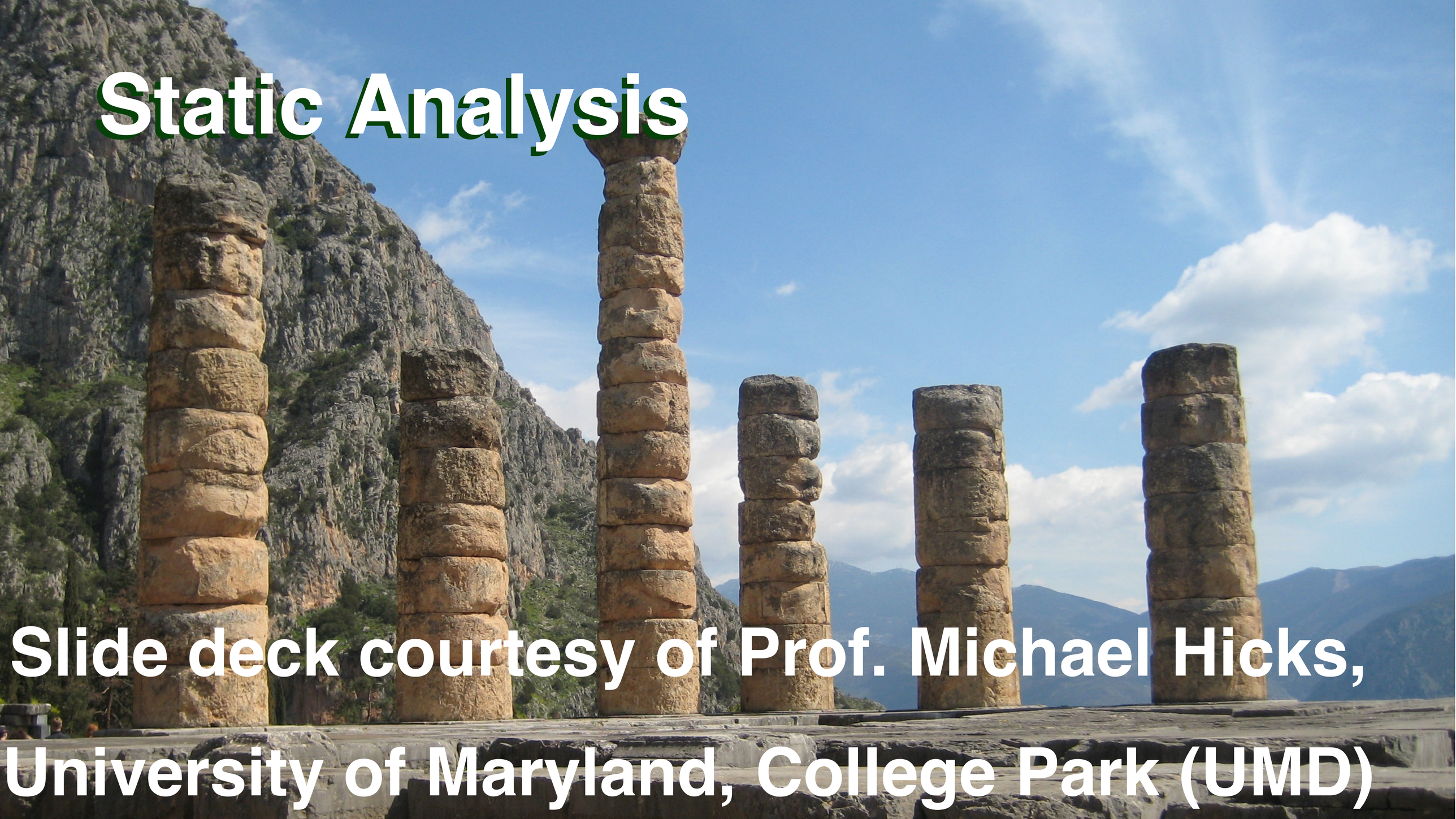


Static Analysis

Slide deck courtesy of Prof. Michael Hicks, University of Maryland, College Park (UMD)

