Type Qualifiers

Slides due to Jeff Foster
University of Maryland

Joint work with Alex Aiken, Rob Johnson, John Kodumal, Tachio Terauchi, and David Wagner

Type Qualifiers: Lightweight Specifications

- Extend standard type systems (C, Java, ML)
  - Programmers already use types
  - Programmers understand types
  - Get programmers to write down a little more...

const int	ANSI C
ptr(tainted char)	Format-string vulnerabilities
kernel ptr(char) → char	User/kernel vulnerabilities

Application: 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);                        ?
```

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
    - Yields (often remote root) exploits

- These bugs still occur in the wild

Using Tainted and Untainted

- Add qualifier annotations

```
int printf(untainted char *fmt, ...)
tainted char *getenv(const char *)
```

tainted = may be controlled by adversary

untainted = must not be controlled by adversary

Taintedness is an **integrity** property

- Dual to confidentiality, as per last lecture
- The technique we describe applies to **confidentiality** too

Subtyping

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

OK

f accepts tainted or untainted data

untainted ≤ tainted

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

Error

g accepts only untainted data

tainted ≰ untainted

untainted < tainted

The Plan

- The Nice Theory
- Polymorphism
- The Icky Stuff in C

Type Qualifiers for MinML

- We'll add type qualifiers to MinML
  - Same approach works for other languages (like C)

- Standard type systems define types as
  - t ::= c0(t, …, t) | … | cn(t, …, t)
    - Where Σ = c0…cn is a set of *type constructors*
- Recall the types of MinML
  - t ::= int | bool | t → t
    - Here Σ = int, bool, → (written infix)

Type Qualifiers for MinML (cont'd)

- Let Q be the set of type qualifiers
  - Assumed to be chosen in advance and fixed
  - E.g., Q = {tainted, untainted}
- Then the *qualified types* are just
  - qt ::= Q s
  - s ::= c0(qt, …, qt) | … | cn(qt, …, qt)
    - Allow a type qualifier to appear on each type constructor
- For MinML
  - qt ::= int^Q | $bool^Q$ | qt $\rightarrow^Q$ qt

Abstract Syntax of MinML with Qualifiers

e ::= x | n | true | false | if e then e else e |
fun f^{Q} (x:qt):qt = e | e e | annot(Q, e) | check(Q, e)

- annot(Q, e) = "expression e has qualifier Q"
- check(Q, e) = "fail if e does not have qualifier Q"
  - Checks only the top-level qualifier

- Examples:
  - fun fread (x:qt):$int^{tainted}$ = …annot(tainted, 42)
  - fun printf (x:qt):qt' = check(untainted, x), …

Typing Rules: Qualifier Introduction

- Newly-constructed values have "bare" types

$$\frac{}{G \vdash n : int}$$

$$\frac{}{G \vdash true : bool} \qquad \frac{}{G \vdash false : bool}$$

- Annotation adds an outermost qualifier

$$\frac{G \vdash e1 : s}{G \vdash annot(Q, e) : Q\ s}$$

Typing Rules: Qualifier Elimination

- By default, discard qualifier at destructors

$$\frac{G \vdash e1 : bool^{Q} \quad G \vdash e2 : qt \quad G \vdash e3 : qt}{G \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : qt}$$

- Use check() if you want to do a test

$$\frac{G \vdash e1 : Q\ s}{G \vdash \text{check}(Q, e) : Q\ s}$$

Subtyping

- Our example used *subtyping*
  - If anyone expecting a T can be given an S instead, then S is a *subtype* of T.
  - Allows untainted to be passed to tainted positions
  - I.e., check(tainted, annot(untainted, 42)) should typecheck

- How do we add that to our system?

Partial Orders

- Qualifiers Q come with a partial order $\leq$:
  - $q \leq q$ (reflexive)
  - $q \leq p, p \leq q \Rightarrow q = p$ (anti-symmetric)
  - $q \leq p, p \leq r \Rightarrow q \leq r$ (transitive)
- Qualifiers introduce subtyping

- In our example:
  - untainted < tainted

Extending the Qualifier Order to Types

$$\frac{Q \leq Q'}{bool^{Q} \leq bool^{Q'}} \qquad \frac{Q \leq Q'}{int^{Q} \leq int^{Q'}}$$

- Add one new rule *subsumption* to type system

$$\frac{G \mid\!-\!- e : qt \quad qt \leq qt'}{G \mid\!-\!- e : qt'}$$

- Means: If any position requires an expression of type qt' , it is safe to provide it a subtype qt

Use of Subsumption

$$\frac{\dfrac{\dfrac{\vdash 42 : \text{int}}{\vdash \text{annot}(\text{untainted}, 42) : \text{untainted int}} \qquad \text{untainted} \leq \text{tainted}}{\vdash \text{annot}(\text{untainted}, 42) : \text{tainted int}}}{\vdash \text{check}(\text{tainted}, \text{annot}(\text{untainted}, 42)) : \text{tainted int}}$$

Subtyping on Function Types

- What about function types?

$$\frac{?}{qt1 \rightarrow^{Q} qt2 \leq qt1' \rightarrow^{Q'} qt2'}$$

- Recall: S is a subtype of T if an S can be used anywhere a T is expected
  - When can we replace a call "f x" with a call "g x"?

Replacing "f x" by "g x"

- When is $qt1' \rightarrow^{Q'} qt2' \leq qt1 \rightarrow^{Q} qt2$?
- Return type:
  - We are expecting qt2 (f's return type)
  - So we can only return *at most* qt2
  - $qt2' \leq qt2$
- Example: A function that returns tainted can be replaced with one that returns untainted

Replacing "f x" by "g x" (cont'd)

- When is $qt1' \rightarrow^{Q'} qt2' \leq qt1 \rightarrow^{Q} qt2$?
