Static Analysis

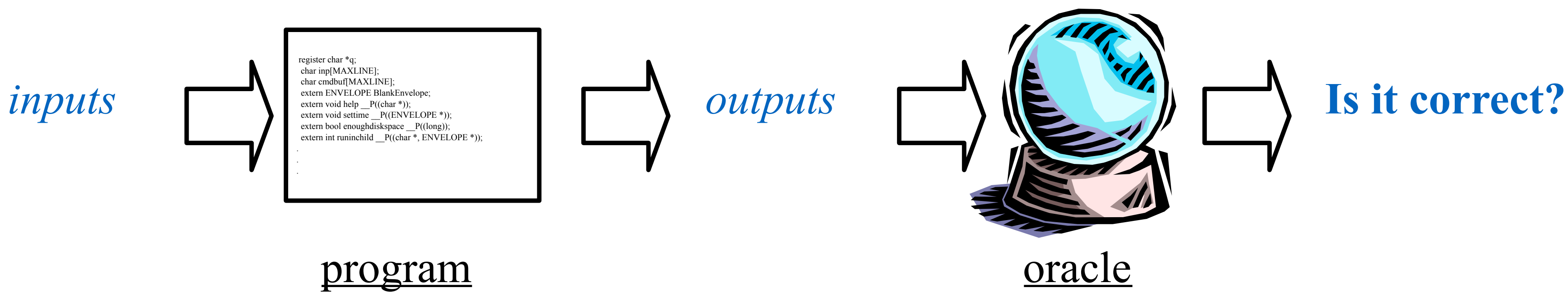
With material from Michelle Mazurek, Dave Levin, Mike Hicks, Dawson Engler, Lujo Bauer, and Jeff Foster

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html

Static Analysis / Symbolic Execution

Current Practice

for Software Assurance



- **Testing:** Check correctness on set of inputs
- **Benefits**: Concrete failure proves issue, aids fix
- **Drawbacks**: Expensive, difficult, coverage?
    - No guarantees

Current Practice

(continued)

- **Code audit:** Convince someone your code is correct
- **Benefit:** Humans can generalize
- **Drawbacks**: Expensive, hard, no guarantees


```
  {
    /* arrange for debugging output to go to remote host */
    (void) dup2(fileno(OutChannel), fileno(stdout));
  }
  settime(e);
  peerhostname = RealHostName;
  if (peerhostname == NULL)
    peerhostname = "localhost";
  CurHostName = peerhostname;
  CurSmtpClient = macvalue('_', e);
  if (CurSmtpClient == NULL)
    CurSmtpClient = CurHostName;

  setproctitle("server %s startup", CurSmtpClient);
#if DAEMON
  if (LogLevel > 11)
  {
    /* log connection information */
    sm_syslog(LOG_INFO, NOQID,
      "SMTP connect from %.100s (%.100s)",
      CurSmtpClient, anynet_ntoa(&RealHostAddr));
  }
#endif

  /* output the first line, inserting "ESMTP" as second word */
  expand(SmtpGreeting, inp, sizeof inp, e);
  p = strchr(inp, '\n');
  if (p != NULL)
    *p++ = '\0';
  id = strchr(inp, ' ');
  if (id == NULL)
    id = &inp[strlen(inp)];
  cmd = p == NULL ? "220 %.*s ESMTP%s" : "220-%.*s ESMTP%s";
  message(cmd, id - inp, inp, id);

  /* output remaining lines */
  while ((id = p) != NULL && (p = strchr(id, '\n')) != NULL)
  {
    *p++ = '\0';
    if (isascii(*id) && isspace(*id))
```

```
  if (!strcasecmp(c->cmdname, cmdbuf))
    break;
}

/* reset errors */
errno = 0;

/*
**  Process command.
**
**  If we are running as a null server, return 550
**  to everything.
*/

if (nullserver)
{
  switch (c->cmdcode)
  {
    case CMDQUIT:
    case CMDHELO:
    case CMDEHLO:
    case CMDNOOP:
    /* process normally */
    break;

    default:
    if (++badcommands > MAXBADCOMMANDS)
      sleep(1);
    usrerr("550 Access denied");
    continue;
  }
}

/* non-null server */
switch (c->cmdcode)
{
  case CMDMAIL:
  case CMDEXPN:
  case CMDVRFY:
```

```
      *p++ = '\0';
      vp = p;

      /* skip to the end of the value */
      while (*p != '\0' && *p != ' ' &&
          !(isascii(*p) && iscntrl(*p)) &&
          *p != '=')
        p++;
    }

    if (*p != '\0')
      *p++ = '\0';

    if (tTd(19, 1))
      printf("RCPT: got arg %s=\"%s\"\n", kp,
        vp == NULL ? "<null>" : vp);

    rcpt_esmtp_args(a, kp, vp, e);
    if (Errors > 0)
      break;
  }
  if (Errors > 0)
    break;

  /* save in recipient list after ESMTP mods */
  a = recipient(a, &e->e_sendqueue, 0, e);
  if (Errors > 0)
    break;

  /* no errors during parsing, but might be a duplicate */
  e->e_to = a->q_paddr;
  if (!bitset(QBADADDR, a->q_flags))
  {
    message("250 Recipient ok%s",
      bitset(QQUEUEUP, a->q_flags) ?
        " (will queue)" : "");
    nrcpts++;
  }
  else
  {
    /* punt -- should keep message in ADDRESS.... */
```

If You’re Worried about Security…

A **malicious adversary** is trying to exploit anything you miss!

