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
Static analysis
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
• 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/]
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 $\approx$ Index in Bounds

- Change all exits to infinite loops (guaranteed no terminate)
- Change out-of-bounds index to exit:
  - $(i \geq 0 \&\& i < a\.length) ? a[i] : exit()$
- 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
**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
Trivially Complete: 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** (and not **both**)
  - … usually *leans* one way or the other
The Art of Static Analysis

• Design goals:
  • **Precision**: Carefully model program, 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
    • OK to forbid yucky code in the name of safety
    • Code that is more understandable to the analysis is more understandable to humans
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` ($\leq$ 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

```c
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*

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

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

```c
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

```c
void f(tainted int);
untainted int a = ...;
f(a);
```

*f accepts `tainted` or `untainted` data

```
untainted \leq \text{tainted}
```

Define allowed flow as a lattice:

```
untainted < \text{tainted}
```

Illegal Flow

```c
void g(untainted int);
tainted int b = ...;
g(b);
```

*g accepts only `untainted` data

```
tainted \not< \text{untainted}
```

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

```c
int printf(untainted char *fmt, ...);
tainted char *fgets(...);
char *name = fgets(..., network_fd);
char *x = name;
printf(x);
```

1. `tainted ≤ α`
2. `α ≤ β`
3. `β ≤ untainted`

Illegal flow!

No possible solution for `α` and `β`

First constraint requires `α = tainted`
To satisfy the second constraint implies `β = tainted`
But then the third constraint is illegal: `tainted ≤ untainted`
Taint Analysis: Adding Sensitivity
But what about?

\[
\begin{align*}
\text{int } \text{printf} & \left(\text{untainted } \text{char } * \text{fmt, ...} \right); \\
\text{tainted } \text{char } * \text{fgets} & \left(\ldots\right);
\end{align*}
\]

\[
\begin{align*}
\alpha \text{ char } * \text{name} & = \text{fgets} \left(\ldots, \text{network_fd}\right); \\
\beta \text{ char } * \text{x} & ; \\
x & = \text{name}; \\
x & = \text{"hello!"}; \\
\text{printf} & \left(\text{x}\right);
\end{align*}
\]

\[
\begin{align*}
\text{tainted} & \leq \alpha \\
\alpha & \leq \beta \\
\text{untainted} & \leq \beta \\
\beta & \leq \text{untainted}
\end{align*}
\]

No constraint solution. Bug? \textbf{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, ...);
tainted char *fgets(...);

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 \text{untainted}

No Alarm

Good solution exists:

\gamma = \text{untainted}
\alpha = \beta = \text{tainted}
Handling conditionals

```c
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

