

Type Qualifiers, Subtyping, and Type Inference

Jeff Foster
CMSC 631
September 17, 2002

Format String Vulnerabilities

- I/O functions in C use format strings

```
printf("Hello!");           Hello!  
printf("Hello, %s!", name); Hello, name!
```

- Instead of

```
printf("%s", name)
```

Why not

```
printf(name);           ?
```

2

Format String Attacks

- Adversary-controlled format specifier

```
name := <data-from-network>  
printf(name);    /* Oops */
```

- Attacker sets name = "%s%s%s" to crash program
- Attacker sets name = "...%n..." to write to memory

- Lots of these bugs in the wild

- New ones weekly on bugtraq mailing list
- Too restrictive to forbid variable format strings

3

Types to the Rescue!

- Observation: two types of strings

- strings that come from the network
- strings that are used as format strings
- ...but both have type `char *`

- Use **qualified types** to distinguish trust levels

- **tainted** `char *` = may be from network
- **untainted** `char *` = must not be from network
- Rule: **tainted** data never used in **untainted** positions

4

Subtyping with Qualifiers

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

OK

f accepts **tainted** or
untainted data

untainted \sqsubseteq **tainted**

untainted < **tainted**

```
void g(untainted int);  
tainted int b;  
f(b);
```

Error

g accepts only **untainted**
data

tainted $\not\sqsubseteq$ **untainted**

5

Demo

<http://www.cs.berkeley.edu/~jfoster>

Adding Type Qualifiers

- Work with simply-typed lambda calculus
 - (Talk about moving to C later)

$e ::= c \mid x \mid \lambda x:t.e \mid e_1 e_2 \mid \dots$

$t ::= \text{char} \mid \text{ptr}(t) \mid t_1 \sqsubseteq t_2$

7

Adding Type Qualifiers

- Work with simply-typed lambda calculus
 - (Talk about moving to C later)

$e ::= c \mid x \mid \lambda x:t.e \mid e_1 e_2 \mid \dots$

$t ::= Q s$

$s ::= \text{char} \mid \text{ptr}(t) \mid t_1 \sqsubseteq t_2$

$Q ::= \text{untainted} \mid \text{tainted} \mid \dots$

- Plus a partial order \sqsubseteq on Q

8

Type Checking Rules — Variables

$$\frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

13

Type Checking Rules — Function Definition

$$\frac{A, x : t1 \vdash e : t2}{A \vdash \lambda x : t1. e : t1 \rightarrow t2}$$

- Use programmer-specified $t1$ as parameter type
- No top-level qualifier on $t1 \rightarrow t2$
 - Must use `annot(...)`
- $t1$ and $t2$ themselves contain qualifiers

14

Type Checking Rules — Function Application

$$\frac{A \vdash e1 : Q(t1 \rightarrow t2) \quad A \vdash e2 : t}{A \vdash e1 \ e2 : t2}$$

$t \rightarrow t1$
/ Subtyping

15

Aren't Type Systems Great?

- Type rules are compact yet clear
- Type systems are well-understood
 - Lots of good foundational work
- No problem with higher-order functions, pointers, recursion, whole-program analysis
- Type systems are (usually) sound

16

Motivation for Inference

- Type systems may be great, but...
 - We have to write types and qualifiers everywhere
 - $x:t.e$
 - $\text{annot}(e, Q)$
 - Function params arguably OK, but can be tedious
 - Qualifier annotations very burdensome

17

Type Inference

- Given: a bare program with
 - no given function parameter types
 - no type qualifier annotations
 - (qualifier checks may occur)
- Automatically infer the missing parameter types and annotations
- ...or prove that no valid typing exists

18

Idea 1: Variables

- Consider the rule for typing integers

$$\frac{}{A \vdash c : \text{char}}$$

- What if we have no annotation?
 - Introduce a **qualifier variable k**
 k fresh
- $$\frac{}{A \vdash c : k \text{ char}}$$

19

Idea 2: Constraints

- Recall rule for typing application

$$\frac{A \vdash e1 : Q(t1 \rightarrow t2) \quad A \vdash e2 : t \quad t \rightarrow t1}{A \vdash e1 \ e2 : t2}$$

- Notice that t and $t1$ may contain variables
 - So we can't necessarily check $t \rightarrow t1$ right away
 - Instead, generate constraint $t \rightarrow t1$
 - Solve constraints when we've seen whole program
 - (Won't discuss implicit constraint on $e1$)

20

Algorithm for Type Inference

- Step 1: Generate a system of constraints
 - Walk over AST of program and apply type rules
 - Only one rule applies to each kind of AST node
 - Generate fresh variables where needed
 - Generate constraints for function application
- Result: A pile of constraints

21

Algorithm for Type Inference (cont'd)

- Step 2: Rewrite constraints in simpler form

$$C \cup \{Q \text{ int} \sqsubseteq Q' \text{ int}\} \sqsubseteq C \cup \{Q \sqsubseteq Q'\}$$
$$C \cup \{Q \text{ ptr}(t) \sqsubseteq Q' \text{ ptr}(t')\} \sqsubseteq C \cup \{Q \sqsubseteq Q'\} \cup \{t \sqsubseteq t'\} \cup \{t' \sqsubseteq t\}$$
$$C \cup \{Q (t_1 \sqsubseteq t_2) \sqsubseteq Q' (t'_1 \sqsubseteq t'_2)\} \sqsubseteq C \cup \{Q \sqsubseteq Q'\} \cup \{t_1' \sqsubseteq t_1\} \cup \{t_2 \sqsubseteq t_2'\}$$