- Argument type:
  - We are supposed to accept $qt1$ (f's argument type)
  - So we must accept *at least* $qt1$
  - $qt1 \leq qt1'$
- Example: A function that accepts untainted can be replaced with one that accepts tainted

Subtyping on Function Types

$$\frac{qt1' \leq qt1 \quad qt2 \leq qt2' \quad Q \leq Q'}{qt1 \rightarrow^{Q} qt2 \leq qt1' \rightarrow^{Q'} qt2'}$$

- We say that $\rightarrow$ is
  - *Covariant* in the range (subtyping dir the same)
  - *Contravariant* in the domain (subtyping dir flips)

Dynamic Semantics with Qualifiers

- Operational semantics tags values with qualifiers
  - $v ::= x \mid n^Q \mid true^Q \mid false^Q$
    $\mid fun\ f^Q\ (x : qt1) : qt2 = e$

- Evaluation rules same as before, carrying the qualifiers along, e.g.,

$$\frac{}{if\ true^Q\ then\ e1\ else\ e2 \rightarrow e1}$$

Dynamic Semantics with Qualifiers (cont'd)

- One new rule checks a qualifier:

$$\frac{Q' \leq Q}{\text{check}(Q, v^{Q'}) \rightarrow v}$$

  - Evaluation at a check can continue only if the qualifier matches what is expected
    - Otherwise the program gets *stuck*
  - (Also need rule to evaluate under a check)
- Goal: don't do any checking at run-time
  - Instead, prove that all checks will succeed

Soundness

- We want to prove
  - Preservation: Evaluation preserves types
  - Progress: Well-typed programs don't get stuck
- Proof: Exercise
  - See if you can adapt proofs to this system
  - (Not too much work; really just need to show that check doesn't get stuck)

Updateable References

- Our MinML language is missing *side-effects*
  - There's no way to write to memory
  - Recall that this doesn't limit expressiveness
    - But side-effects sure are handy

Language Extension

- We'll add ML-style references
  - e ::= … | ref^Q e | !e | e := e
    - ref^Q e -- Allocate memory and set its contents to e
      - Returns memory location
      - Q is qualifier on pointer (not on contents)
    - !e -- Return the contents of memory location e
    - e1 := e2 -- Update e1's contents to contain e2
  - Things to notice
    - No null pointers (memory always initialized)
    - No mutable local variables (only pointers to heap allowed)

Static Semantics

- Extend type language with references:
  - $qt ::= \ldots \mid ref^{Q}\ qt$
    - Note: In ML the ref appears on the right

$$\frac{G \vdash e : qt}{G \vdash ref^{Q}\ e : ref^{Q}\ qt}$$

$$\frac{G \vdash e : ref^{Q}\ qt}{G \vdash !e : qt} \qquad \frac{G \vdash e1 : ref^{Q}\ qt \quad G \vdash e2 : qt}{G \vdash e1 := e2 : qt}$$

Subtyping References

- The *wrong* rule for subtyping references is

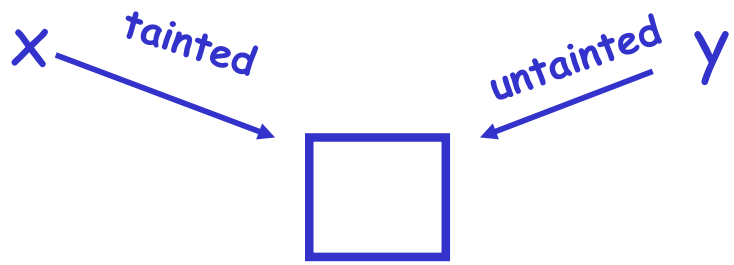
$$\frac{Q \leq Q' \qquad qt \leq qt'}{ref^{Q}\ qt \leq ref^{Q'}\ qt'}$$

- Counterexample

```
let x : ref^Q untainted int = ref 0^untainted in        for any Q
let y : ref^Q tainted int = x in                        ok if ref t int ≤ ref ut int
    y := 3^tainted;
    check(untainted, !x)                                oops!
```

You've Got Aliasing!

- We have multiple names for the same memory location
  - But they have different types
  - *And* we can **write** into memory at different types



Solution #1: Java's Approach

- Java uses this subtyping rule
  - If S is a subclass of T, then S[] is a subclass of T[]

- Counterexample:
  - Foo[] a = new Foo[5];
  - Object[] b = a;
  - b[0] = new Object(); // forbidden at runtime
  - a[0].foo(); // …so this can't happen

Solution #2: Purely Static Approach

- Reason from rules for functions
  - A reference is like an object with two methods:
    - get : unit $\rightarrow$ qt
    - set : qt $\rightarrow$ unit
  - Notice that qt occurs both co- and contravariantly
- The right rule:

$$\frac{Q \leq Q' \quad qt \leq qt' \quad qt' \leq qt}{ref^{Q}\ qt \leq ref^{Q'}\ qt'} \quad \text{or} \quad \frac{Q \leq Q' \quad qt = qt'}{ref^{Q}\ qt \leq ref^{Q'}\ qt'}$$

Challenge Problem: Soundness

- We want to prove
  - Preservation: Evaluation preserves types
  - Progress: Well-typed programs don't get stuck

- Can you prove it with updateable references?
  - Hint: You'll need a stronger induction hypothesis
    - You'll need to reason about types in the store
      - E.g., so that if you retrieve a value out of the store, you know what type it has

Type Qualifier Inference

- Recall our motivating example
  - We gave a legacy C program that had *no information* about qualifiers
  - We added signatures *only* for the standard library functions
  - Then we checked whether there were any contradictions

- This requires *type qualifier inference*

Type Qualifier Inference Statement

- Given a program with
  - Qualifier annotations
  - Some qualifier checks
  - And no other information about qualifiers
- Does there exist a valid typing of the program?
  - I.e., can we produce a legal typing derivation?
- We want an algorithm to solve this problem

First Problem: Subsumption Rule

$$\frac{G \vdash e : qt \quad qt \leq qt'}{G \vdash e : qt'}$$

- We're allowed to apply this rule at any time
  - Makes it hard to develop a deterministic algorithm
  - Type checking is not *syntax driven*
- Fortunately, we don't have that many choices
  - For each expression e, we need to decide
    - Do we apply the "regular" rule for e?
    - Or do we apply subsumption (how many times)?