Static Analysis

for **Secure Development**

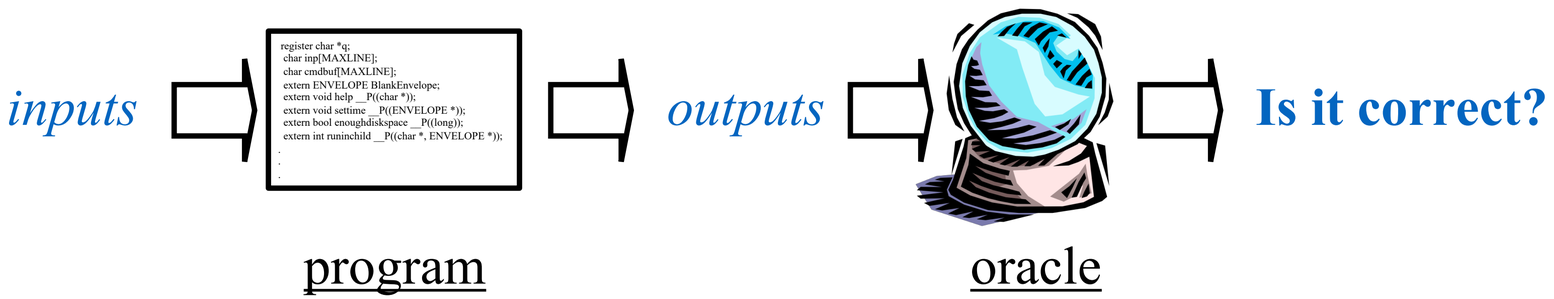
- **Introduction**
  - *Static analysis*: **What**, and **why**?
- **Basic analysis**
  - *Example*: **Flow analysis**
- **Increasing precision**
  - *Context*-, *flow*-, and *path* **sensitivity**
- **Scaling it up**
  - Pointers, arrays, information flow, …

Non Examinable (Scaling it up)

Current Practice

for Software Assurance

- **Testing**
  - Make sure program runs correctly on set of inputs



  - **Benefits**: Concrete failure proves issue, aids in fix
  - **Drawbacks**: Expensive, difficult, ***hard to cover all code paths***, no guarantees

Current Practice
(cont'd)

