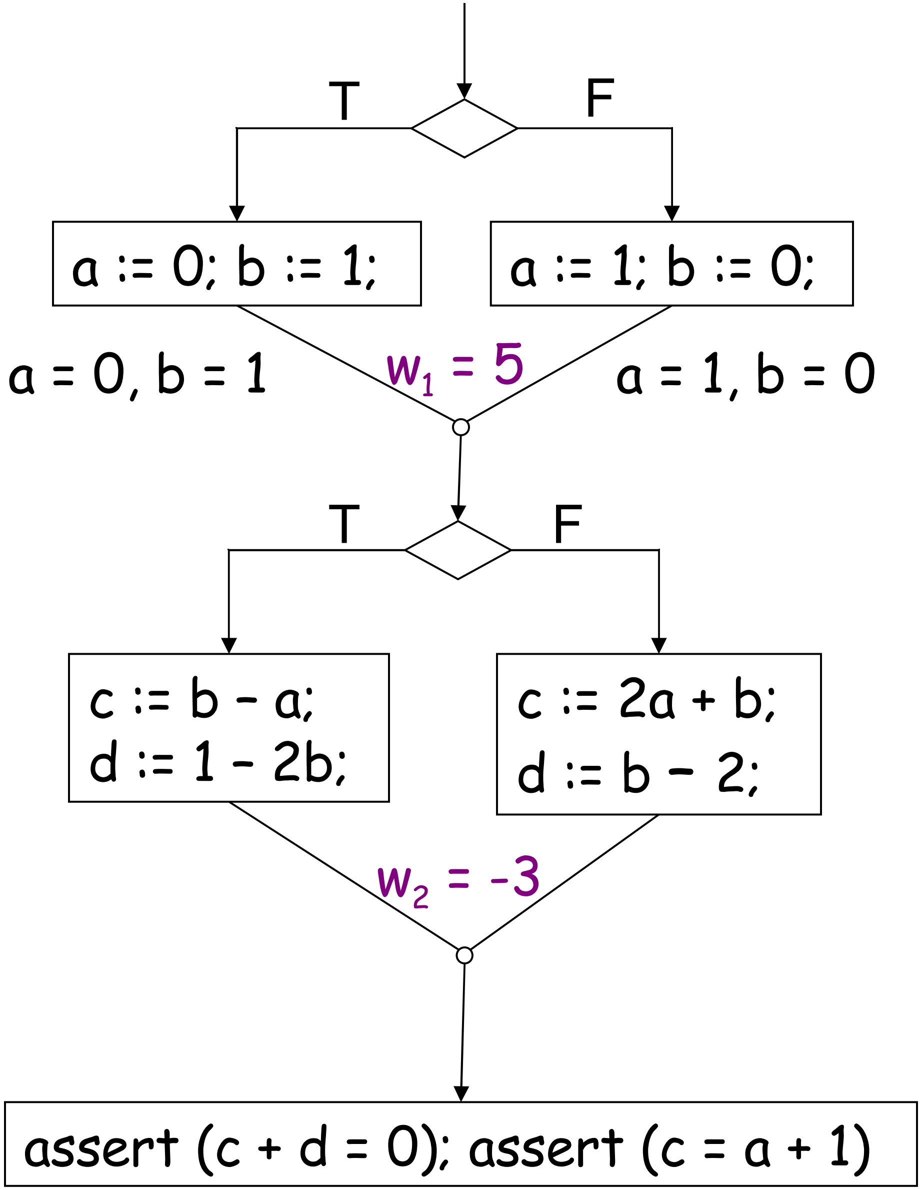
a := 4; b := 1;

$a = 2 \oplus_7 4 = -10$

$b = 3 \oplus_7 1 = 15$

$(w = 7)$

Example 1



a := 0; b := 1;

a := 1; b := 0;

a = 0, b = 1

$w_1 = 5$

a = 1, b = 0

c := b – a;
d := 1 – 2b;