What more can we do?

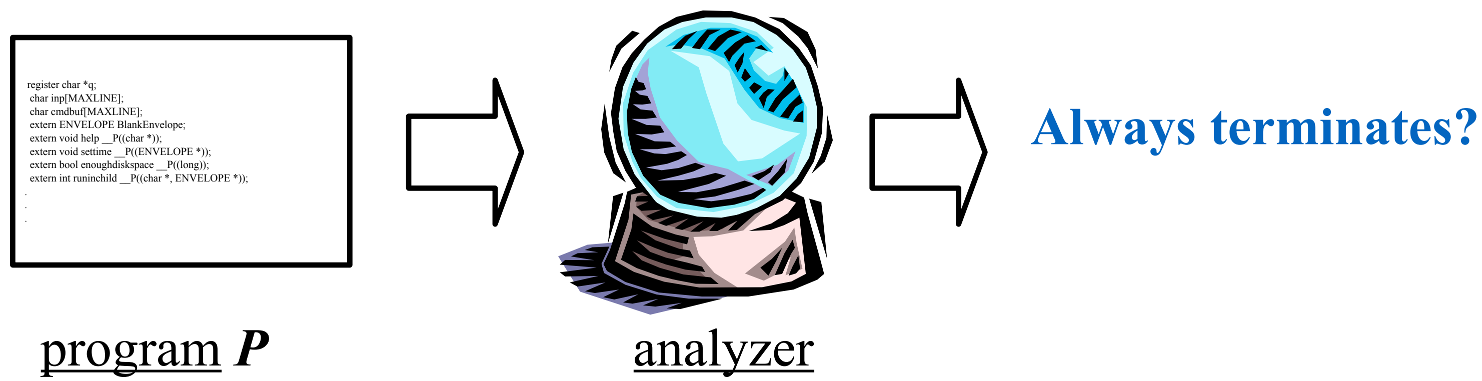
Static analysis

- Analyze program's code without running it
    - In a sense, ask a computer to do code review

- **Benefit:** (much) **higher coverage**
    - Reason about many possible runs of the program
        - Sometimes *all of them*, providing a **guarantee**
    - Reason about incomplete programs (e.g., libraries)

- **Drawbacks:**
    - Can only analyze limited properties
    - May miss some errors, or have false alarms
    - Can be time- and resource-consuming

The Halting Problem



- Can we write an analyzer that can prove, for any program *P* and inputs to it, *P* will terminate?
  - Doing so is called the **halting problem**
  - Unfortunately, this is **undecidable:** any analyzer will fail to produce an answer for at least some programs and/or inputs

Some material inspired by work of Matt Might: http://matt.might.net/articles/intro-static-analysis/

Check other properties instead?

- Perhaps security-related properties are feasible
  - E.g., that all accesses `a[i]` are in bounds
- *But* these **properties can be converted into the halting problem** by transforming the program
  - A perfect array bounds checker could solve the halting problem, which is impossible!
- Other undecidable properties (Rice's theorem)
  - Does this **SQL string** come from a **tainted source**?
  - Is this **pointer used after** its memory is **freed**?
  - Do any variables experience **data races**?

Halting ≈ Index in Bounds

- Change all exits to infinite loops (guaranteed no terminate)
- Change out-of-bounds index to exit:
  - (i >= 0 && i < a.length) ? a[i] : exit()
- Now if the array bounds checker
  - … **finds an error**, then the original program **halts**
  - … claims there are **no such errors**, then the original program **does not halt**
  - … **contradiction!** with halting undecidability

So is static analysis impossible?