Getting Rid of Subsumption

- Lemma: Multiple sequential uses of subsumption can be collapsed into a single use
  - Proof: Transitivity of $\leq$

- So now we need only apply subsumption once after each expression

Getting Rid of Subsumption (cont'd)

- We drop the separate subsumption rule
  - Incorporate it directly into the other rules

$$\frac{\dfrac{G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad qt1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt2}{G \vdash e1 : qt1 \rightarrow^{Q} qt2} \qquad \dfrac{G \vdash e2 : qt \qquad qt \leq qt1}{G \vdash e2 : qt1}}{G \vdash e1\ e2 : qt2}$$

Getting Rid of Subsumption (cont'd)

- 1. Fold e2 subsumption into rule

$$\frac{\dfrac{G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad qt1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt2}{G \vdash e1 : qt1 \rightarrow^{Q} qt2} \qquad G \vdash e2 : qt \quad qt \leq qt1}{G \vdash e1\ e2 : qt2}$$

Getting Rid of Subsumption (cont'd)

- 2. Fold e1 subsumption into rule

$$\frac{qt1 \leq qt' \quad Q' \leq Q \quad qt'' \leq qt2 \qquad G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad G \vdash e2 : qt \quad qt \leq qt1}{G \vdash e1\ e2 : qt2}$$

Getting Rid of Subsumption (cont'd)

- 3. We don't use Q, so remove that constraint

$$\frac{qt1 \leq qt' \qquad qt'' \leq qt2 \qquad G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad G \vdash e2 : qt \quad qt \leq qt1}{G \vdash e1\ e2 : qt2}$$

Getting Rid of Subsumption (cont'd)

- 4. Apply transitivity of $\leq$
  - Remove intermediate qt1

$$\frac{qt'' \leq qt2 \qquad G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad G \vdash e2 : qt \quad qt \leq qt'}{G \vdash e1\ e2 : qt2}$$

Getting Rid of Subsumption (cont'd)

- 5. We're going to apply subsumption afterward, so no need to weaken qt''

$$\frac{G \vdash e1 : qt' \rightarrow^{Q'} qt'' \qquad G \vdash e2 : qt \quad qt \leq qt'}{G \vdash e1\ e2 : qt''}$$

Getting Rid of Subsumption (cont'd)

- We similarly adjust the other rules
  - We're left with a purely syntax-directed system
- Good! Now we're half-way to an algorithm

Second Problem: Assumptions

- Let's take a look at the rule for functions:

$$\frac{G, f: qt1 \rightarrow^{Q} qt2, x:qt1 \vdash e : qt2' \quad qt2' \leq qt2}{G \vdash \text{fun } f^{Q}\ (x:qt1):qt2 = e : qt1 \rightarrow^{Q} qt2}$$

- There's a problem with applying this rule
  - We're assuming that we're given the argument type qt1 and the result type qt2
  - But in the problem statement, we said we only have annotations and checks

Unknowns in Qualifier Inference

- We've got *regular* type annotations for functions
  - (We could even get away without these…)

$$\frac{G, f{:}\ ? \rightarrow^{Q} ?, x{:}? \ \vdash e : qt2' \qquad qt2' \leq qt2}{G \ \vdash \text{fun } f^{Q}\ (x{:}t1){:}t2 = e : qt1 \rightarrow^{Q} qt2}$$

- How do we pick the qualifiers for f?
  - We generate fresh, unknown *qualifier variables* and then solve for them

Adding Fresh Qualifiers

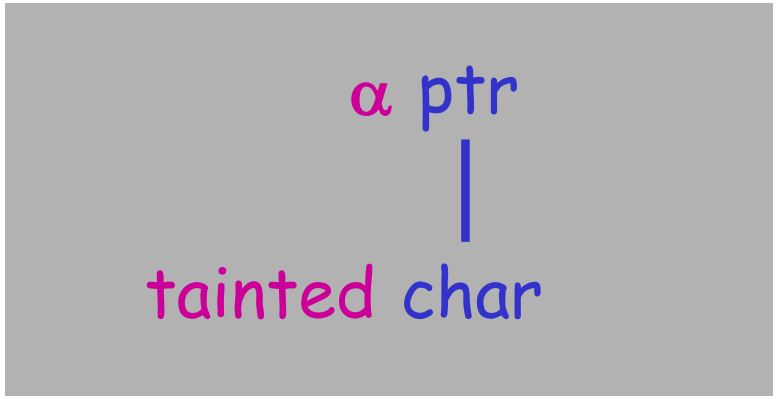
- We'll add qualifier variables a, b, c, … to our set of qualifiers
  - (Letters closer to p, q, r will stand for constants)
- Define fresh : t → qt as
  - $fresh(int) = int^a$
  - $fresh(bool) = bool^a$
  - $fresh(ref\ t) = ref^a\ fresh(t)$
  - $fresh(t1 \rightarrow t2) = fresh(t1) \rightarrow^a fresh(t2)$
    - Where a is fresh

Rule for Functions

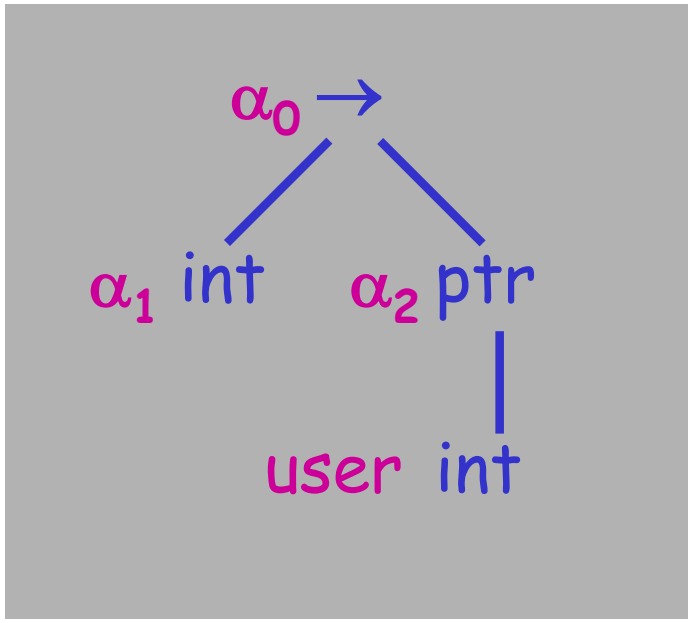
$$\frac{\begin{array}{c} qt1 = fresh(t1) \quad qt2 = fresh(t2) \\ G, f: qt1 \rightarrow^{Q} qt2, x{:}qt1 \vdash e : qt2' \\ qt2' \leq qt2 \end{array}}{G \vdash \text{fun } f^{Q}\ (x{:}t1){:}t2 = e : qt1 \rightarrow^{Q} qt2}$$

A Picture of Fresh Qualifiers

ptr(tainted char)



int → user ptr(int)



Where Are We?