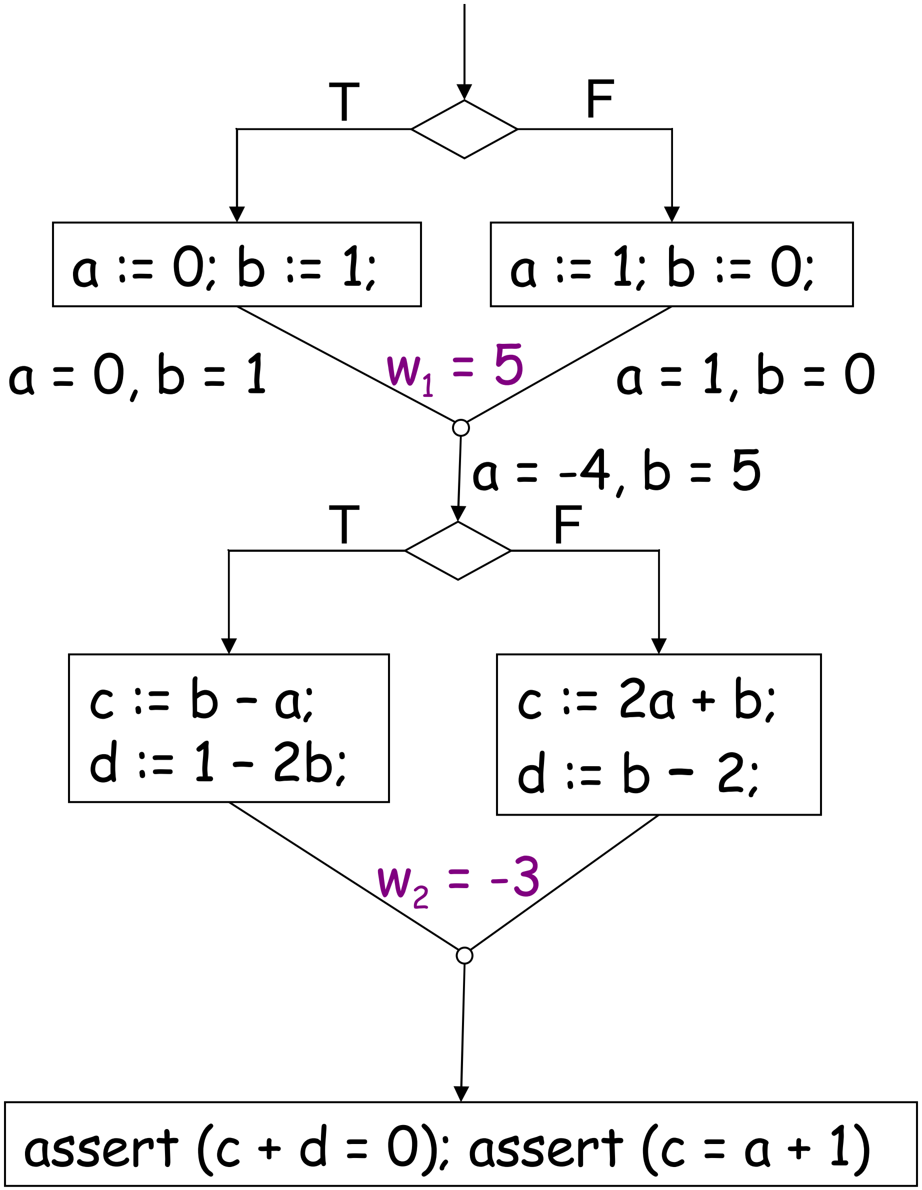
c := 2a + b;
d := b – 2;

$w_2 = -3$

assert (c + d = 0); assert (c = a + 1)

- Choose a random weight for each join independently.
- All choices of random weights verify the first assertion
- *Almost* all choices contradict the second assertion.

