# Type Qualifiers

CS 6340

## Software Quality Today

Even after large, extensive testing efforts, commercial software is shipped riddled with errors ("bugs").

-- PITAC Report to the President, February 24, 1999

Trustworthy Computing is computing that is available, reliable, and secure as electricity, water services and telephony....No Trustworthy Computing platform exists today.

-- Bill Gates, January 15, 2002
(highest priority for Microsoft)

## Conclusion?

Software is buggy

## So What?

- Software has always been buggy
- But now...
  - More people use software
  - Computers keep getting faster
    - Speed/quality tradeoff changing
  - Cost of fixing bugs is high

## Common Techniques for Software Quality

- Testing
- Code auditing
- Drawbacks: Expensive, difficult, error-prone, limited assurances
- What more can we do?
  - Tools that analyze source code
  - Techniques for avoiding programming mistakes

## Tools Need Specifications

```
spin_lock_irqsave(&tty->read_lock, flags);
put_tty_queue_nolock(c, tty);
spin_unlock_irqrestore(&tty->read_lock, flags);
```

- Goal: Add specifications to programs

  In a way that...
  - Programmers will accept
    - Lightweight
  - Scales to large programs
  - Solves many different problems

## Type Qualifiers

- 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) | Security vulnerabilities |
| int → ptr(open FILE) | File operations |

## 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

- Lots of these bugs in the wild
  - New ones weekly on bugtraq mailing list
  - Too restrictive to forbid variable format strings