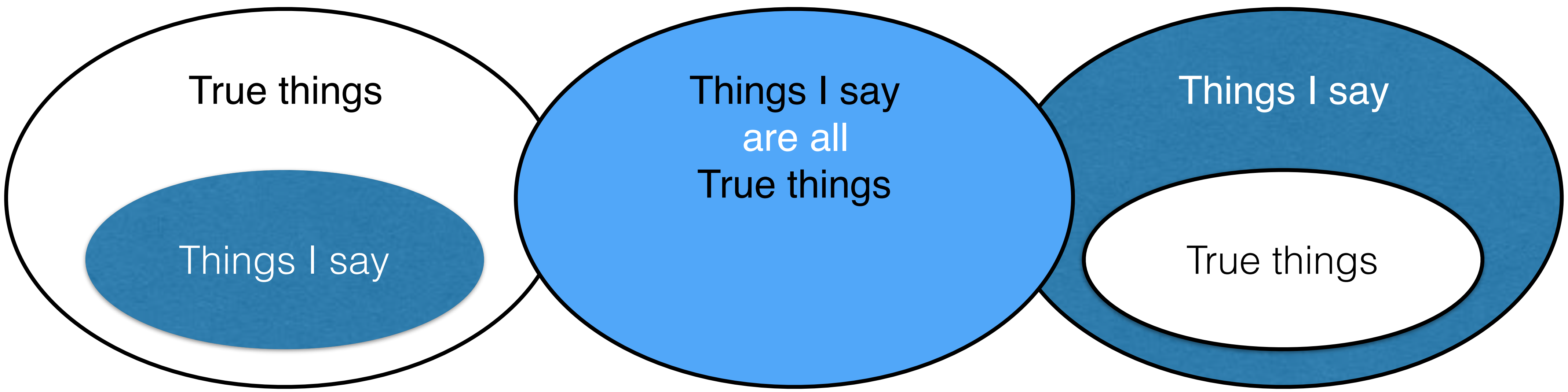
- **Perfect** static analysis is **not possible**
- **Useful** static analysis is **perfectly possible**, despite
  1. **Nontermination** - analyzer never terminates, or
  2. **False alarms** - claimed errors are not really errors, or
  3. **Missed errors** - no error reports ≠ error free
- Nonterminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors

Completeness

If analysis says that X is true, then X is true.

Soundness

If X is true, then analysis says X is true.



Trivially Complete: Say nothing

Trivially Sound: Say everything

Sound and **Complete**:
Say exactly the set of true things

Stepping back

- **Soundness**: No error found = no error exists
  - Alarms may be false errors
- **Completeness**: Any error found = real error
  - Silence does not guarantee no errors
- Basically any useful analysis
  - is neither **sound** nor **complete** (def. not **both**)
  - … usually *leans* one way or the other
    - Academic analyses lean towards **sound**

The Art of Static Analysis

- Design goals:
  - **Precision**: Carefully model program, minimize false positives/negatives
  - **Scalability**: Successfully analyze large programs
  - **Understandability**: Error reports should be actionable
- Observation: **Code style is important**
  - Aim to be precise for “good” programs
    - OK to forbid yucky code in the name of safety
    - Code that is more understandable to the analysis is more understandable to humans

First, a few words on different types of analyses…

Many Kinds of Analyses

- Constraint-Based
- Type-Based
- Abstract Interpretation
- Symbolic Execution
- Shape
- Pointer
- Dataflow
- Interprocedural

And many, many more!!!

All analyses have one thing in common:

AST

They define how to take each piece of the program and interpret it in some part of the analysis framework

This is what changes!

A few examples…

- Constraints
- Points in Lattice
- Sets of numbers / values

Tainted Flow Analysis

- Cause of many attacks is **trusting unvalidated input**
  - Input from the user (network, file) is **tainted**
  - Various data is used, assuming it is **untainted**

- Examples expecting untainted data
  - source string of `strcpy` (≤ target buffer size)
  - format string of `printf` (contains no format specifiers)
  - form field used in constructed SQL query (contains no SQL commands)

Recall: Format String Attack

- Adversary-controlled format string

```
char *name = fgets(…, network_fd);
printf(name);      // Oops
```

  - Attacker sets name = “`%s%s%s`” to crash program
  - Attacker sets name = “`…%n…`” to write to memory
    - Yields code injection exploits
- These bugs still occur in the wild occasionally
  - Too restrictive to forbid non-constant format strings

The problem, in types

- Specify our requirement as a *type qualifier*

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

- **tainted** = possibly controlled by adversary
- **untainted** = must not be controlled by adversary

```
tainted char *name = fgets(…,network_fd);
printf(name);    // FAIL: tainted ≠ untainted
```

Analyzing taint flows

- **Goal**: For all possible inputs, prove tainted data will never be used where untainted data is expected
  - **untainted** annotation: indicates a **trusted** **sink**
  - **tainted** annotation: an **untrusted** **source**
  - *no annotation* means: not sure (analysis must figure it out)
- Solution requires inferring **flows** in the program
  - What **sources can reach what sinks**
  - If any flows are *illegal*, i.e., whether a **tainted** source may flow to an **untainted** sink
- We will aim to develop a *sound* analysis

Legal Flow

```
void f(tainted int);
untainted int a = …;
f(a);
```

`f` accepts **tainted** *or* **untainted** data

untainted $\leq$ **tainted**

Illegal Flow

```
void g(untainted int);
tainted int b = …;
g(b);
```

`g` accepts *only* **untainted** data

tainted $\not\leq$ **untainted**

Define allowed flow as a **lattice:**

untainted < **tainted**

At each program step, **test** whether inputs $\leq$ policy

Analysis Approach

- If no qualifier is present, we must **infer** it
- Steps:
  - **Create** a **name** for each missing qualifier (e.g., **α, β**)
  - For each program statement, **generate constraints**
    - Statement `x = y` generates constraint $q_y \leq q_x$
  - **Solve the constraints** to produce solutions for **α, β,** etc.
    - A solution is a *substitution* of qualifiers (like **tainted** or **untainted**) for names (like **α** and **β**) such that all of the constraints are legal flows
- If there is **no solution**, we (may) have an **illegal flow**