Example 1



T

F

a := 0; b := 1;

a := 1; b := 0;

a = 0, b = 1

$w_1 = 5$

a = 1, b = 0

a = -4, b = 5

T

F

c := b – a;
d := 1 – 2b;

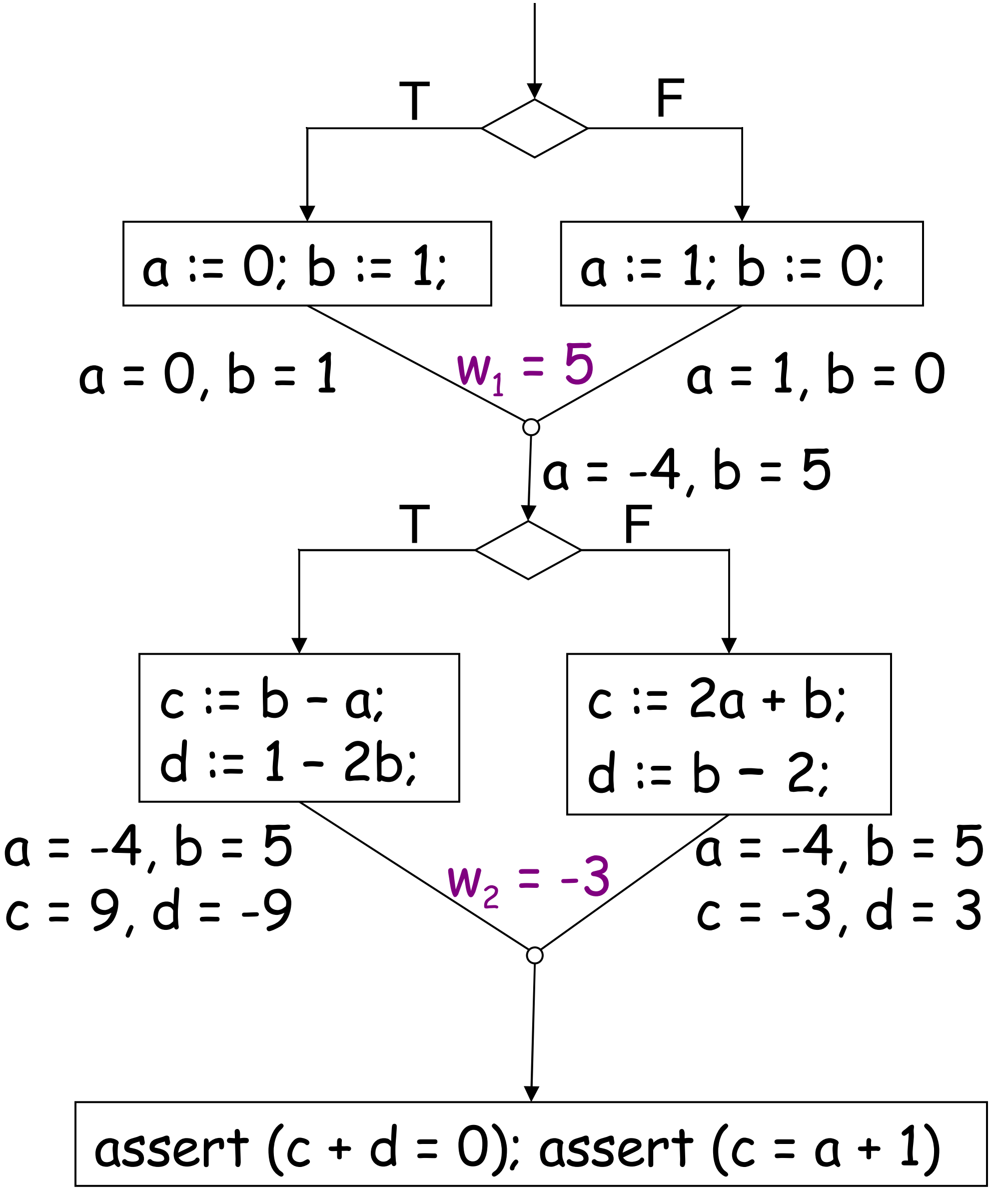
c := 2a + b;
d := b – 2;