- A syntax-directed system
  - For each expression, clear which rule to apply
- Constant qualifiers
- Variable qualifiers
  - Want to find a valid assignment to constant qualifiers
- Constraints $qt \leq qt'$ and $Q \leq Q'$
  - These restrict our use of qualifiers
  - These will limit solutions for qualifier variables

Qualifier Inference Algorithm

- 1. Apply syntax-directed type inference rules
  - This generates fresh unknowns and constraints among the unknowns
- 2. Solve the constraints
  - Either compute a *solution*
  - Or fail, if there is no solution
    - Implies the program has a type error
    - Implies the program *may* have a security vulnerability

Solving Constraints: Step 1

- Constraints of the form $qt \leq qt'$ and $Q \leq Q'$
  - $qt ::= int^Q \mid bool^Q \mid qt \rightarrow^Q qt \mid ref^Q\ qt$
- Solve by simplifying
  - Can read solution off of simplified constraints
- We'll present algorithm as a rewrite system
  - S ==> S' means constraints S rewrite to (simpler) constraints S'
  - Rules are derived from standard subtyping rules

Solving Constraints: Step 1

- $S + \{ int^{Q} \leq int^{Q'} \}$ ==> $S + \{ Q \leq Q' \}$
- $S + \{ bool^{Q} \leq bool^{Q'} \}$ ==> $S + \{ Q \leq Q' \}$
- $S + \{ qt1 \rightarrow^{Q} qt2 \leq qt1' \rightarrow^{Q'} qt2' \}$ ==>

  $S + \{ qt1' \leq qt1 \} + \{ qt2 \leq qt2' \} + \{ Q \leq Q' \}$
- $S + \{ ref^{Q}\ qt1 \leq ref^{Q'}\ qt2 \}$ ==>

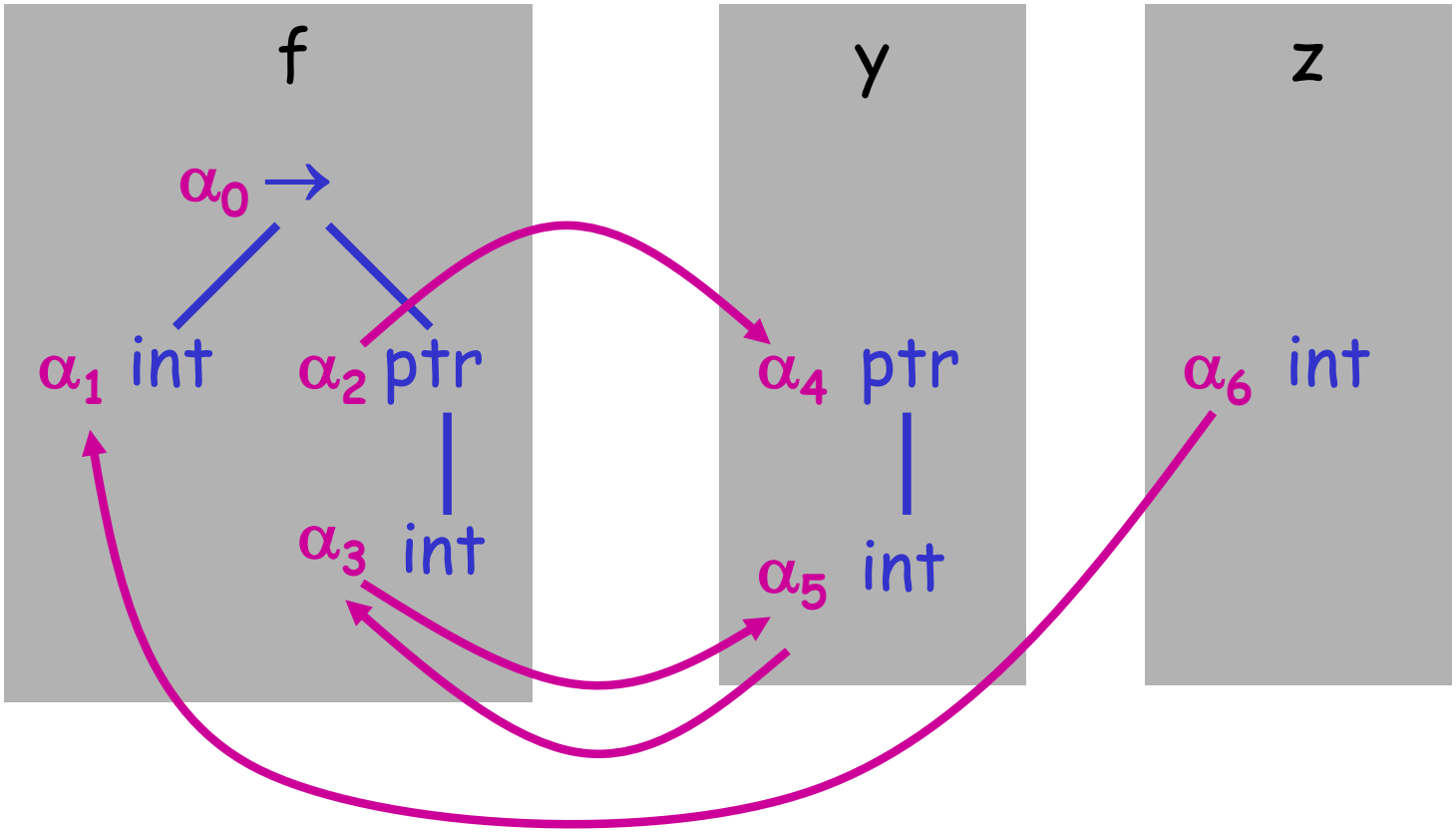
  $S + \{ qt1 \leq qt2 \} + \{ qt2 \leq qt1 \} + \{ Q \leq Q' \}$
- $S + \{$ mismatched constructors $\}$ ==> error
  - Can't happen if program correct w.r.t. std types

Solving Constriants: Step 2

- Our type system is called a *structural subtyping system*
  - If $qt \leq qt'$, then qt and qt' have the same shape
- When we're done with step 1, we're left with constraints of the form $Q \leq Q'$
  - Where either of Q, Q' may be an unknown
  - This is called an *atomic subtyping system*
  - That's because qualifiers don't have any "structure"

Constraint Generation

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

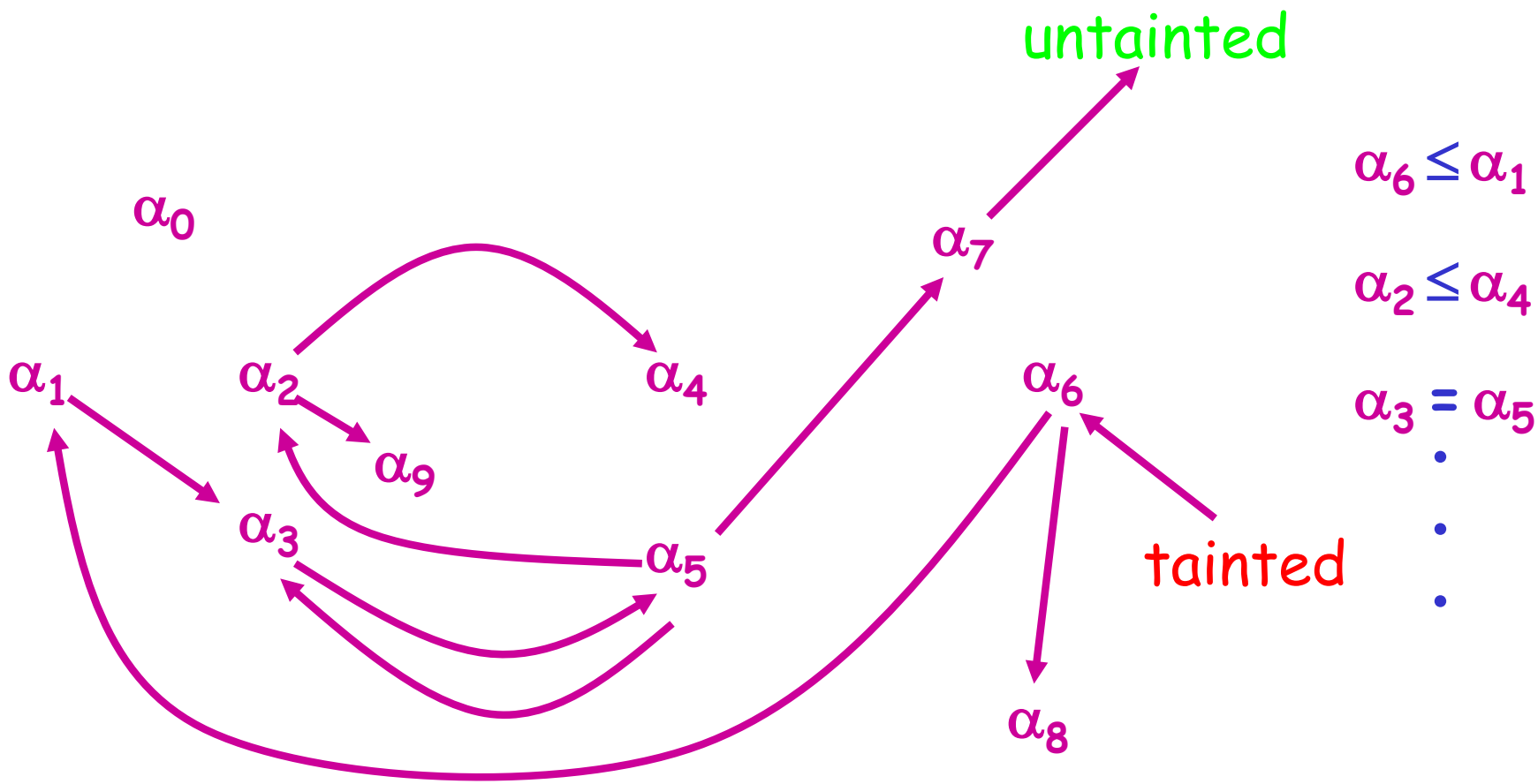


$\alpha_6 \leq \alpha_1$