Example Analysis

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
α char *name = fgets(…, network_fd);
β char *x = name;
printf(x);
```

1. **tainted** $\leq$ α
2. $\alpha \leq \beta$
3. $\beta \leq$ **untainted**

Illegal flow!

No possible solution for α and β

First constraint requires $\alpha =$ **tainted**
To satisfy the second constraint implies $\beta =$ **tainted**
But then the third constraint is illegal: **tainted ≤ untainted**

Taint Analysis: Adding ***Sensitivity***

But what about?

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
α char *name = fgets(…, network_fd);
β char *x;
x = name;
x = “hello!”;
printf(x);
```

$\textbf{tainted} \leq \alpha$

$\alpha \leq \beta$

$\textbf{untainted} \leq \beta$

$\beta \leq \textbf{untainted}$

No constraint solution. Bug?

False Alarm!

Flow Sensitivity

- Our analysis is **flow *in*sensitive**
  - Each variable has **one qualifier**
  - Conflates the taintedness of all values it ever contains
- **Flow-sensitive analysis** accounts for variables whose contents change
  - Allow each assigned use of a variable to have a different qualifier
    - E.g., α_1 is x's qualifier at line 1, but α_2 is the qualifier at line 2, where α_1 and α_2 can differ
  - Could implement this by transforming the program to assign to a variable at most once

Reworked Example

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
→ α char *name = fgets(…, network_fd);
char β *x1, γ *x2;
x1 = name;
x2 = "%s";
printf(x2);
```

tainted $\leq \alpha$

$\alpha \leq \beta$

untainted $\leq \gamma$

$\gamma \leq$ untainted

No Alarm

Good solution exists:

γ = untainted

$\alpha = \beta$ = tainted

Handling conditionals

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
α char *name = fgets(…, network_fd);
β char *x;
if (…)  x = name;
else    x = “hello!”;
printf(x);
```

tainted $\leq$ α

α $\leq$ β

~~untainted $\leq$ β~~

β $\leq$ untainted

Constraints still unsolvable

Illegal flow

Multiple Conditionals

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
void f(int x) {
   α char *y;
   if (x) y = “hello!”;
   else   y = fgets(…, network_fd);
   if (x) printf(y);
}
```

~~untainted ≤ α~~

tainted ≤ α

α ≤ untainted

No solution for **α**. Bug?

False Alarm!

(and flow sensitivity won’t help)

Path Sensitivity

- Consider *path feasibility*. E.g., **`f(x)`** can execute path
  - **1-2-4-5-6** when **`x`** $\neq$ `0`, or
  - **1-3-4-6** when **`x`** `==` `0`. But,
  - path **1-3-4-5-6** *infeasible*

```
void f(int x) {
  char *y;
  1if (x) 2y = “hello!”;
   else   3y = fgets(…);
  4if (x) 5printf(y);
6}
```

- A **path sensitive analysis** checks feasibility, e.g., by qualifying each constraint with a **path condition**
  - `x` $\neq 0 \Longrightarrow$ **untainted** $\leq$ **α** (segment 1-2)
  - `x` $= 0 \Longrightarrow$ **tainted** $\leq$ **α** (segment 1-3)
  - `x` $\neq 0 \Longrightarrow$ **α** $\leq$ **untainted** (segment 4-5)

Why *not* use flow/path sensitivity?

- Flow sensitivity **adds precision**, path sensitivity adds more
  - Reduce false positives: less developer effort!
- But both of these **make solving more difficult**
  - Flow sensitivity *increases the number of nodes* in the constraint graph
  - Path sensitivity *requires more general solving procedures* to handle path conditions
- In short: **precision** (often) **trades off scalability**
  - Ultimately, limits the size of programs we can analyze

Handling Function Calls

```
α char *a = fgets(…);
β char *b = id(a);
```

```
δ char *id(γ char *x) {
  return x;
}
```

- Names for arguments and return value
- Calls create flows
  - from **caller's data** to **callee's arguments**,
  - from **callee's result** to **caller's returned value**

Handling Function Calls

```
α char *a = fgets(…);
β char *b = id(a);
```

```
δ char *id(γ char *x) {
    return x;
}
```

tainted ≤ α

$\alpha \leq \gamma$

$\gamma \leq \delta$

$\delta \leq \beta$

Result: b is tainted (as expected)

Function Call Example

```
α char *a = fgets(…);
β char *b = id(a);
ω char *c = "hi";
printf(c);
```

```
δ char *id(γ char *x) {
  return x;
}
```

$$\text{tainted} \le \alpha$$
$$\alpha \le \gamma$$
$$\gamma \le \delta$$
$$\delta \le \beta$$
$$\text{untainted} \le \omega$$
$$\omega \le \text{untainted}$$

No Alarm

Good solution exists:

ω = **untainted**

$\alpha = \beta = \gamma = \delta$ = **tainted**

Two Calls to Same Function

