Static Analysis

With material from Dave Levin, Mike Hicks, Dawson Engler, Lujo Bauer

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html
How to get software assurance?

- **Testing**: Check correctness on set of inputs

- **Benefits**: Concrete failure proves issue, aids fix

- **Drawbacks**: Expensive, difficult, coverage?
  - No guarantees
How to get software assurance?

- **Code audit**: Convince someone your code is correct
- **Benefit**: Humans can generalize
- **Drawbacks**: Expensive, hard, no guarantees
• How can we do better?
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/](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 *string* come from a *tainted source*?
  - Is this *pointer used after* its memory is *freed*?
  - Do any variables experience *data races*?
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
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
• Last time: Intro to static analysis
  • Precision (sound vs. complete), scalability, understandability

• Today: Taint analysis (briefly)
  • One example of a static technique
  • Then web security starts

• No class on Thursday!
Adding some depth: Taint (flow) analysis
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
The problem, in types

- Specify our requirement as a type qualifier

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

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

```c
tainted char *name = fgets(...,network_fd);
printf(name);  // **FAIL**: untainted <- tainted
```
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 specified (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 (mostly) *sound* analysis
Define allowed flow as a lattice:

\[
\text{untainted} < \text{tainted}
\]

At each program step, **test** whether \( \text{inputs} \leq \text{policy} \)

(Read as: input less tainted (or equal) than policy)
Analysis Approach

• If no qualifier is present, we must infer it

• Steps:
  • **Create a name** for each missing qualifier (e.g., $\alpha$, $\beta$)
  • For each program statement, **generate constraints**
    • Statement $x = y$ generates constraint $q_y \leq q_x$
  • **Solve the constraints** to produce solutions for $\alpha$, $\beta$, etc.
    • A solution is a *substitution* of qualifiers (like tainted or untainted) for names (like $\alpha$ and $\beta$) such that all of the constraints are legal flows

• If there is **no solution**, we (may) have an **illegal flow**
Example Analysis

First constraint requires $\alpha = \text{tainted}$
To satisfy the second constraint implies $\beta = \text{tainted}$
But then the third constraint is illegal: $\text{tainted} \leq \text{untainted}$
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);
```

tainted ≤ α
α ≤ β
untainted ≤ β
β ≤ untainted

No constraint solution. Bug? False Alarm!
Flow Sensitivity

- Our analysis is **flow insensitive**
  - 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., $\alpha_1$ is x’s qualifier at line 1, but $\alpha_2$ is the qualifier at line 2, where $\alpha_1$ and $\alpha_2$ can differ
  - Could implement this by transforming the program to assign to a variable at most once
Reworked Example

int printf(untainted char *fmt, ...);
\textit{tainted} char *fgets(...);

\[\begin{align*}
\alpha \text{ char } *\text{name} & = \text{fgets}(..., \text{network\_fd}); \\
\beta \text{ char } *x_{\text{1}}, \gamma *x_{\text{2}} & \quad \text{ (x1 = name)} \\
& \text{x\text{2} = ”%s”;} \\
& \text{printf(x2);}
\end{align*}\]

\textit{tainted} \leq \alpha \\
\alpha \leq \beta \\
\textit{untainted} \leq \gamma \\
\gamma \leq \text{untainted}

\textbf{No Alarm} \quad \textbf{No Alarm} \quad \textbf{No Alarm} \\
\textbf{Good solution exists:} \\
\gamma = \text{untainted} \\
\alpha = \beta = \text{tainted}
Handling conditionals

```c
int printf(untainted char *fmt, ...);
int 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
Multiple Conditionals

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

void f(int x) {
    char *y;
    if (x) y = "hello!";
    else y = fgets(..., network_fd);
    if (x) printf(y);
}
```

\[ \alpha \leq \text{untainted} \]
\[ \text{tainted} \leq \alpha \]
\[ \alpha \leq \text{untainted} \]

No solution for \( \alpha \). 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*

```c
void f(int x) {
    char *y;
    if (x)
        y = "hello!";
    else
        y = fgets(...);
    if (x)
        printf(y);
}
```

• A **path sensitive analysis** checks feasibility, e.g., by qualifying each constraint with a **path condition**
  • \( x \neq 0 \implies \text{untainted} \leq \alpha \) (segment 1-2)
  • \( x = 0 \implies \text{tainted} \leq \alpha \) (segment 1-3)
  • \( x \neq 0 \implies \alpha \leq \text{untainted} \) (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
Implicit flows

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

Illega flow:

**tainted **≠ **untainted**
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?
        }
    }
}
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:
    - $\text{label}(y) \leq \text{label}(x)$ (as usual)
    - $pc \leq \text{label}(x)$
Implicit flow example

\[
\begin{align*}
&pc_1 = \text{untainted} & \text{tainted} \text{ int src;} \\
&pc_2 = \text{tainted} & \alpha \text{ int dst;} \\
&pc_3 = \text{tainted} & \text{if (src == 0)} \\
&pc_4 = \text{untainted} & \quad \text{dst} = 0; \\
& & \text{else} \\
& & \quad \text{dst} = 1; \\
& & \quad \text{dst} += 0;
\end{align*}
\]

Taint on $\alpha$ is identified.

Discovers implicit flow!
Why not implicit flow?

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

    ```java
    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 \(\approx\) 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
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**
Static analysis in practice

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