$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

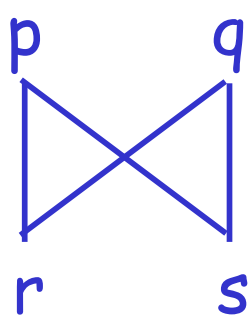
Constraints as Graphs



Some Bad News

- Solving atomic subtyping constraints is NP-hard in the general case

- The problem comes up with some really weird partial orders

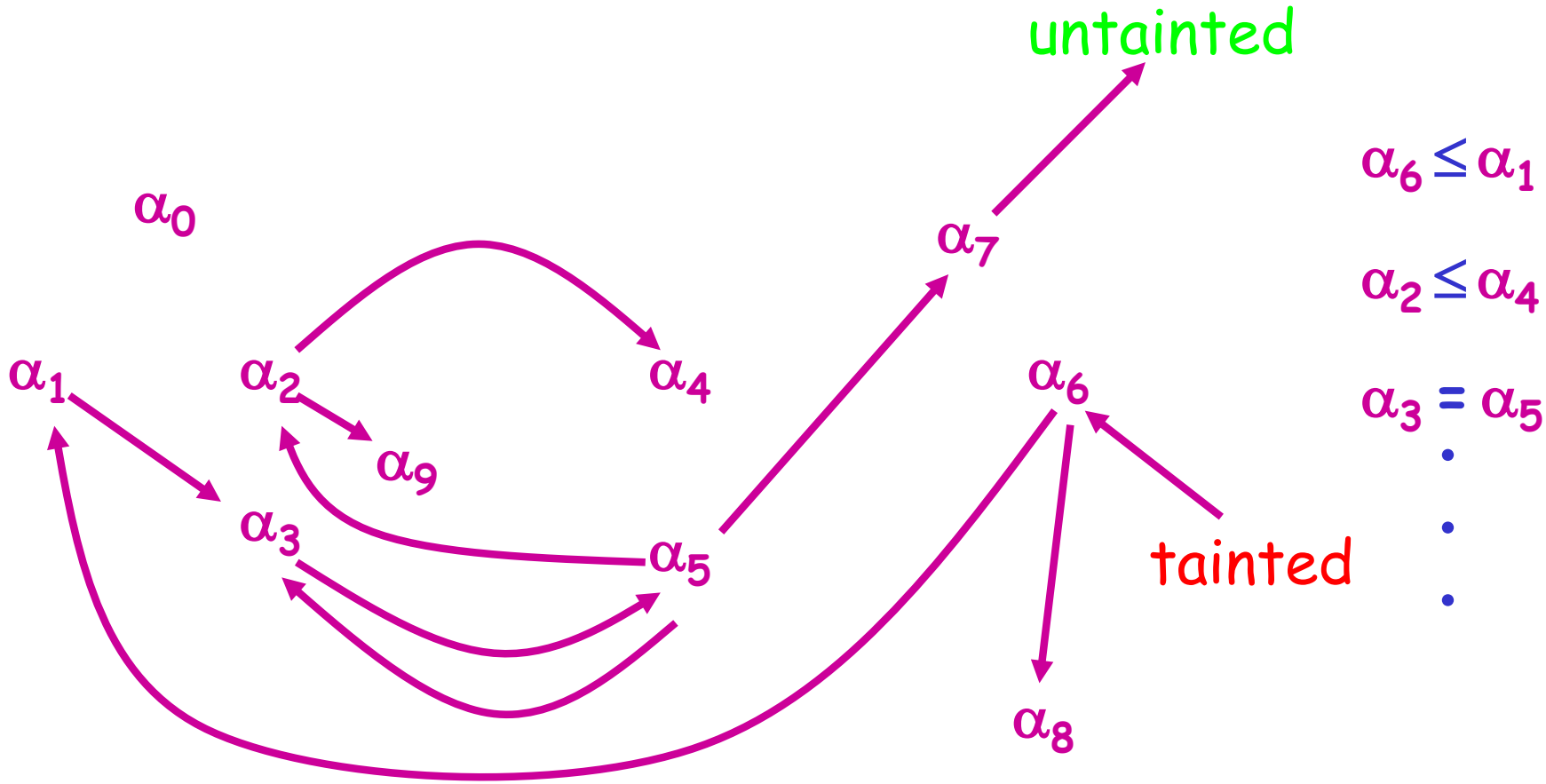


But that's OK

- These partial orders don't seem to come up in practice
- Most qualifier partial orders have one of two desirable properties:
  - They either always have *least upper bounds* or *greatest lower bounds* for any pair of qualifiers
- If Q is a **lattice**, it turns out we can use a really simple algorithm to check satisfiability of constraints over Q

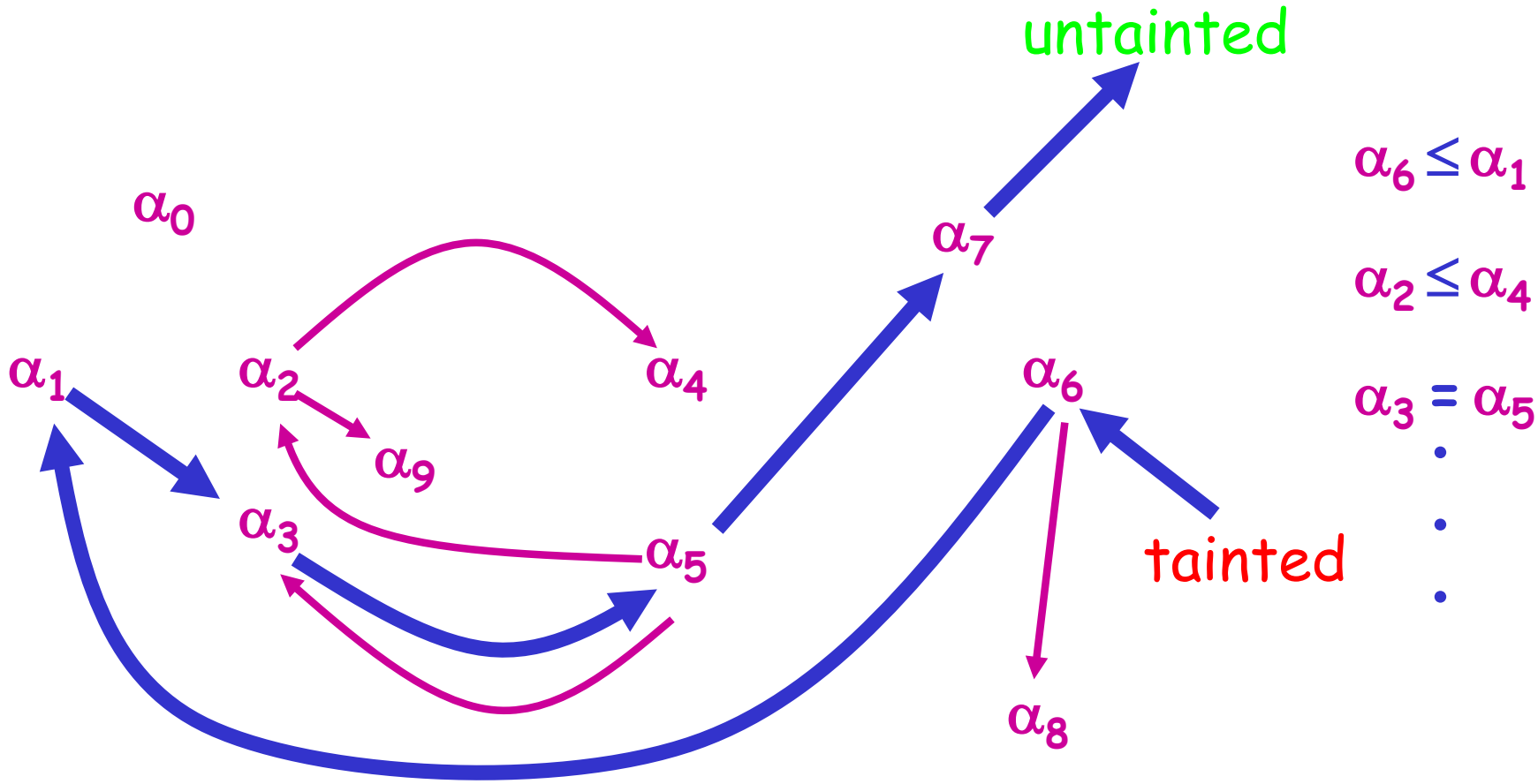
Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



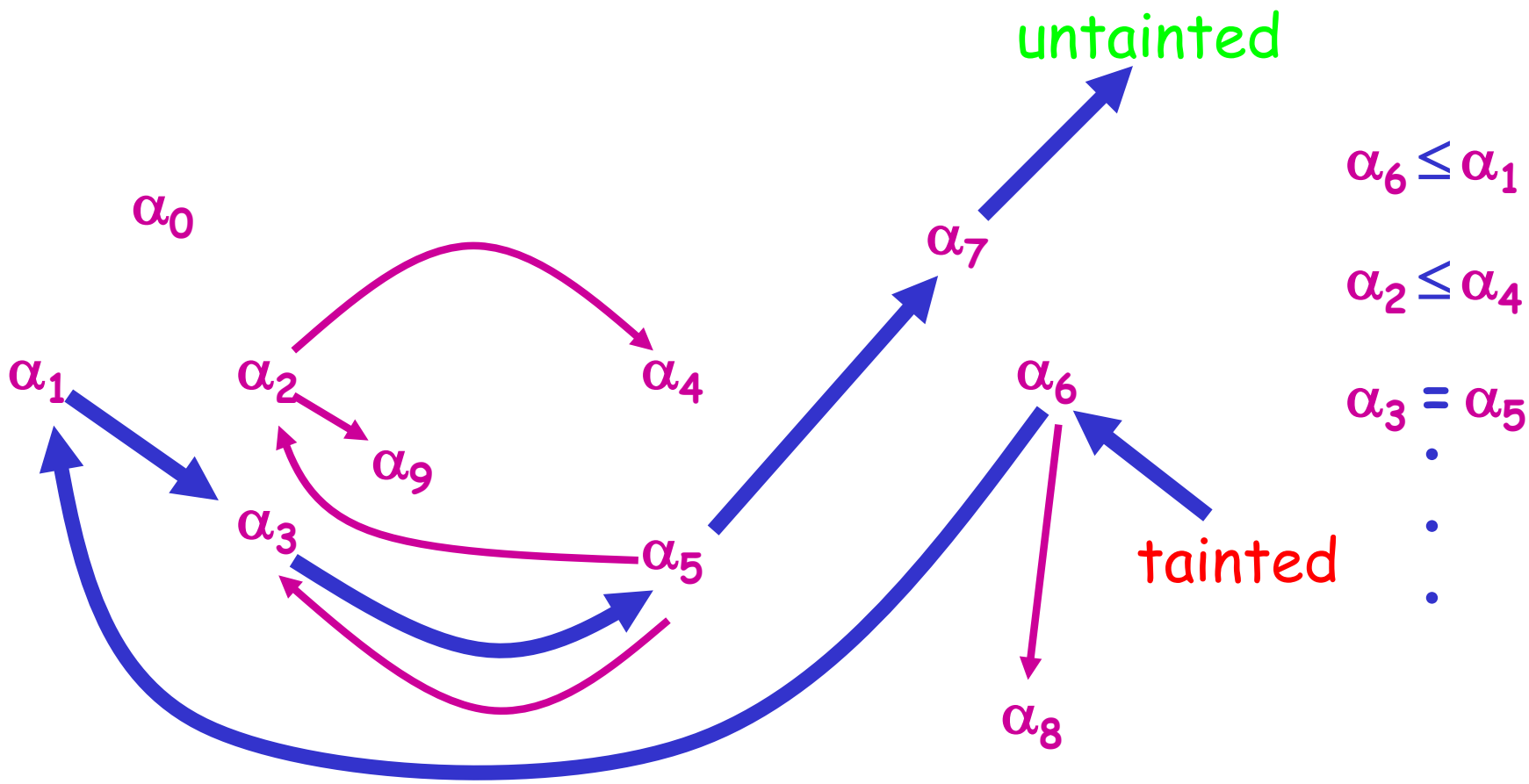
Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



Satisfiability via Graph Reachability

YES: tainted $\leq \alpha_6 \leq \alpha_1 \leq \alpha_3 \leq \alpha_5 \leq \alpha_7 \leq$ untainted



Satisfiability in Linear Time

- Initial program of size n
  - Fixed set of qualifiers tainted, untainted, …
- Constraint generation yields O(n) constraints
  - Recursive abstract syntax tree walk
- Graph reachability takes O(n) time
  - Works for semi-lattices, discrete p.o., products

Limitations of Subtyping

- Subtyping gives us a kind of *polymorphism*
  - A *polymorphic* type represents multiple types
  - In a subtyping system, qt represents qt and all of qt's subtypes
- As we saw, this flexibility helps make the analysis more precise
  - But it isn't always enough…

Limitations of Subtype Polymorphism

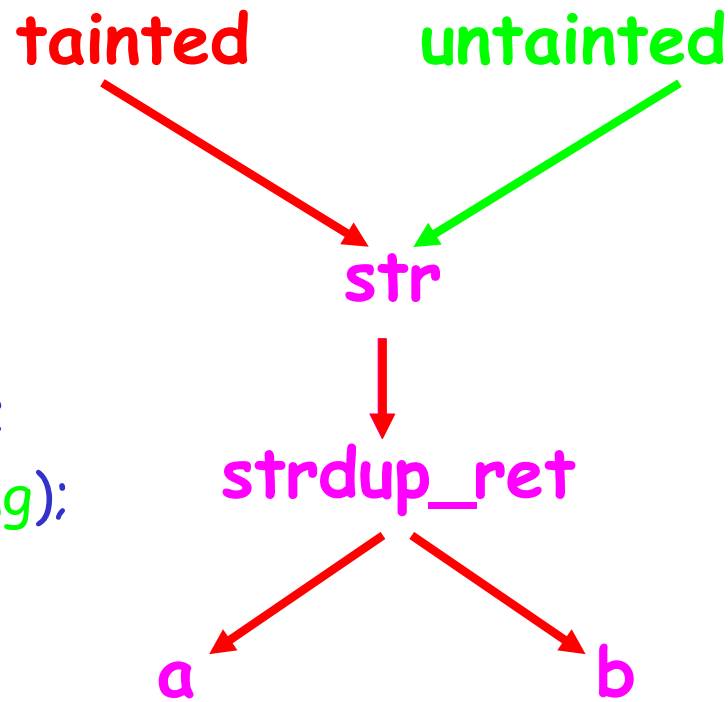
- Consider tainted and untainted again
  - untainted $\leq$ tainted
- Let's look at the identity function
  - fun id (x:int):int = x
- What qualified types can we infer for id?

Types for id

- fun id (x:int):int = x (ignoring int, qual on id)
  - tainted → tainted
    - Fine but untainted data passed in becomes tainted
  - untainted → untainted
    - Fine but can't pass in tainted data
  - untainted → tainted
    - Not too useful
  - tainted → untainted
    - Impossible