```
α char *a = fgets(…);
β char *b = id(a);
ω char *c = id(“hi”);
printf(c);
```

```
δ char *id(γ char *x) {
    return x;
}
```

tainted $\leq \alpha$

$\alpha \leq \gamma$

$\gamma \leq \delta$

$\delta \leq \beta$

untainted $\leq \gamma$

$\delta \leq \omega$

$\omega \leq$ untainted

No solution. Real bug?

False Alarm!

Two Calls to Same Function

```
α char *a = fgets(…);
β char *b = id(a);
ω char *c = id("hi");
printf(c);
```

```
δ char *id(γ char *x) {
    return x;
}
```

tainted ≤ α ≤ γ ≤ δ ≤ ω ≤ untainted

Problematic constraints represent an infeasible path

False Alarm!

Context (In)sensitivity

- This is a problem of **context insensitivity**
  - All call sites are “conflated” in the graph
- **Context sensitivity** solves this problem by:
  - **Labeling call sites** in some way (e.g. line number)
  - **Matching calls** with the corresponding **returns**
    - Label call and return edges
    - Allow flows if the labels match

Two Calls to Same Function

```
α char *a = fgets(…);
β char *b = id1(a);
ω char *c = id2("hi");
printf(c);
```

```
δ char *id(γ char *x) {
    return x;
}
```

$\text{tainted} \leq \alpha$

$\alpha \leq_1 \gamma$

$\gamma \leq \delta$

$\cancel{\delta \leq_1 \beta}$

$\cancel{\text{untainted} \leq_2 \gamma}$

$\delta \leq_2 \omega$

$\omega \leq \text{untainted}$

Indexes don't match up

Infeasible flow not allowed

No Alarm

Discussion

- **Context sensitivity:** another precision/scalability **tradeoff**
  - *O(n)* insensitive algorithm becomes *O(n^3)* sensitive algorithm
  - But: Eliminates infeasible paths (makes *n* smaller)
  - Sometimes *higher precision improves performance*
- Compromises possible
  - Only *some* call sites treated sensitively
    - Conflate *groups* of call sites
    - Sensitivity only up to a ***certain call depth***

Flow Analysis:
Scaling it up
to a complete
language and
problem set

Pointers

```
α char *a = “hi”;
(β char *)*p = &a;
(γ char *)*q = p;
ω char *b = fgets(…);
*q = b;
printf(*p);
```

untainted $\leq$ α

$\alpha \leq \beta$

$\beta \leq \gamma$

tainted $\leq$ ω

$\omega \leq \gamma$

$\beta \leq$ untainted

Solution exists:

$\alpha = \beta =$ **untainted**

$\omega = \gamma =$ **tainted**

Misses illegal flow!

- `p` and `q` are aliases
  - so writing **tainted** data to `q`
  - makes `p`'s contents **tainted**

Pointers

```
α char *a = “hi”;
(β char *)*p = &a;
(γ char *)*q = p;
ω char *b = fgets(…);
*q = b;
printf(*p);
```

~~Solution exists:~~

$\alpha = \beta =$ untainted

$\omega = \gamma =$ tainted

untainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \gamma$

tainted $\leq \omega$

$\gamma \leq \beta$

$\omega \leq \gamma$

$\beta \leq$ untainted

Pointers

```
α char *a = “hi”;
(β char *)*p = &a;
(γ char *)*q = p;
ω char *b = fgets(…);
*q = b;
printf(*p);
```

~~Solution exists:~~

$\alpha = \beta =$ untainted

$\omega = \gamma =$ tainted

$\text{untainted} \le \alpha$

$\alpha \le \beta$

$\beta \le \gamma$

$\gamma \le \beta$

$\text{tainted} \le \omega$

$\omega \le \gamma$

$\beta \le \text{untainted}$

Flow and pointers

- An assignment via a pointer "flows both ways"
  - Ensures that **aliasing constraints are sound**
  - But can lead to **false alarms**

- Reducing alarms
  - If pointers are never assigned to (`const`) then backward flow is not needed (sound)
  - Drop backward flow edge anyway
    - Trades false alarms for missed errors (unsoundness)

Implicit flows

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
  untainted int i;
  for (i = 0; i<len; i++) {
    dst[i] = src[i]; //illegal
  }
}
```

Illegal flow :

tainted $\not\leq$ untainted

Implicit flows

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
  untainted int i, j;
  for (i = 0; i<len; i++) {
    for (j = 0; j<sizeof(char)*256; j++) {
      if (src[i] == (char)j)
        dst[i] = (char)j;     //legal?
    }
  }   untainted char     untainted char
}
```

Missed flow !

Implicit flow analysis

- **Implicit flow:** one value *implicitly* influences another
- One way to find these: maintain a scoped **program counter (*pc*) label**
  - Represents the maximum taint affecting the current pc
- Assignments generate constraints involving the *pc*
  - `x = y` produces two constraints:

    $\textit{label}(\mathtt{y}) \leq \textit{label}(\mathtt{x})$ (as usual)

    $\textit{pc} \leq \textit{label}(\mathtt{x})$

Implicit flow example