- **Code Auditing**
  - Convince someone else your source code is correct
  - **Benefit**: humans can *generalize beyond single runs*
  - **Drawbacks**: Expensive, hard, no guarantees


```
register char *q;
 char inp[MAXLINE];
 char cmdbuf[MAXLINE];
 extern ENVELOPE BlankEnvelope;
 extern void help __P((char *));
 extern void settime __P((ENVELOPE *));
 extern bool enoughdiskspace __P((long));
 extern int runinchild __P((char *, ENVELOPE *));
 extern void checksmtpattack __P((volatile int *, int, char *, ENVELOPE *));

 if (fileno(OutChannel) != fileno(stdout))
 {
  /* 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))
```

```
cmd < &cmdbuf[sizeof cmdbuf - 2])
    *cmd++ = *p++;
 *cmd = '\0';

 /* throw away leading whitespace */
 while (isascii(*p) && isspace(*p))
  p++;

 /* decode command */
 for (c = CmdTab; c->cmdname != NULL; c++)
 {
  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:
```

```
while (isascii(*p) && isspace(*p))
    p++;
  if (*p == '\0')
   break;
  kp = p;

  /* skip to the value portion */
  while ((isascii(*p) && isalnum(*p)) || *p == '-')
   p++;
  if (*p == '=')
  {
   *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, we are asking a computer to do what a human might do during a code review
- **Benefit** is (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 consuming to run

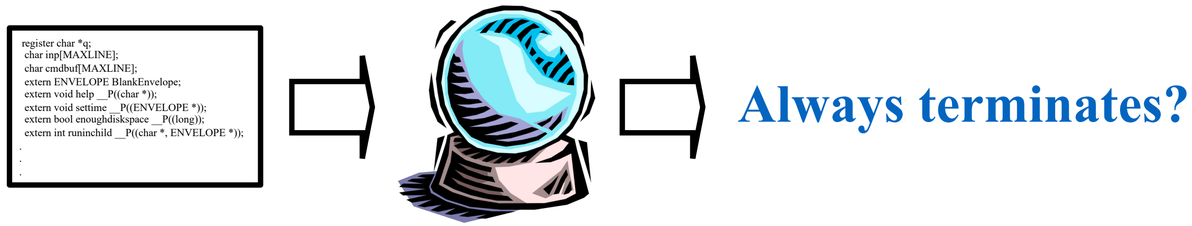
Impact

- Thoroughly check limited but useful properties
  - **Eliminate categories of errors**
  - **Developers** can **concentrate** on deeper **reasoning**
- **Encourages better development practices**
  - Develop programming models that avoid mistakes in the first place
  - Encourage programmers to think about and make manifest their assumptions
    - Using **annotations** that improve tool precision
- Seeing **increased commercial adoption**

What is **Static Analysis?**

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, the halting problem is **undecidable**
  - That is, it is **impossible** to write such an analyzer: it will fail to produce an answer for at least some programs (and/or some inputs)

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

Other properties?

- 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
  - I.e., 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

- Proof by transformation
  - Change indexing expressions `a[i]` to exit
    - `(i >= 0 && i < a.length) ? a[i] : exit()`
    - Now all array bounds errors instead result in termination
  - Change program exit points to out-of-bounds accesses
    - `a[a.length+10]`

- 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 undecidability of the halting problem

Static analysis is impossible?

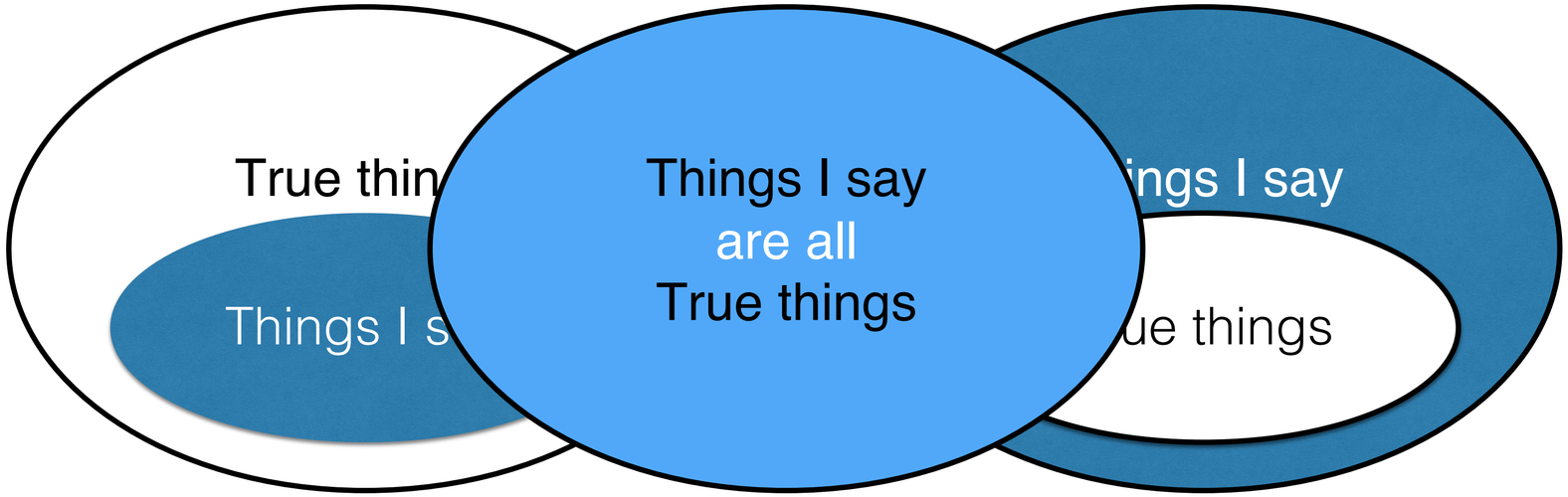
- **Perfect** static analysis is **not possible**
- **Useful** static analysis is **perfectly possible**, despite
  1. **Nontermination** - analyzer never terminates, or
  2. **False alarms** - claimed errors are not really errors, or
  3. **Missed errors** - no error reports ≠ error free
- Nonterminating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

Soundness

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

Completeness

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



Trivially Sound: Say nothing

Sound

and Trivially Complete: Say everything

Complete

Say exactly the set of true things

Stepping back

- **Soundness**: if the program is claimed to be error free, then it really is
  - *Alarms do not imply erroneousness*
- **Completeness**: if the program is claimed to be erroneous, then it really is
  - *Silence does not imply error freedom*
- Essentially, most interesting analyses
  - are neither **sound** nor **complete** (and not **both**)
  - … usually ***lean*** toward soundness (“soundy”) or completeness

The Art of Static Analysis

- Analysis design tradeoffs
  - **Precision**: Carefully model program behavior, to minimize false alarms
  - **Scalability**: Successfully analyze large programs
  - **Understandability**: Error reports should be actionable
- Observation: **Code style is important**
  - Aim to be precise for “good” programs
    - It’s OK to forbid yucky code in the name of safety
  - False alarms viewed positively: reduces complexity
    - Code that is more understandable to the analysis is more understandable to humans

Flow Analysis

Tainted Flow Analysis

- The **root cause** of many attacks is **trusting unvalidated input**
  - Input from the user 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
  - 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
```