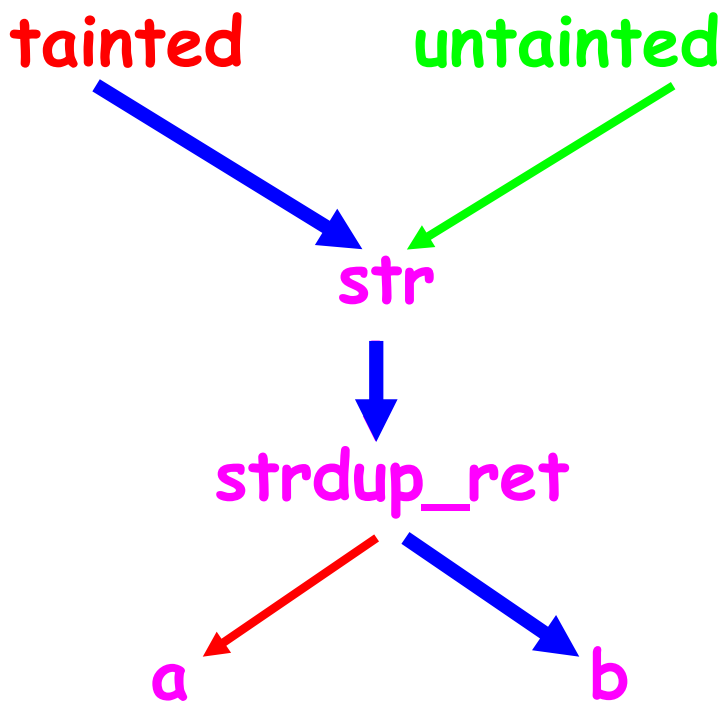
Function Calls and Context-Sensitivity

```
char *strdup(char *str) {
  // return a copy of str
}
char *a = strdup(tainted_string);
char *b = strdup(untainted_string);
```



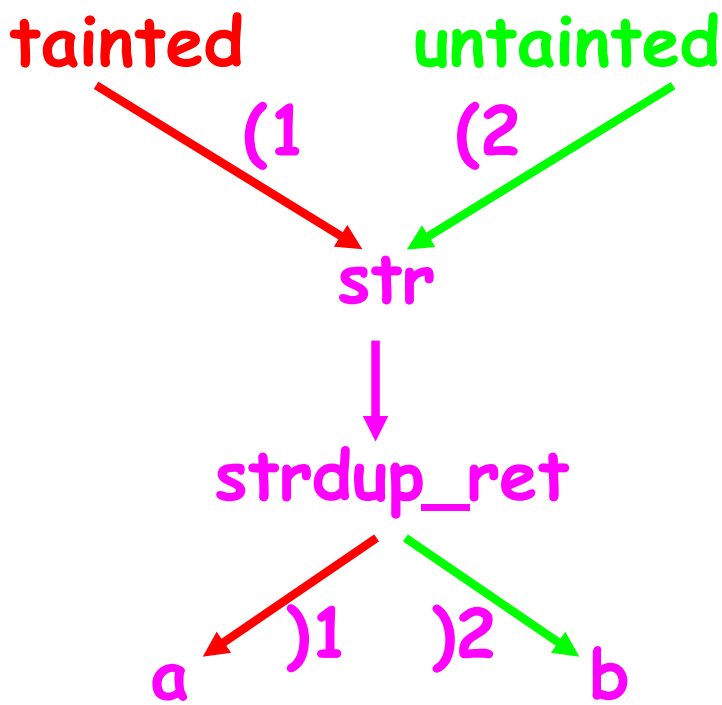
- All calls to strdup conflated
  - *Monomorphic* or *context-insensitive*

The Problem Restated: Unrealizable Paths



- No execution can exhibit that particular call/return sequence

Only Propagate Along Realizable Paths



- Add edge labels for calls and returns
  - Only propagate along *valid* paths whose returns balance calls

CFL Reachability

- We're trying to find paths through the graph whose edges are a language in some grammar
  - Called the *CFL Reachability* problem
  - Computable in cubic time

CFL Reachability Grammar

```
S ::= P N
P ::= M P
    |  )i P          for any i
    |                empty
N ::= M N
    |  (i N          for any i
    |                empty
M ::= (i M )i        for any i
    |  M M
    |  d             regular subtyping edge
    |                empty
```

- Paths may have **un**matched but not **mis**matched parens

Efficiency

- Constraint generation yields $O(n)$ constraints
  - Same as before
  - Important for scalability
- Context-free language reachability is $O(n^3)$
  - But a few tricks make it practical (not much slowdown in analysis times)
- For more details, see
  - Rehof + Fahndrich, POPL'01

Security via Type Qualifiers: The Icky Stuff in C

Experiment: Format String Vulnerabilities

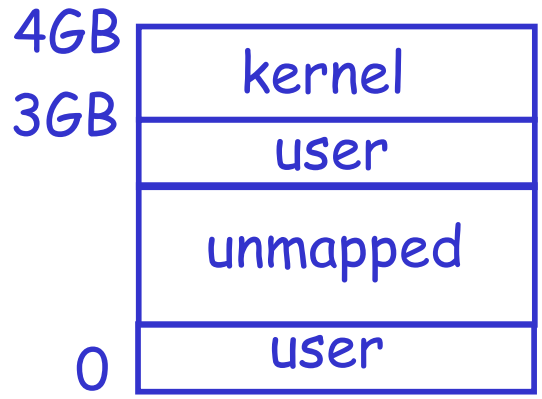
- Analyzed 10 popular unix daemon programs
  - Annotations shared across applications
    - One annotated header file for standard libraries
    - Includes annotations for polymorphism
      - Critical to practical usability
- Found several known vulnerabilities
  - Including ones we didn't know about
- User interface critical

Results: Format String Vulnerabilities

Name	Warn	Bugs
identd-1.0.0	0	0
mingetty-0.9.4	0	0
bftpd-1.0.11	1	1
muh-2.05d	2	~2
cfengine-1.5.4	5	3
imapd-4.7c	0	0
ipopd-4.7c	0	0
mars_nwe-0.99	0	0
apache-1.3.12	0	0
openssh-2.3.0p1	0	0

Experiment: User/kernel Vulnerabilities (Johnson + Wagner 04)