## 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

## 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;
f(b);
```

Error

g accepts only untainted data

tainted ≰ untainted

untainted < tainted

## Framework

- Pick some qualifiers
  - and relation (partial order) among qualifiers

untainted int < tainted int
readwrite FILE < read FILE

- Add a few explicit qualifiers to program

- Infer remaining qualifiers
  - and check consistency

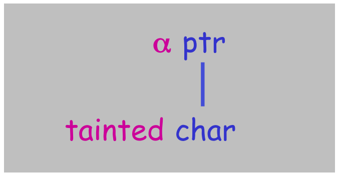
## Type Qualifier Inference

- Two kinds of qualifiers
  - Explicit qualifiers: tainted, untainted, ...
  - Unknown qualifiers: $\alpha_0, \alpha_1, \ldots$
- Program yields constraints on qualifiers

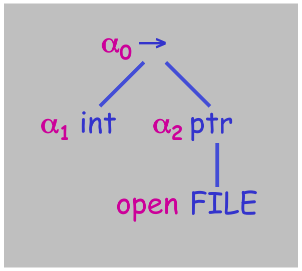
  tainted $\leq \alpha_0$ $\alpha_0 \leq$ untainted
- Solve constraints for unknown qualifiers
  - Error if no solution

## Adding Qualifiers to Types

ptr(tainted char)

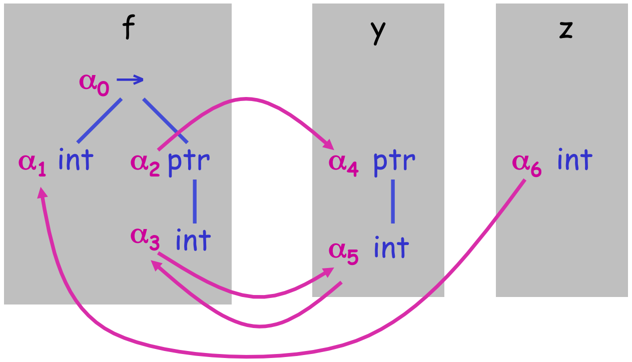


int → ptr( open FILE)



## Constraint Generation

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



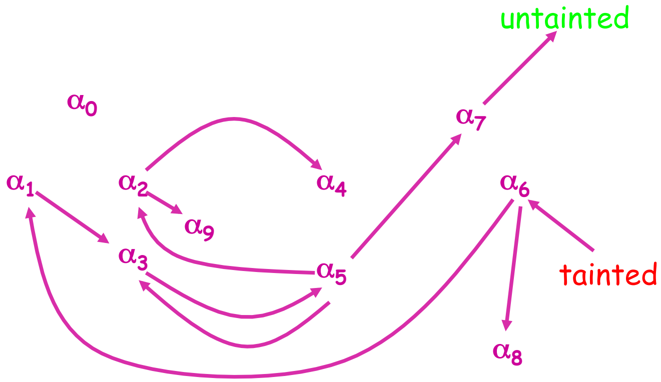
$\alpha_6 \leq \alpha_1$

$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

## Constraints as Graphs

Key idea: programs → constraints → graphs



$\alpha_6 \leq \alpha_1$

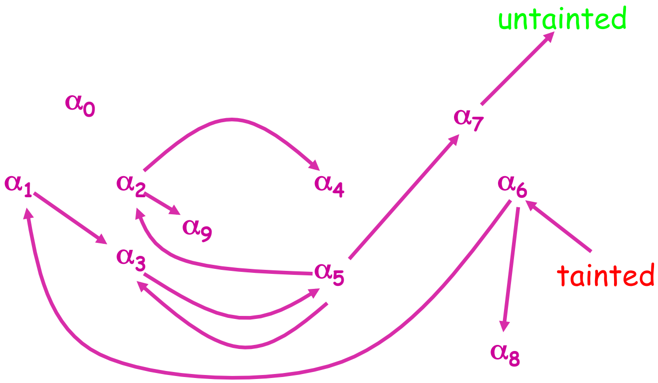
$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

⋮

## Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



$\alpha_6 \leq \alpha_1$

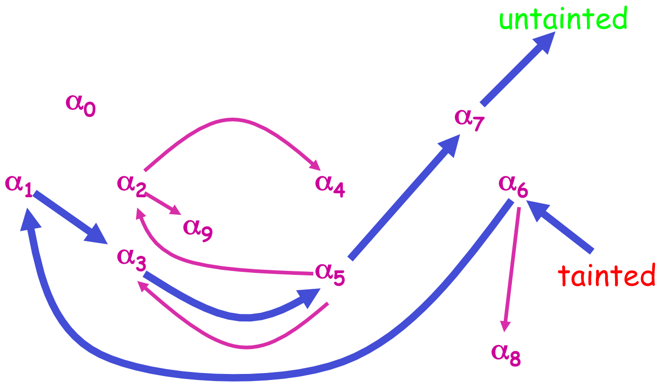
$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

$\vdots$

## Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



$\alpha_6 \leq \alpha_1$

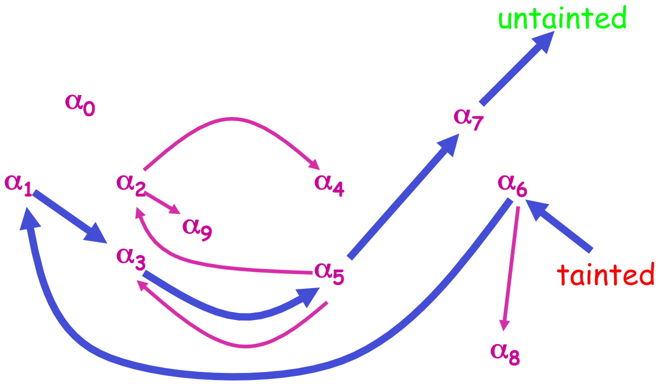
$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

$\vdots$

## Satisfiability via Graph Reachability

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



$\alpha_6 \leq \alpha_1$

$\alpha_2 \leq \alpha_4$

$\alpha_3 = \alpha_5$

$\vdots$

## Satisfiability in Linear Time

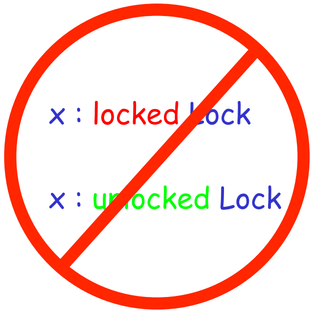
- Initial program of size n
  - Fixed set of qualifiers tainted, untainted, ...
- Constraint generation yields O(n) constraints
  - Recursive abstract syntax tree walk
- Graph reachability takes O(n) time
  - Works for semi-lattices, discrete p.o., products

## The Story So Far...

- Type qualifiers as subtyping system
  - Qualifiers live on the standard types
  - Programs → constraints → graphs
- Useful for a number of real-world problems
- Up next: State change and type qualifiers
  - A glimpse of a more complex system
- Followed by: Applications, experiments

## Application: Locking

```
Lock x;

lock(x);
...critical section...
unlock(x);
```


x : locked Lock

x : unlocked Lock

## Flow-Sensitive Type Qualifiers

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

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

- We need *flow-sensitivity*
  - Qualifiers may change during execution

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

## Some Challenges

- How do we deal with aliasing?