$w_2 = -3$

assert (c + d = 0); assert (c = a + 1)

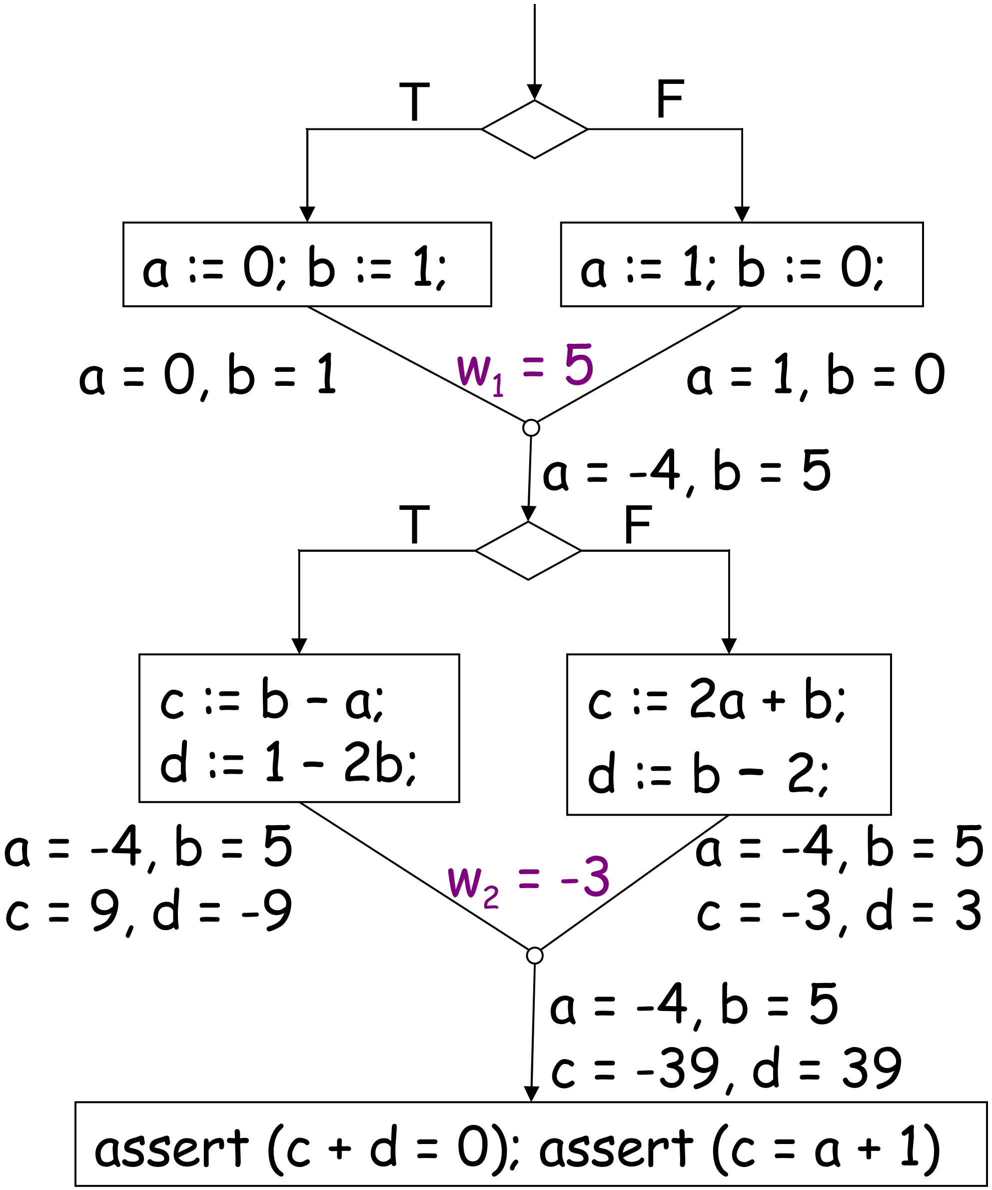
- Choose a random weight for each join independently.
- All choices of random weights verify the first assertion
- *Almost* all choices contradict the second assertion.