```
tainted int src;
α int dst;
if (src == 0)
 dst = 0;
else
 dst = 1;

dst += 0;
```

pc_1 = **untainted**

pc_2 = **tainted**

pc_3 = **tainted**

pc_4 = **untainted**

untainted $\leq$ **α**
$pc_2 \leq$ **α**

untainted $\leq$ **α**
$pc_3 \leq$ **α**

untainted $\leq$ **α**
$pc_4 \leq$ **α**

: **tainted** $\leq$ **α**

Taint on α is identified.

Discovers implicit flow!

Why not implicit flow?

- Tracking implicit flows can lead to **false alarms**
  - E.g., ignores values

```
tainted int src;
α int dst;
if (src > 0) dst = 0;
else         dst = 0;
```

- Extra constraints **hurt performance**
- The evil copying example is *pathological*
  - We typically don't write programs like this*
  - Implicit flows will have little overall influence
- *So*: **taint analyses tend to ignore implicit flows**

* Exception coming in two slides

Other challenges

- Taint through operations
  - `tainted a; untainted b; c=a+b` — is `c` tainted? (yes, probably)
- Function calls and context sensitivity
  - Function pointers: Flow analysis to compute possible targets
- Struct fields
  - Track taint for the whole struct, or each field?
  - Taint per instance, or shared among all of them (or something in between)?
    - Note: objects ≈ structs + function pointers
- Arrays: Track taint per element or across whole array?

No single correct answer!
(Tradeoffs: Soundness, completeness, performance)

Other refinements

- Label *additional* sources and sinks
  - e.g., Array accesses must have untainted index
- Handle ***sanitizer functions***
  - Convert tainted data to untainted
- Complementary goal: Leaking confidential data
  - Don’t want **secret sources** to go to **public sinks**
    - Implicit flows more relevant (malicious code)
  - *Dual* of tainting

Other kinds of analysis

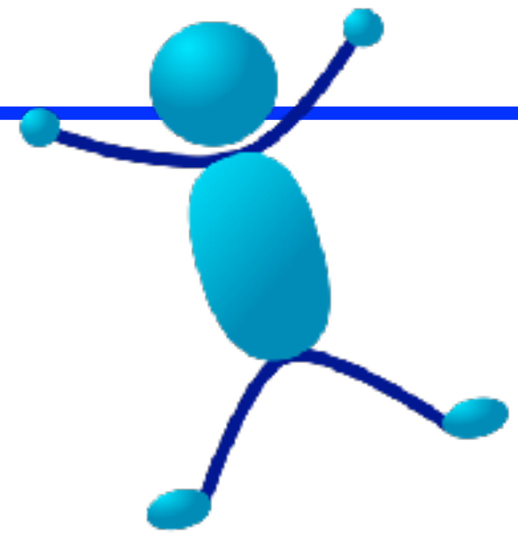
- **Pointer Analysis** (“points-to” analysis)
  - Determine whether pointers point to the same locations
  - Shares many elements of flow analysis. Really advanced in the last 10 years.

- **Data Flow Analysis**
  - Invented in the early 1970’s. Flow sensitive, tracks “data flow facts” about variables in the program

- **Abstract interpretation**
  - Invented in the late 1970’s as a theoretical foundation for data flow analysis, and static analysis generally.
  - Associated with certain analysis algorithms

Symbolic Execution

Introduction



- Static analysis is great
  - Lots of interesting ideas and tools
  - Commercial companies sell, use static analysis
  - It all looks good on paper, and in papers

- But can developers use it?
  - Our experience: Not easily
  - Results in papers describe use by static analysis experts