- In the Linux kernel, the kernel and user/mode programs share address space



  - The top 1GB is reserved for the kernel
  - When the kernel runs, it doesn't need to change VM mappings
    - Just enable access to top 1GB
    - When kernel returns, prevent access to top 1GB

Tradeoffs of This Memory Model

- Pros:
  - Not a lot of overhead
  - Kernel has direct access to user space

- Cons:
  - Leaves the door open to attacks from untrusted users
  - A pain for programmers to put in checks

An Attack

- Suppose we add two new system calls

```
int x;
void sys_setint(int *p) { memcpy(&x, p, sizeof(x)); }
void sys_getint(int *p) { memcpy(p, &x, sizeof(x)); }
```

- Suppose a user calls getint(buf)
  - Well-behaved program: buf points to user space
  - Malicious program: buf points to unmapped memory
  - Malicious program: buf points to kernel memory
    - We've just written to kernel space! Oops!

Another Attack

- Can we compromise security with setint(buf)?
  - What if buf points to private kernel data?
    - E.g., file buffers
  - Result can be read with getint

The Solution: copy_from_user, copy_to_user

- Our example should be written

```
int x;
void sys_setint(int *p) { copy_from_user(&x, p, sizeof(x)); }
void sys_getint(int *p) { copy_to_user(p, &x, sizeof(x)); }
```

- These perform the required safety checks
  - Return number of bytes that couldn't be copied
  - from_user pads destination with 0's if couldn't copy

It's Easy to Forget These

- Pointers to kernel and user space look the same
  - That's part of the point of the design
- Linux 2.4.20 has 129 syscalls with pointers to user space
  - All 129 of those need to use copy_from/to
  - The ioctl implementation passes user pointers to device drivers (without sanitizing them first)
- The result: Hundreds of copy_from/_to
  - One (small) kernel version: 389 from, 428 to
  - And there's no checking

User/Kernel Type Qualifiers

- We can use type qualifiers to distinguish the two kinds of pointers
  - kernel -- This pointer is under kernel control
  - user -- This pointer is under user control

- Subtyping kernel < user