Example 1



- Choose a random weight for each join independently.
- All choices of random weights verify the first assertion
- *Almost* all choices contradict the second assertion.

Example 1

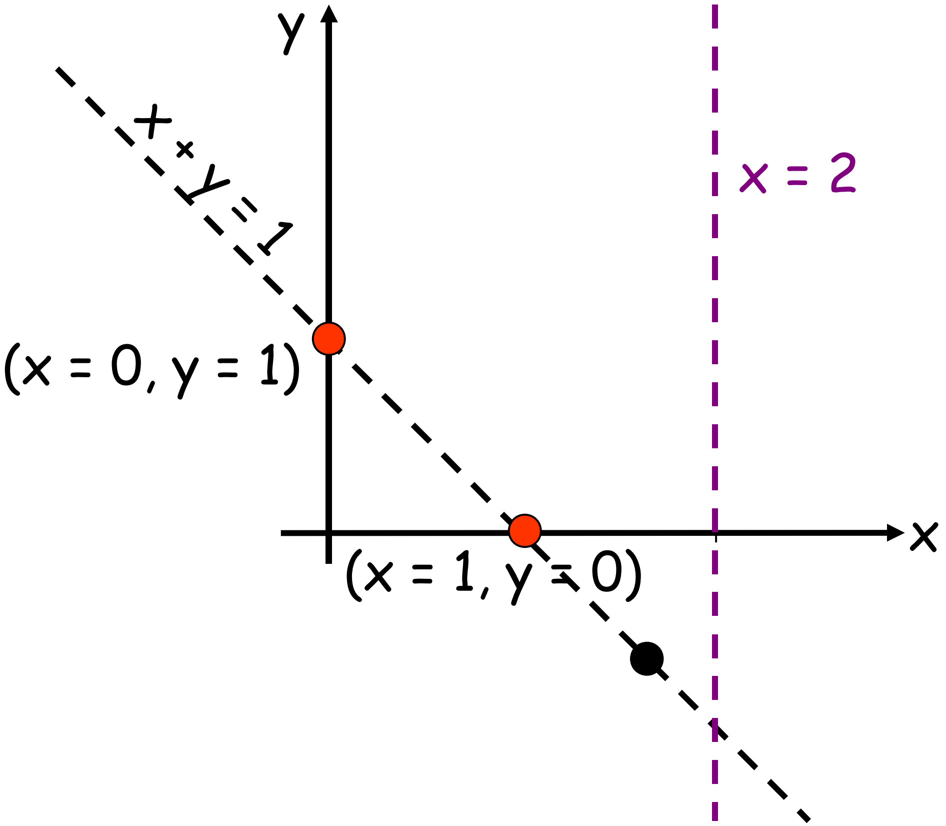


- Choose a random weight for each join independently.
- All choices of random weights verify the first assertion
- *Almost* all choices contradict the second assertion.