  - Commercial tools have a huge code mass to deal with developer confusion, false positives, warning management, etc

One Issue: Abstraction

- Abstraction lets us scale and model all possible runs
  - But it also introduces conservatism
  - *-sensitivities attempt to deal with this
    - * = flow-, context-, path-, field-, etc
  - But they are never enough

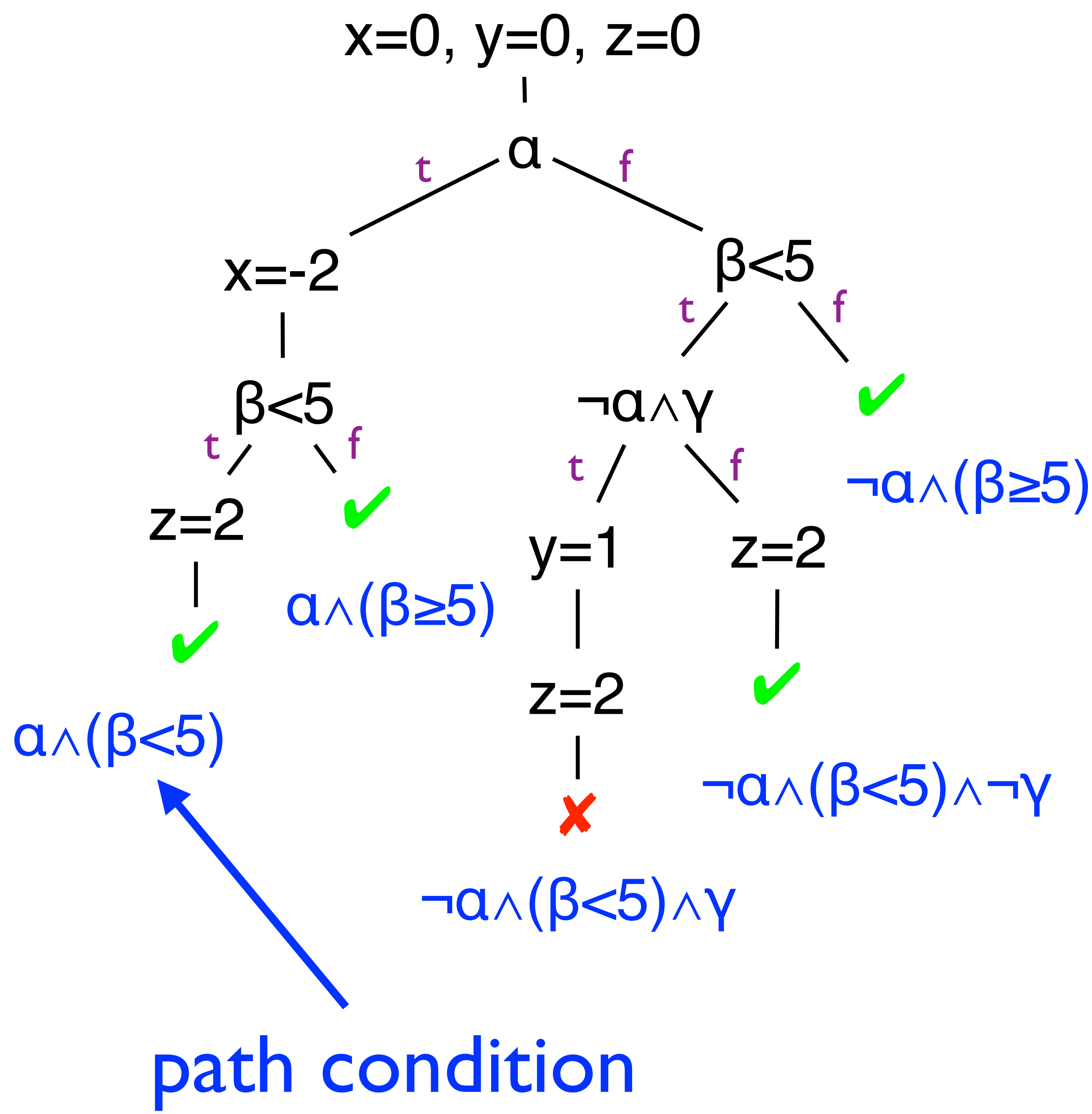
- Static analysis abstraction $\neq$ developer abstraction
  - Because the developer didn't have them in mind

Symbolic Execution

- Testing works
  - But, each test only explores one possible execution
    - assert(f(3) == 5)
  - We *hope* test cases generalize, but no guarantees
- Symbolic execution generalizes testing
  - Allows *unknown* symbolic variables in evaluation
    - y = α; assert(f(y) == 2*y-1);
  - If execution path depends on unknown, conceptually *fork* symbolic executor
    - int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }

Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Insight

- Each symbolic execution path stands for *many* actually program runs
  - In fact, exactly the set of runs whose concrete values satisfy the path condition
- Thus, we can cover a lot more of the program's execution space than testing can

Early work on symbolic execution

- Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In ICRS, pages 234–245, 1975.
- James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. **(most cited)**
- Leon J. Osterweil and Lloyd D. Fosdick. Program testing techniques using simulated execution. In ANSS, pages 171–177, 1976.
- William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4):266–278, 1977.

The problem

- Computers were small (not much memory) and slow (not much processing power) then
  - Apple's iPad 2 is as fast as a Cray-2 from the 1980's

- Symbolic execution is potentially extremely expensive
  - Lots of possible program paths
  - Need to query solver a lot to decide which paths are feasible, which assertions could be false
  - Program state has many bits

Today

- Computers are much faster, memory is cheap
- There are very powerful SMT/SAT solvers today
  - SMT = Satisfiability Modulo Theories = SAT++
  - Can solve very large instances, very quickly
    - Lets us check assertions, prune infeasible paths
  - We've used Z3, STP, and Yices
- Recent success: bug finding
  - Heuristic search through space of possible executions
  - Find really interesting bugs

Path explosion

- Usually can’t run symbolic execution to exhaustion
  - Exponential in branching structure

```
int a = α, b = β, c = γ;    // symbolic
if (a) ... else ...;
if (b) ... else ...;
if (c) ... else ...;
```

    - Ex: 3 variables, 8 program paths
  - Loops on symbolic variables even worse