Analysis problem

- **No tainted data flows**: For all possible inputs, prove that 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 figures it out)
- A 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**

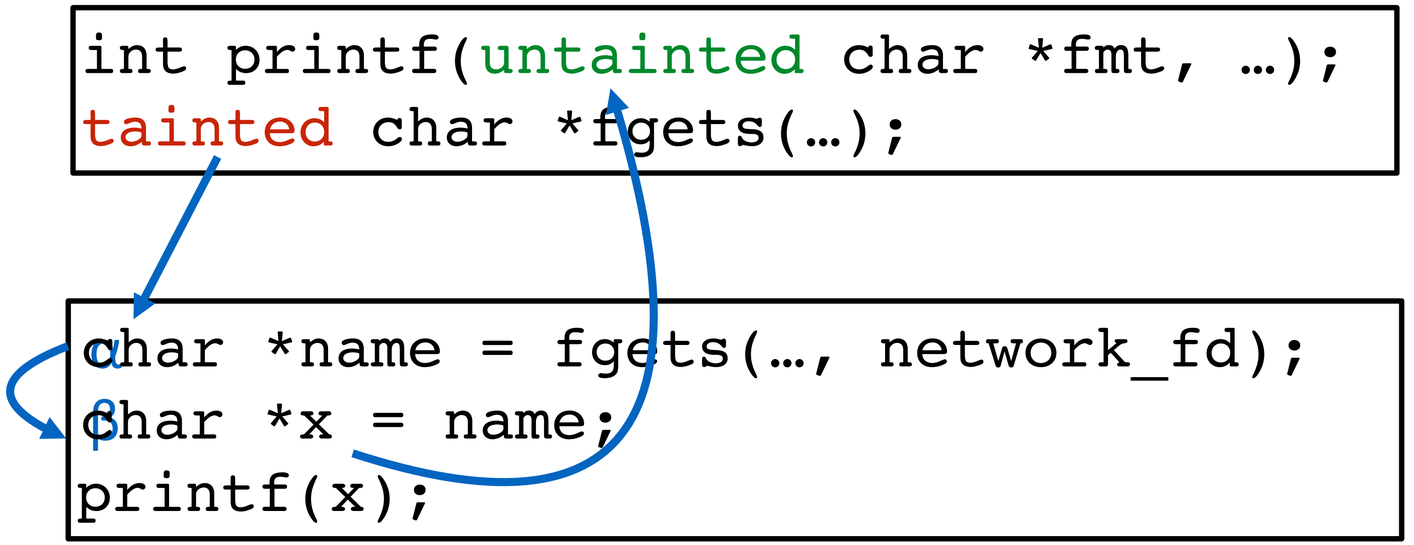
Allowed flow as a **lattice**

untainted < **tainted**

Analysis Approach

- Think of **flow analysis** as a kind of **type inference**
  - If no qualifier is present, we must infer it
- Steps:
  - **Create** a **name** for each missing qualifier (e.g., α, β)
  - For each statement in the program, **generate constraints** (of the form $q_1 \leq q_2$) on possible solutions
    - Statement `x = y` generates constraint qy ≤ qx where qy is `y`'s qualifier and qx is `x`'s qualifier
  - **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



tainted ≤ α

α ≤ β

β ≤ **untainted**

Illegal flow!