Geometric Interpretation of the Affine Join operation

🔴 : State before the join

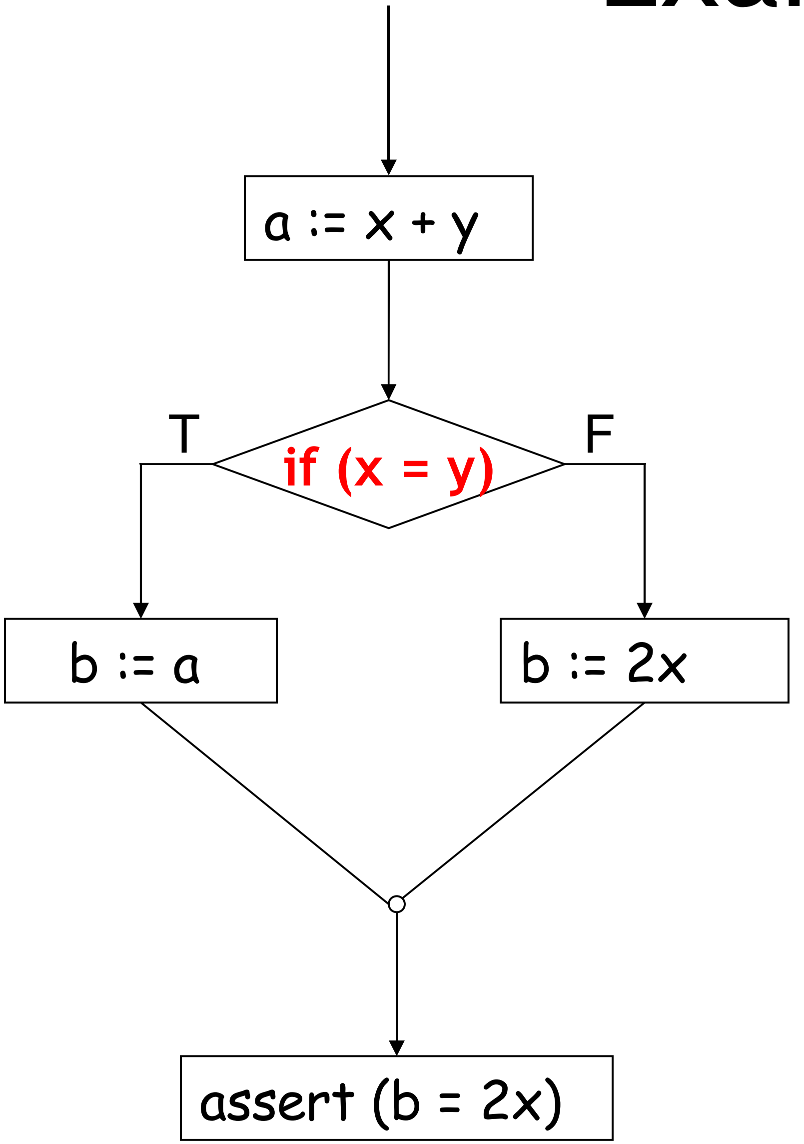
● : State after the join



● satisfies all the affine relationships that are satisfied by both 🔴 (e.g. x + y = 1, z = 0)

Given any relationship that is *not* satisfied by any of 🔴 (e.g. x=2), ● also does not satisfy it <u>with high probability</u>