```
int a = α;    // symbolic
while (a) do ...;
```

    - Potentially 2^31 paths through loop!

Search strategies

- Need to prioritize search
  - Try to steer search towards paths more likely to contain assertion failures
  - Only run for a certain length of time
    - So if we don't find a bug/vulnerability within time budget, too bad
- Think of program execution as a dag
  - Nodes = program states
  - Edge(n1,n2) = can transition from state n1 to state n2
- Then we need some kind of graph exploration strategy
  - At each step, pick among all possible paths

Basic search

- Simplest ideas: algorithms 101
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - Which of these did we implement?

- Potential drawbacks
  - Neither is guided by any higher-level knowledge
    - Probably a bad sign
  - DFS could easily get stuck in one part of the program
    - E.g., it could keep going around a loop over and over again
  - Of these two, BFS is a better choice

Randomness

- We don’t know a priori which paths to take, so adding some randomness seems like a good idea
  - Idea 1: pick next path to explore uniformly at random (Random Path, RP)
  - Idea 2: randomly restart search if haven’t hit anything interesting in a while
  - Idea 3: when have equal priority paths to explore, choose next one at random
    - All of these are good ideas, and randomness is very effective
- One drawback: reproducibility
  - Probably good to use psuedo-randomness based on seed, and then record which seed is picked
  - (More important for symbolic execution implementers than users)

Coverage-guided heuristics

- Idea: Try to visit statements we haven't seen before
- Approach
  - Score of statement = # times it's been seen and how often
  - Pick next statement to explore that has lowest score
- Why might this work?
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.
- Why might this not work?
  - Maybe never be able to get to a statement if proper precondition not set up
- KLEE = RP + coverage-guided

Generational search

- Hybrid of BFS and coverage-guided
- Generation 0: pick one program at random, run to completion
- Generation 1: take paths from gen 0, negate *one* branch condition on a path to yield a new path prefix, find a solution for that path prefix, and then take the resulting path
  - Note will semi-randomly assign to any variables not constrained by the path prefix
- Generation n: similar, but branching off gen n-1
- Also uses a coverage heuristic to pick priority

Combined search

- Run multiple searches at the same time
- Alternate between them
  - E.g., Fitnext

- Idea: no one-size-fits-all solution
  - Depends on conditions needed to exhibit bug
  - So will be as good as “best” solution, which a constant factor for wasting time with other algorithms
  - Could potentially use different algorithms to reach different parts of the program

SMT solver performance

- SAT solvers are at core of SMT solvers
  - In theory, could reduce all SMT queries to SAT queries
  - In practice, SMT and higher-level optimizations are critical

- Some examples
  - Simple identities (x + 0 = x, x * 0 = 0)
  - Theory of arrays (read(42, write(42, x, A)) = x)
    - 42 = array index, A = array, x = element
  - Caching (memoize solver queries)
  - Remove useless variables
    - E.g., if trying to show path feasible, only the part of the path condition related to variables in guard are important

Libraries and native code

- At some point, symbolic execution will reach the “edges” of the application
  - Library, system, or assembly code calls
- In some cases, could pull in that code also
  - E.g., pull in libc and symbolically execute it
  - But glibc is really complicated
    - Symbolic execution can easily get stuck in it
  - ⇒ pull in a simpler version of libc, e.g., newlib
    - libc versions for embedded systems tend to be simpler
- In other cases, need to make models of code
  - E.g., implement ramdisk to model kernel fs code
  - This is a lot of work!

Concolic execution

- Also called *dynamic symbolic execution*
- Instrument the program to do symbolic execution as the program runs
  - I.e., shadow concrete program state with symbolic variables
- Explore one path, from start to completion, at a time
  - Thus, always have a concrete underlying value to rely on

Concretization

- Concolic execution makes it really easy to concretize
  - Replace symbolic variables with concrete values that satisfy the path condition
    - Always have these around in concolic execution
- So, could actually do system calls
  - But we lose symbolic-ness at such calls
- And can handle cases when conditions too complex for SMT solver
  - But can do the same in pure symbolic system

Resurgence of symbolic exection

- Two key systems that triggered revival of this topic:
  - DART — Godefroid and Sen, PLDI 2005
    - Godefroid = model checking, formal systems background
  - EXE — Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006
    - Ganesh and Dill = SMT solver called “STP” (used in implementation)
      - Theory of arrays
    - Cadar and Engler = systems

KLEE: Coreutils crashes

`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`pr -e t2.txt`
`tac -r t3.txt t3.txt`
`mkdir -Z a b`
`mkfifo -Z a b`
`mknod -Z a b p`
`md5sum -c t1.txt`
`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`ptx x t4.txt`
`seq -f %0 1`

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

Static analysis in practice

- Thoroughly check limited but useful properties
  - **Eliminate** some categories of errors
  - Developers can concentrate on **deeper reasoning**
- Encourage **better development practices**
  - Programming models that avoid mistakes
  - Teach programmers to manifest their assumptions
    - Using **annotations** that improve tool precision
- Seeing **increased commercial adoption**