```c
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);
}
```

untainted ≤ α

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

- A path sensitive analysis checks feasibility, e.g., by qualifying each constraint with a path condition
  - $x \neq 0 \Rightarrow \text{untainted} \leq \alpha$ (segment 1-2)
  - $x = 0 \Rightarrow \text{tainted} \leq \alpha$ (segment 1-3)
  - $x \neq 0 \Rightarrow \alpha \leq \text{untainted}$ (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
Handling Function Calls

\[ \alpha \text{ char } *a = \text{fgets}(\ldots); \]
\[ \beta \text{ char } *b = \text{id}(a); \]
\[ \delta \text{ char } *\text{id}(\gamma \text{ char } *x) \{ \]
\[ \quad \text{return } x; \]
\[ \}

- Names for arguments and return value
- Calls create flows
  - from \textbf{caller’s data} to \textbf{callee’s arguments},
  - from \textbf{callee’s result} to \textbf{caller’s returned value}
Handling Function Calls

\[
\begin{align*}
\alpha & \quad \text{char } *a = \text{fgets}(\ldots); \\
\beta & \quad \text{char } *b = \text{id}(a); \\
\gamma & \quad \text{char } *x \\
\delta & \quad \text{char } *\text{id}(\gamma \text{ char } *x) \\
& \quad \{ \\
& \quad \quad \text{return } x; \\
& \quad \}
\end{align*}
\]

\text{tainted} \leq \alpha
\alpha \leq \gamma
\gamma \leq \delta
\delta \leq \beta

Result: b is tainted (as expected)
Function Call Example

\[ \alpha \text{ char } *a = \text{fgets(...)}; \]
\[ \beta \text{ char } *b = \text{id}(a); \]
\[ \omega \text{ char } *c = "hi"; \]
\[ \text{printf}(c); \]

\[ \delta \text{ char } *\text{id}(\gamma \text{ char } *x) \{ \]
\[ \text{return } x; \]
\[ \} \]

\[ \text{tainted} \leq \alpha \]
\[ \alpha \leq \gamma \]
\[ \gamma \leq \delta \]
\[ \delta \leq \beta \]
\[ \text{untainted} \leq \omega \]
\[ \omega \leq \text{untainted} \]

No Alarm

Good solution exists:

\[ \omega = \text{untainted} \]
\[ \alpha = \beta = \gamma = \delta = \text{tainted} \]
Two Calls to Same Function

\[\alpha\] char *a = fgets(…);
\[\beta\] char *b = id(a);
\[\omega\] char *c = id("hi");
printf(c);

\[\delta\] char *id(\gamma\ char *x) {
  return x;
}

\text{tainted} \leq \alpha
\alpha \leq \gamma
\gamma \leq \delta
\delta \leq \beta
\text{untainted} \leq \gamma
\delta \leq \omega
\omega \leq \text{untainted}

No solution. Real bug? \textbf{False Alarm!}
Two Calls to Same Function

\begin{align*}
\alpha & \text{ char } \ast a = \text{fgets}(\ldots); \\
\beta & \text{ char } \ast b = \text{id}(a); \\
\omega & \text{ char } \ast c = \text{id}(\text{“hi”}); \\
\delta & \text{ printf}(c);
\end{align*}

\begin{align*}
\delta & \text{ char } \ast \text{id}(\gamma \text{ char } \ast x) \{ \\
& \quad \text{return } x; \\
\}
\end{align*}

\text{Problematic constraints represent an infeasible path} \quad \text{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 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

```c
char *id(char *x) {
    return x;
}
```

```c
char *a = fgets(...);
char *b = id(a);
char *c = id("hi");
printf(c);
```

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*
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 ≠ 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 flows

Missed flow!
Information 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)
    - \( pc \leq \text{label}(x) \)
Information flow example

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

Taint on $\alpha$ is identified.
Discovers implicit flow!
Why not information flow?

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

```
tainted int src;
a 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 pointers
  • Where can this call go? 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
Meta-level compilation (briefly)
Meta-level compilation

(Engler et al., 2000)

- Compilers are good at verifying rules
  - But don’t have domain-specific knowledge
- Developers have domain knowledge
  - But manual inspection is painful, erratic
- Metaccompilation: Devs give compiler extra rules
1. Write rules
   • Define legal and error states
   • Specify state transitions

2. Meta-compiler checks rules
   • Flow sensitive, context sensitive (more or less)
   • Scales well to large, real programs
Simple example: Interrupts

After interrupts are disabled, they should be re-enabled.

1. Define states

- s_enabled
- s_disabled
- ERROR
- WARNING
Simple example: Interrupts

After interrupts are disabled, they should be re-enabled.

2. Define transitions
More interesting example: Free

\( v = \text{malloc}() \)

**unknown**
- \( v = 0 \): TRUE
- \( v \neq 0 \): FALSE

**null**
- use \(*v\) as operand

**non-null**
- \( v = 0 \): FALSE
- \( v \neq 0 \): TRUE

**freed**
- free(v)

**ERROR**
- free(v)
- use v as operand
Developer writes state machine

Many rules are transferable to other programs

```c
// No dereferences of null or unknown ptrs.
v.null, v.unknown: { *(any *)v } ==> 
    { err("Using ptr illegally!"); };

// Allow free of all non-freed variables.
v.unknown, v.null, v.not_null: 
    { free(v); } ==> v.freed;

// Check for double free and use after free.
v.freed:
    { free(v) } ==> { err("Dup free!"); }
    | { v } ==> { err("Use-after-free!"); };

// Overwriting v's value kills its state
v.all: { v = v1 } ==> v.ok;
```
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.