Example 2

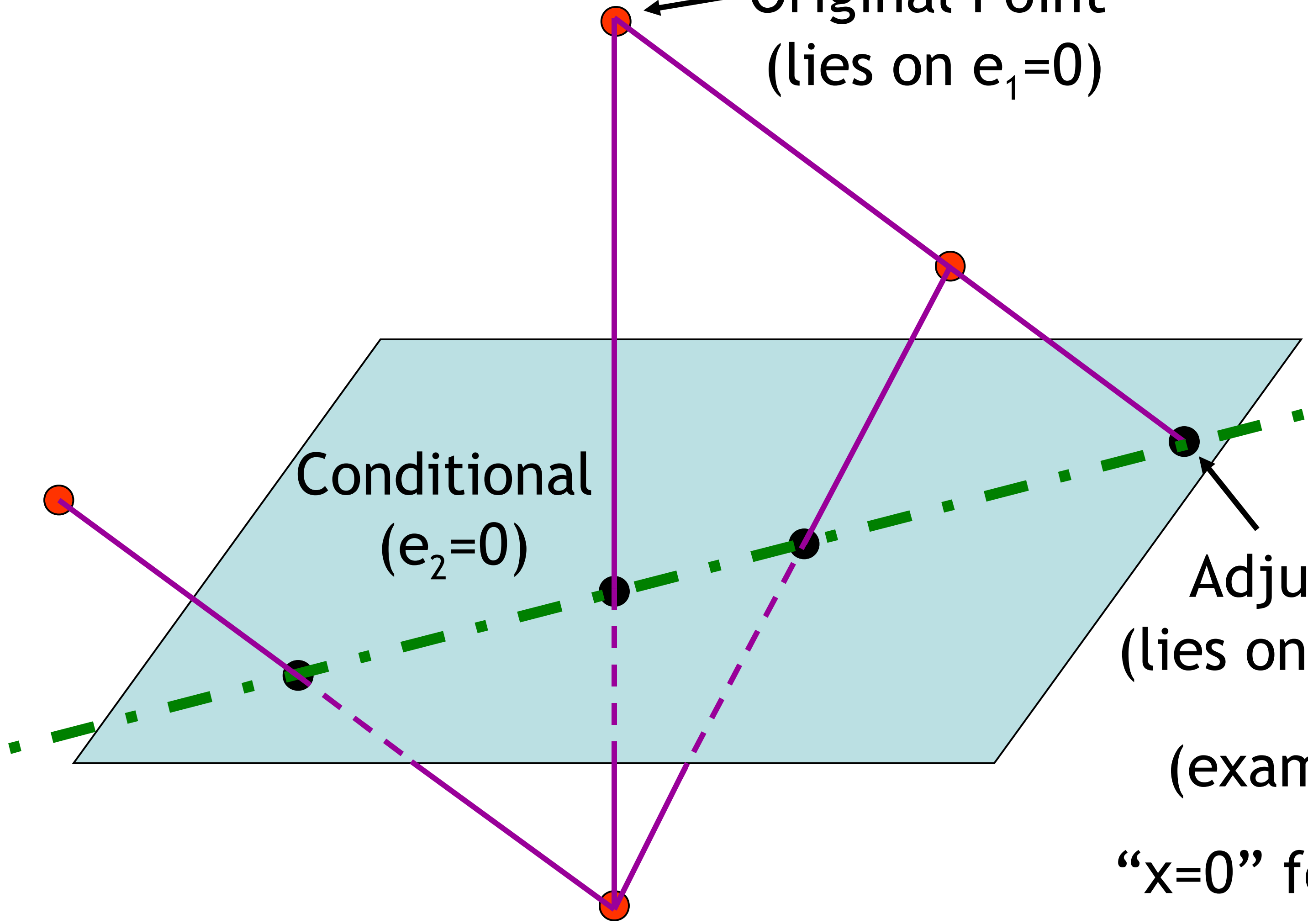


- Idea #1 is not enough
- We need to make use of the conditional x=y on the true branch

Idea #2: Adjust Operation

- Execute multiple runs of the program in parallel
- “Sample” = Collection of states at each program point
- “Adjust” the sample before a conditional (by taking affine joins of the states in the sample) such that
  - Adjustment preserves original relationships
  - Adjustment satisfies the equality in the conditional
- Use adjusted sample on the true branch

Geometric Interpretation of the Adjust Operation



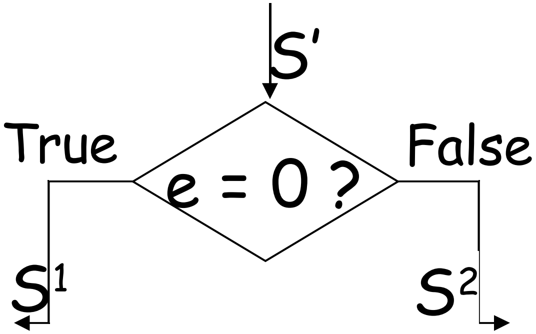
Original Point
(lies on $e_1=0$)

Conditional
($e_2=0$)

Adjusted Point
(lies on $e_1=0 \cap e_2=0$)

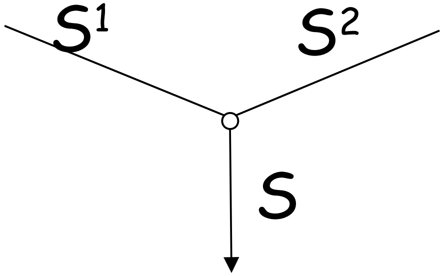
(examples show
"x=0" for simplicity)