  - It turns out copy_from/copy_to can accept pointers to kernel space where they expect pointers to user space

Type Signatures

- We add signatures for the appropriate fns:

```
int copy_from_user(void *kernel to,
                   void *user from, int len)
int memcpy(void *kernel to,
           void *kernel from, int len)
int x;
void sys_setint(int *user p) {
    copy_from_user(&x, p, sizeof(x)); }
void sys_getint(int *user p) {
    memcpy(p, &x, sizeof(x)); }
```

Lives in kernel

OK

OK

Error

Experimental Results

- Ran on two Linux kernels
  - 2.4.20 -- 11 bugs found
  - 2.4.23 -- 10 bugs found
- Needed to add 245 annotations
  - Copy_from/to, kmalloc, kfree, …
  - All Linux syscalls take user args (221 calls)
    - Could have be done automagically (All begin with sys_)
- Ran both single file (unsound) and whole-kernel
  - Disabled subtyping for single file analysis

More Detailed Results

- 2.4.20, full config, single file
  - 512 raw warnings, 275 unique, 7 exploitable bugs
    - Unique = combine msgs for user qual from same line
- 2.4.23, full config, single file
  - 571 raw warnings, 264 unique, 6 exploitable bugs
- 2.4.23, default config, single file
  - 171 raw warnings, 76 unique, 1 exploitable bug
- 2.4.23, default config, whole kernel
  - 227 raw warnings, 53 unique, 4 exploitable bugs

Observations

- Quite a few false positives
  - Large code base magnifies false positive rate
- Several bugs persisted through a few kernels
  - 8 bugs found in 2.4.23 that persisted to 2.5.63
  - An unsound tool, MECA, found 2 of 8 bugs
  - ==> Soundness matters!

Observations

- Of 11 bugs in 2.4.23…
  - 9 are in device drivers
  - Good place to look for bugs!
  - Note: errors found in "core" device drivers
    - (4 bugs in PCMCIA subsystem)
- Lots of churn between kernel versions
  - Between 2.4.20 and 2.4.23
    - 7 bugs fixed
    - 5 more introduced

Conclusion

- Type qualifiers are specifications that…
  - Programmers will accept
    - Lightweight
  - Scale to large programs
  - Solve many different problems

- General type qualifiers now available for Java
  - See work by Ernst et al on pluggable types
    - Applying to information flow properties