First constraint requires α = **tainted**

To satisfy the second constraint implies β = **tainted**

But then the third constraint is illegal: **tainted ≤ untainted**

No possible solution for α and β

Flow Analysis: Adding ***Sensitivity***

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 ≤ α

α ≤ β

untainted ≤ β

β ≤ **untainted**

Constraints still unsolvable

Illegal flow

Dropping the Conditional

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

→

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

tainted ≤ α

α ≤ β

untainted ≤ β

β ≤ **untainted**

Same constraints, different semantics!

False Alarm

Flow Sensitivity

- Our analysis is **flow *in*sensitive**
  - Each variable has **one qualifier** which abstracts the taintedness of all values it ever contains

- A **flow sensitive analysis** would account 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
    - Called **static single assignment (SSA)** form

Reworked Example

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

→

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

tainted ≤ α

α ≤ β

untainted ≤ γ

γ ≤ **untainted**

No Alarm

Good solution exists:

γ = **untainted**

α = β = **tainted**

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**

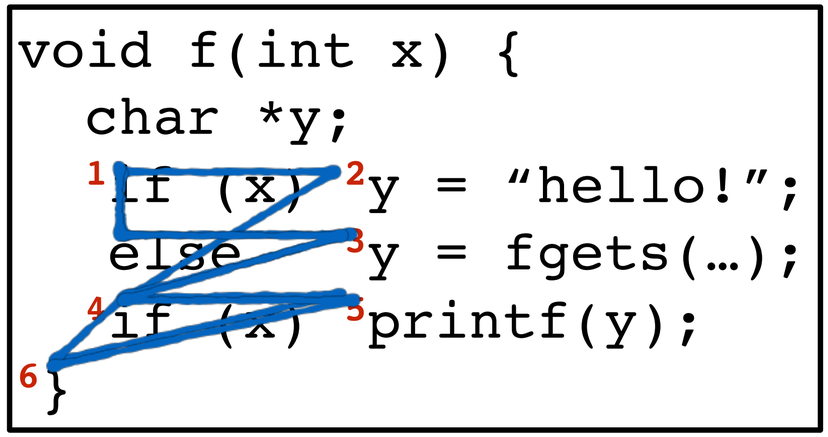
no solution for α

False Alarm!

(and flow sensitivity won’t help)

Path Sensitivity

- An analysis may consider *path feasibility*. E.g., `f(x)` can execute path
  - **1-2-4-5-6** when `x` is *not* `0`, or
  - **1-3-4-6** when `x` is `0`. But,
  - path **1-3-4-5-6** *infeasible*



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

Why *not* flow/path sensitivity?

- Flow sensitivity **adds precision**, and path sensitivity adds even more, which is *good*
- But both of these **make solving more difficult**
  - Flow sensitivity also *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

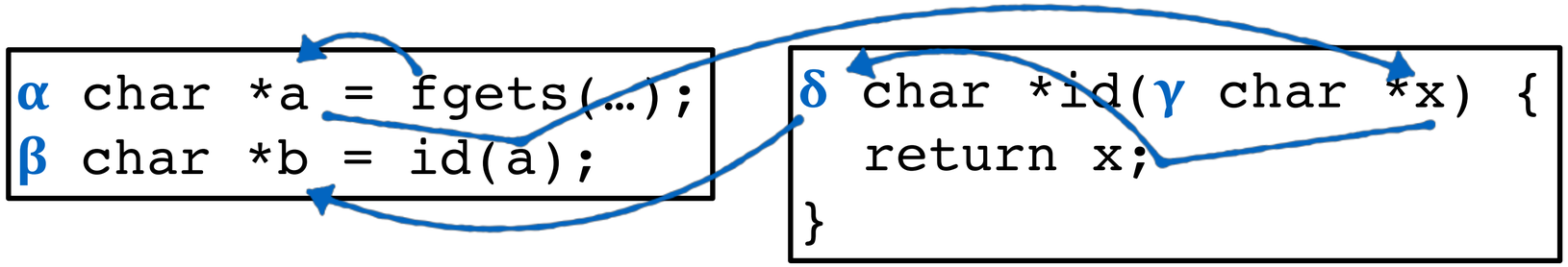
Handling Function Calls

```
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



tainted ≤ α

α ≤ γ

γ ≤ δ

δ ≤ **β**

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;
}
```

tainted ≤ α

α ≤ γ

γ ≤ δ

δ ≤ **β**

untainted ≤ ω

ω ≤ **untainted**

No Alarm

Good solution exists:

