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

• Instead of
  printf("%s", name)

  Why not
  printf(name);

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

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
Subtyping with Qualifiers

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

OK

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

Error

f accepts tainted or un tainted data
untainted [] tainted
untainted < tainted

tainted [] un tainted
tainted < un tainted

Adding Type Qualifiers

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

  e ::= c | x | tQ:e | e1 e2 | ...

  t ::= char | ptr(t) | t1 [] t2

  Q ::= untainted | tainted | ...

  - Plus a partial order ≤ on Q

Demo

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

Adding Type Qualifiers

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

  e ::= c | x | tQ:e | e1 e2 | ...

  t ::= Q s
  s ::= char | ptr(t) | t1 [] t2
  Q ::= untainted | tainted | ...

  - Plus a partial order ≤ on Q
Adding Type Qualifiers

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

  \[ e ::= c \mid x \mid \text{dec} \cdot e \mid e_1 e_2 \mid \ldots \]
  \[ t ::= Q s \]
  \[ s ::= \text{char} \mid \text{ptr}(t) \mid t_1 \uparrow t_2 \]
  \[ Q ::= \text{untainted} \mid \text{tainted} \mid \ldots \]

  - Plus a partial order \( \sqsubseteq \) on \( Q \)

Type Judgments

**Type environment**

\[ A \vdash k : t \]

Expression

"In type environment \( A \), expression \( e \) has type \( t \)"

Type Checking Rules — Characters

- \[ A \vdash k : \text{char} \]
  - No qualifier

- \[ A \vdash k : s \]
  - \[ A \vdash \text{annot}(e, Q) : Q' s \]

Example: \( A \vdash \text{annot}(‘a’, \text{tainted}) : \text{tainted char} \)

Type Checking Rules — Qualifier Checks

- \[ A \vdash k : Q' s \]
  - \[ Q' \sqsubseteq Q \]
  - \[ A \vdash \text{check}(e, Q) : Q' s \]

Example: \( A \vdash \text{check}(\text{annot}(‘a’, \text{tainted}), \text{untainted}) : t \)
Type Checking Rules — Variables

\[ x \in \text{dom}(A) \]
\[ A[x : A(x)] \]

Type Checking Rules — Function Definition

\[ A, x : t_1 \vdash e : t_2 \]
\[ A \vdash x : t_1 \]

- Use programmer-specified \( t_1 \) as parameter type
- No top-level qualifier on \( t_1 \) \( \quad t_2 \)
  - Must use \( \text{annot}(...) \)
- \( t_1 \) and \( t_2 \) themselves contain qualifiers

Type Checking Rules — Function Application

\[ A \vdash s_1 : Q(t_1 \quad t_2) \quad A \vdash s_2 : t \]
\[ A \vdash s_1 \ e_2 : t_2 \]

\[ t \quad t_1 \]

Subtyping

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
Motivation for Inference

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

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

Idea 1: Variables

- Consider the rule for typing integers
  \[ A \vdash x : \text{char} \]
  - What if we have no annotation?
- Introduce a qualifier variable \(k\)
  \[ k \text{ fresh} \]
  \[ A \vdash k : k \text{ char} \]

Idea 2: Constraints

- Recall rule for typing application
  \[ A \vdash e_1 : Q (t_1 \varnothing t_2) \quad A \vdash e_2 : t \]
  \[ t \varnothing t_1 \]
  \[ A \vdash e_1 \ e_2 : t_2 \]
- Notice that \(t\) and \(t_1\) may contain variables
  - So we can’t necessarily check \(t \varnothing t_1\) right away
  - Instead, generate constraint \(t \varnothing t_1\)
  - Solve constraints when we’ve seen whole program
  - (Won’t discuss implicit constraint on \(e_1\)
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

Algorithm for Type Inference (cont'd)

- Step 2: Rewrite constraints in simpler form

\[
C \cup \{Q \text{ int } Q' \text{ int} \} \quad C \cup \{Q \text{ Q}'\}
\]
\[
C \cup \{Q \text{ ptr}(t) Q' \text{ ptr}(t')\} \quad C \cup \{Q \text{ Q'}\} \cup \{t \text{  t'}\} U \{t' \text{  t}\}
\]
\[
C \cup \{Q (t1 \text{  t2}) Q'(t'1 \text{  t'2})\} \quad C \cup \{Q \text{ Q'}\} U \{t1' \text{  t1}\} U \{t2 \text{  t'2}\}
\]

Algorithm for Type Inference (cont'd)

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

Computing a Qualifier Constraint Solution \( S \)

- For \textit{tainted} and \textit{untainted}
  - Initially set \( S(k) = \text{untainted} \) for all \( k \)
  - Until no change
    - Pick a \( Q \sqcap Q' \)
    - If \( S(Q) \) \textit{tainted}, set \( S(Q') \) to \textit{tainted}
      - Error if \( Q' \) is \textit{untainted}
Putting It Together: Constraints as Graphs

\[
\text{ptr(int) } f(x : \text{int}) = \{ \ldots \} \quad y := f(z)
\]

Constraint Resolution via Graph Reachability

\[
\text{tainted} \bullet k_0 \bullet k_1 \bullet k_2 \bullet k_3 \bullet k_4 \bullet k_5 \bullet \text{untainted}
\]

Constraint Resolution in Linear Time

- Initial program of size \( n \)
  - Fixed set of qualifiers \text{tainted}, \text{untainted}, ...
  - Assume annotated with standard types
- Constraint generation yields \( O(n) \) constraints
  - Recursive abstract syntax tree walk
- Graph reachability takes \( O(n) \) time
L-Values and R-Values in C

- Consider:
  ```c
  int x;
  x = 42;
  // wrong!
  3 = 42;
  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
  ```

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)  
  x = 42;  
  y = *x;  
  y = 3;
```

Aggregate Types in C

- Use product constructor for structs
  ```c
  s ::= t1 x ... x tn | ...
  C U Q (t1 x ... x tn) Q' (t1' x ... x tn') Q''
  C U (Q Q') U (t1 U t1') U ... U (tn U tn')
  ```

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

Type Casts

- C lets you assign any type to an expression

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

- Solution 1: Add qualifiers at casts

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

- Solution 2: Propagate qualifiers through casts

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

- Note: not completely sound
Whole-Program Analysis

- C programs usually spread across several files
  - Need to analyze all files together
    - Equate types of globals across files
      \[ t_{x1} \subseteq t_{x2} \land t_{x2} \subseteq t_{x1} \]\n    - Note: need to match up structs
      - Memoize matching to be efficient

```
char x, y;
y = x;  // File 1

extern char x;
x = <tainted data>;  // File 2
```

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

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 */
    ```