22

Algorithm for Type Inference (cont'd)

- Step 3: Solve qualifier constraints
 - Remaining constraints of the form $Q \sqsubseteq Q'$
 - Q and Q' may be variables
 - Use standard fixpoint computation

23

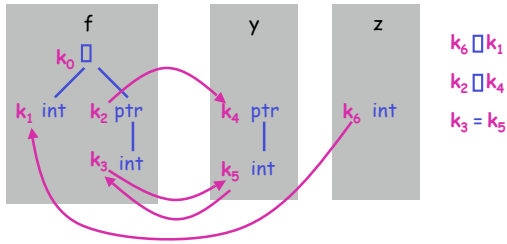
Computing a Qualifier Constraint Solution S

- For **tainted** and **untainted**
 - Initially set $S(k) = \text{untainted}$ for all k
 - Until no change
 - Pick a $Q \sqsubseteq Q'$
 - If $S(Q)$ **tainted**, set $S(Q')$ to **tainted**
 - Error if Q' is **untainted**

24

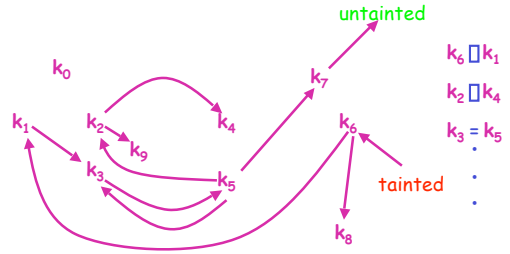
Putting It Together: Constraints as Graphs

ptr(int) f(x : int) = { ... } y := f(z)



25

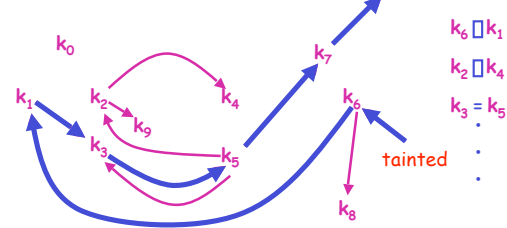
Putting It Together: Constraints as Graphs



26

Constraint Resolution via Graph Reachability

tainted \sqsupseteq k₆ \sqsupseteq k₁ \sqsupseteq k₃ \sqsupseteq k₅ \sqsupseteq k₇ \sqsupseteq untainted



27

Constraint Resolution in Linear Time

- Initial program of size n
 - Fixed set of qualifiers tainted, untainted, ...
 - Assume annotated with standard types
- Constraint generation yields O(n) constraints
 - Recursive abstract syntax tree walk
- Graph reachability takes O(n) time

28

L-Values and R-Values in C

- Consider:

```
int x;  
x = 42;      3 = 42; /* wrong! */  
y = x;      y = 3;
```

- But both `x` and `3` have type `int`

- Clearly something is missing from the type
 - C uses syntax to distinguish l- and r-values

29

Distinguishing L- and R-values with Types

- L-values given `ptr()` types

- Only `ptr()` types on l.h.s. of assignment
- Implicit dereference of l-value in r-position

```
x : ptr(int)    3 : int  
x = 42;        3 = 42;  
y = *x;        y = 3;
```

30

Aggregate Types in C

- Use product constructor for `structs`

```
s ::= t1 x ... x tn | ...
```

```
C U Q (t1 x ... x tn) Q' (t1' x ... x tn')  
C U {Q Q'} U {t1 t1'} U ... U {tn tn'}
```

- Treat `unions` as `structs`
 - ANSI C says they have to be safe
 - ...or treat like type casts

31

Type Casts

- C lets you assign any type to an expression

- Solution 1: Add qualifiers at casts

```
tainted char *x; y = (tainted void *)x;
```

- Solution 2: Propagate qualifiers through casts

```
tainted char *x; y = (void*) x;
```

- Note: not completely sound

32

Whole-Program Analysis

- C programs usually spread across several files

```
char x, y;  
y = x;
```

File 1

```
extern char x;  
x = <tainted data>;
```

File 2

- Need to analyze all files together
 - Equate types of globals across files
 - $t_{x1} \sqsubseteq t_{x2}, t_{x2} \sqsubseteq t_{x1}$
 - Note: need to match up **structs**
 - Memoize matching to be efficient

33

Library Functions

- C programs use standard library functions
 - Don't necessarily have source code
- Solution 1: Write set of stub functions
- Solution 2: Write complete type signatures
 - For format-string vulnerabilities, give parametric polymorphic (universally quantified) signatures

34

A Hint: Flow-Sensitive Type Qualifiers

- Standard type systems are flow-insensitive
 - Types don't change during execution

```
/* x : int */ x := ...; /* x : int */
```

- For some problems, we need *flow-sensitivity*
 - Allow qualifiers to change during execution

```
/* y : locked Lock */ y := ...; /* y : unlocked Lock */
```

35