ω = **untainted**

α = β = γ = δ = **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 ≤ α

α ≤ γ

γ ≤ δ

δ ≤ **β**

untainted ≤ γ

δ ≤ ω

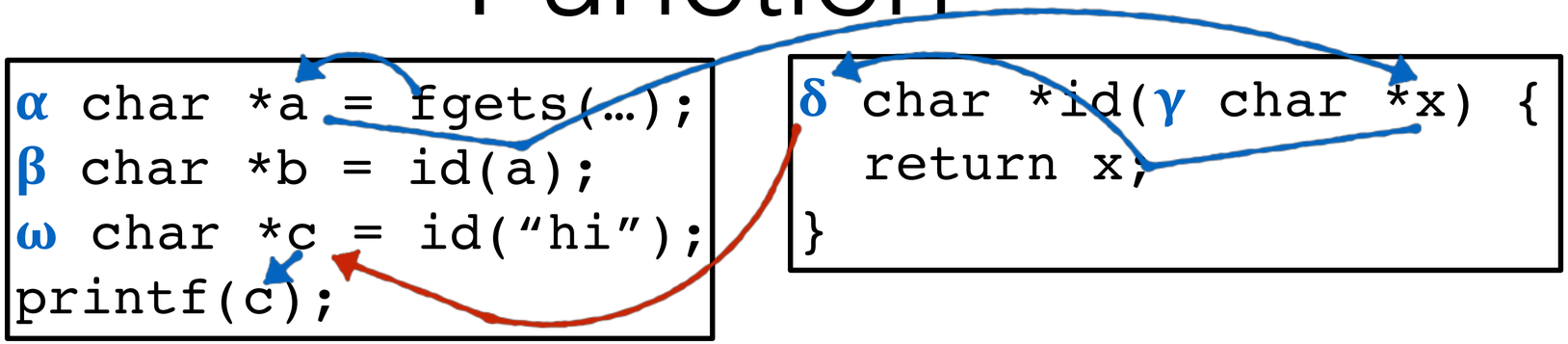
ω ≤ **untainted**

False Alarm!

No solution, and yet

no true tainted flow

Two Calls to Same Function



tainted ≤ α ≤ γ ≤ δ ≤ ω ≤ **untainted**

Problematic constraints represent an infeasible path

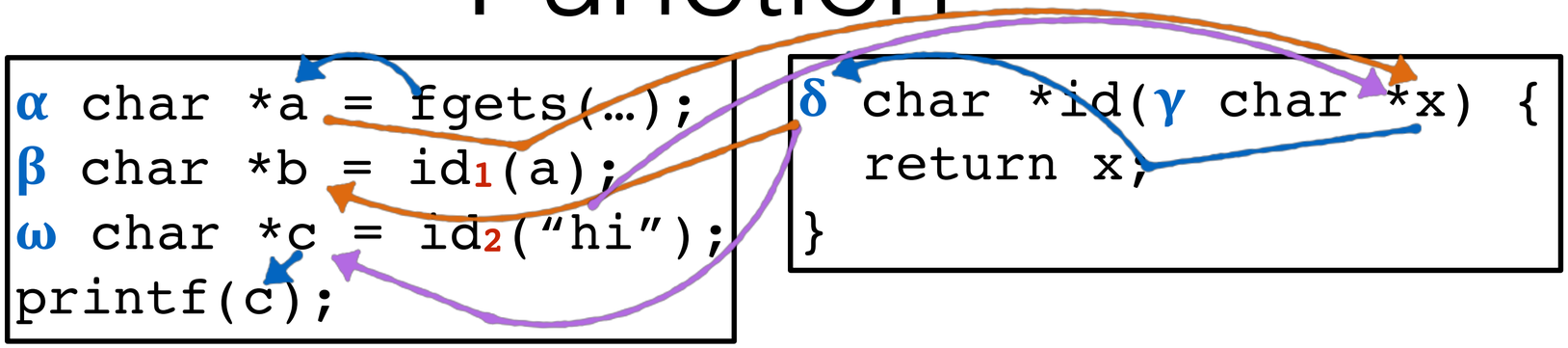
False Alarm!