The Randomized Interpreter R

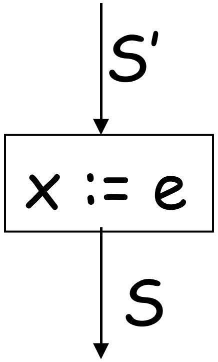


$S^1 = \mathrm{Adjust}(S', e)$

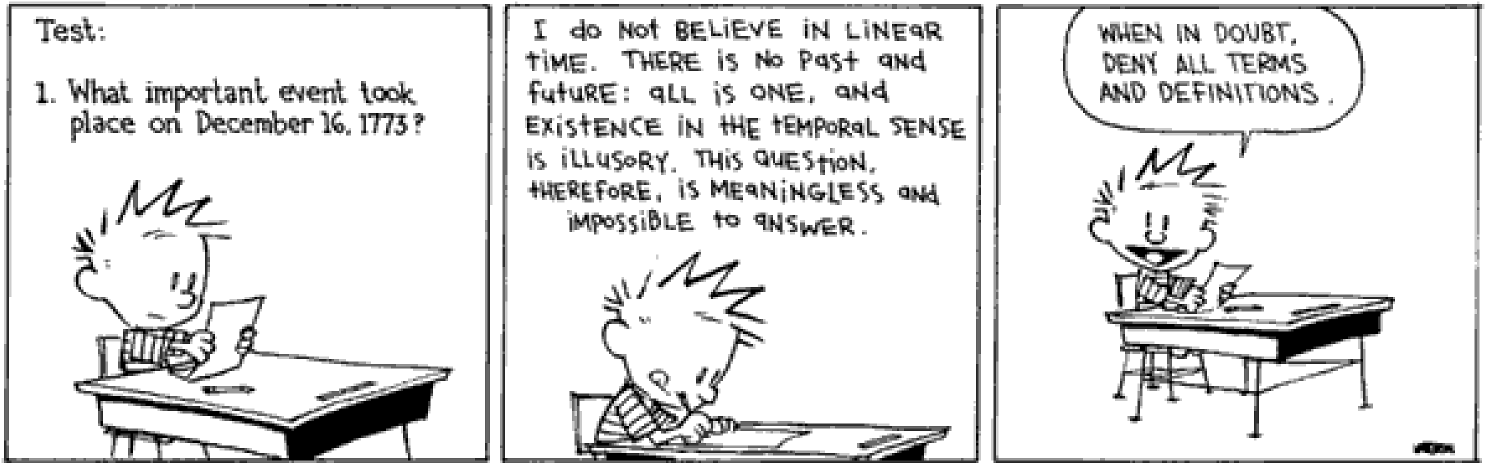
$S^2 = S'$



$S_i = S^1_i \oplus_{w_i} S^2_i$



$S_i = S'_i[x \leftarrow e]$



Completeness and soundness of R

- We compare the randomized interpreter R with a suitable actual interpreter A
  - Actual Interpreter A would be too slow (etc.) to use in real life!

- R mimics A with high probability
  - R is as complete as A
  - R is sound *with high probability*

Soundness Theorem

- If A $\not\Rightarrow$ g = 0, then *with high probability*
  R $\not\models$ g = 0
- Error probability $\leq (2d)^b \left(\frac{j+1}{d}\right)^r$
  - b: number of branches
  - j: number of joins
  - d: size of the field
  - r: number of points in the sample
- If j = b = 10, r = 15, d $\approx 2^{32}$, then
  error probability $\leq \frac{1}{2^{98}}$

Conclusions, Wessy Summary

- Randomization can help achieve simplicity and efficiency at the expense of making soundness probabilistic
- Has been extended to handle uninterpreted function symbols, interprocedural analyses, randomized decision procedures for theorem proving, combined abstract interpreters, …
- May help with complicated security analyses
- Go to grad school!

Homework

- Final Exam Soon ...