No solution, and yet no true tainted flow

Context (In)sensitivity

- This is a problem of **context insensitivity**
  - All call sites are “conflated” in the graph
- **Context sensitivity** solves this problem by
  - **distinguishing call sites** in some way
    - We can give them a label ***i***, e.g., the line number in the program
  - **matching** up **calls** with the corresponding **returns**
    - Label call and return edges
    - Allow flows if the labels and **polarities** match
    - Use index **-i** for **argument passing**, i.e., $q_1 \leq_{-i} q_2$
    - Use index **+i** for **returned values**, i.e., $q_1 \leq_{+i} q_2$

Two Calls to Same Function



tainted ≤ α

α ≤**-1** γ

γ ≤ δ

δ ≤**+1** β

untainted ≤**-2** γ

δ ≤**+2** ω

ω ≤ **untainted**

Indexes don’t match up

Infeasible flow not allowed

No Alarm

Discussion

- **Context sensitivity** is a **tradeoff** again
  - *Precision* vs. *scalability*
    - *O(n)* insensitive algorithm becomes $O(n^3)$ sensitive algorithm
  - But: sometimes *higher precision improves performance*
    - Eliminates infeasible paths from consideration (makes *n* smaller)
- Compromises possible
  - Only ***some* call sites treated sensitively**
    - Rest conflated
  - **Conflate *groups* of call sites**
    - Give them the same index
  - **Sensitivity only *up to a certain call depth***
    - Don’t do exact matching of edges beyond that depth

Non Examinable

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

Non Examinable

Pointers

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

Solution exists:

α = β = **untainted**

ω = γ = **tainted**

untainted ≤ α

α ≤ β

β ≤ γ

tainted ≤ ω

ω ≤ γ

β ≤ **untainted**

Misses illegal flow!

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

Non Examinable

Pointers

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

~~Solution exists:~~

α = β = **untainted**

ω = γ = **tainted**

untainted ≤ α

α ≤ β

β ≤ γ

tainted ≤ γ

ω ≤ γ

β ≤ **untainted**

Non Examinable

Pointers

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

~~Solution exists:~~

α = β = **untainted**

ω = γ = **tainted**

untainted ≤ α

α ≤ β

β ≤ γ

γ ≤ β

tainted ≤ ω

ω ≤ γ

β ≤ **untainted**

Non Examinable

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)

Non Examinable

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
untainted char    tainted char
}
```

Illegal flow :

tainted ≠ untainted

Non Examinable

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

Non Examinable

Information flow analysis

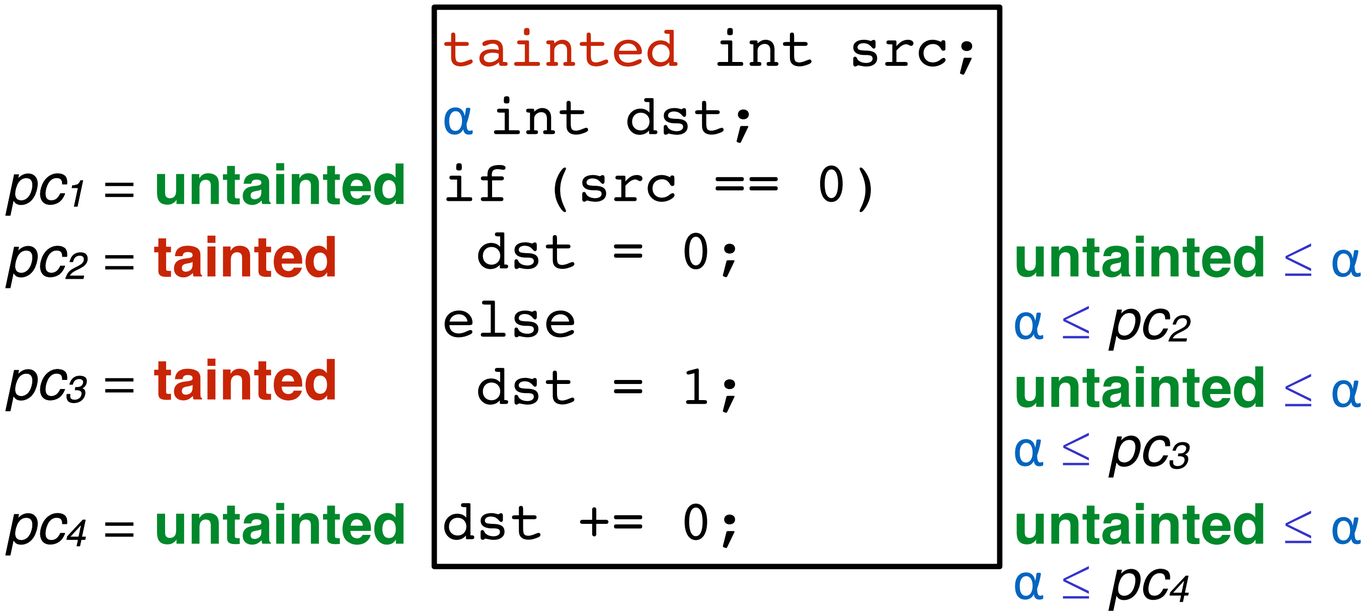
- The prior flow is an **implicit flow**, since information in one value *implicitly* influences another
- One way to discover these is to maintain a scoped **program counter (*pc*) label**
  - Represents the maximum taint affecting the current pc
- Assignments generate constraints involving the *pc*
  - `x = y` produces two constraints:

  label(`y`) ≤ *label*(`x`) (as usual)

  pc ≤ *label*(`x`)
- **Generalized analysis** tracks **information flow**

Non Examinable

Info flow example



Solution requires α **= tainted**

Discovers implicit flow

Non Examinable

Why not information flow?

- Tracking implicit flows with a *pc* label 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 also **hurt performance**
- Our copying example is *pathological*
  - We typically don't write programs like this
  - Implicit flows will have little overall influence
- *So*: **tainting analyses tend to ignore implicit flows**

Non Examinable

Other challenges

- **Taint through operations**
  - `tainted a; untainted b; c=a+b` — is `c` tainted? (yes, probably)
- **Function pointers**
  - What function can this call go to?
  - Can flow analysis to compute possible targets
- **Struct fields**
  - Track the taintedness of the whole struct, or each field?
  - Taintedness for each struct instance, or shared among all of them (or something in between)?
    - Note: objects ≈ structs + function pointers
- **Arrays**
  - Keep track of taintedness of each array element, or one element representing the whole array?

Non Examinable

Refining taint analysis

- Can label *additional* sources and sinks
  - Array bounds accesses: must have untainted index
- Can expand taint analysis to **handle *sanitizers***
  - Functions to convert tainted data to untainted data
- Other application: Leaking confidential data
  - Don’t want **secret sources** to go to **public sinks**
    - **Implicit flows more relevant** in this setting
  - *Dual* of tainting

Non Examinable

Other kinds of analysis

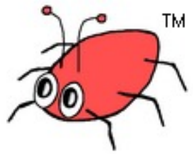
- **Pointer Analysis** (“points-to” analysis)
  - Determine whether pointers point to the same locations
  - Shares many elements of flow analysis. Really advanced in the last 10 years.
- **Data Flow Analysis**
  - Invented in the early 1970’s. Flow sensitive, tracks “data flow facts” about variables 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

Non Examinable

Static analysis in practice

Commercial products

Open source tools



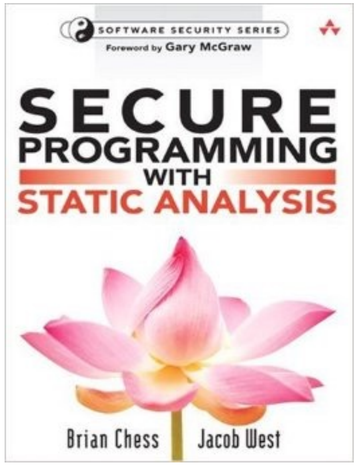
FindBugs

clang analyzer & KLEE

Caveat: appearance in the above list is not an implicit endorsement, and these are only a sample of available offerings

Learning more

- **Secure Programming with Static Analysis**, by Brian Chess, goes into more depth about how static analysis tools work, and can aid secure software development



- **Principles of Program Analysis**, by Nielson, Nielson, and Hankin, is a formal, mathematical presentation of different analysis methods
  - A bit dense for the casual reader, but good for introducing the academic field