```
p = &x;    *p = ...;
```

- How do we make the analysis scale?
  - Too expensive to model full state at each point
- What happens when too much is aliased?
  - How does the programmer control aliasing?

## Modeling State with Abstract Stores

- Track each variable's type at each point
  - Abstract stores map variables to types
  - ...and types contain qualifiers

{ x : t,   y : r,   z : s, … }

x := …;

{ x : t',   y : r,   z : s, … }

y := …;

{ x : t',   y : r',   z : s, … }

## What About Aliasing?

- Suppose p points to x:

{ x : q int,   p : ptr(q int), … }

*p := …;

{ x : q int,   p : ptr(q' int), … }

  - Variable names alone are insufficient

- Solution: Add a level of indirection
  - Stores map *locations* to types
  - Pointer types point to locations

## Unification-Based Alias Analysis

- Initial flow-insensitive pass computes aliasing
  - Before flow-sensitive analysis
  - Simultaneous with standard type inference
    - *Types* are not flow-sensitive, only *qualifiers*

- Associate a location $\rho$ with each pointer
  - Unify locations that may alias

…   $*p : \mathrm{ptr}^{\rho}(\mathrm{int})$   $x : \mathrm{ptr}^{\overset{\rho}{\cancel{\sigma}}}(\mathrm{int})$

p = &x;   /* require $\rho = \sigma$ */

## Using Locations in Stores

- Suppose p points to x:

$*p : \mathrm{ptr}^{\rho}(\mathrm{int})$   $x : \mathrm{ptr}^{\rho}(\mathrm{int})$

{ $\rho$ : q int,   $\eta$ : ptr($\rho$), … }

*p := …;

{ $\rho$ : q' int,   $\eta$ : ptr($\rho$), … }

## What About Scalability?

- Stores are too big

  $\{\ \rho : t,\ \ \eta : r,\ \ \nu : s, \ldots\ \}$

  - A program of size $n$ may have
    - $n$ locations
    - $n$ program points
    - $\Rightarrow n^2$ space to represent stores
- We need a more compact representation
  - Idea: represent *differences* between stores

## Constructing Stores

- Three kinds of stores $S$

| | |
|---|---|
| $S ::= \varepsilon$ | Unknown store |
| $\mid\ \mathit{Alloc}(S, \rho : \tau)$ | Like store S, but $\rho$ allocated with type $\tau$ |
| $\mid\ \mathit{Assign}(S, \rho : \tau)$ | Like store S, but update type of $\rho$ with $\tau$ |

- Store constraints $S \leq \varepsilon$
  - Control flow from $S$ to $\varepsilon$
- Solution maps $\varepsilon$ to $\{\ \rho : t,\ \ \eta : r,\ \ \nu : s, \ldots\ \}$
  - Key: only write down necessary portion of soln.

## Example

```
Lock x;
while (...) {
    lock(x);
    y := ...
    unlock(x);
}
```

$\mathit{Alloc}(\varepsilon_0, \rho : \mathit{unlocked}) \leq \varepsilon_1$

$\mathit{Assn}(\mathit{Assn}(\mathit{Assn}(\varepsilon_1, \rho : \mathit{locked}), \eta : q\ \tau), \rho : \mathit{unlocked}) \leq \varepsilon_1$

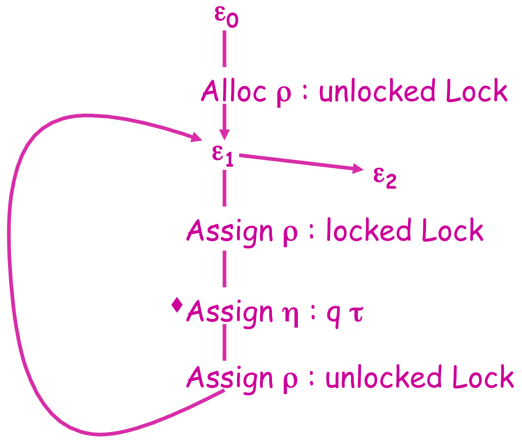
$\varepsilon_1 \leq \varepsilon_2$

## Example

```
Lock x^ρ;
while (...) {
    lock(x^ρ);
    y^η := ...
    unlock(x^ρ);
}
```

check $\varepsilon_1(\rho)$ : unlocked Lock

check ♦$(\rho)$ : locked Lock



## Example

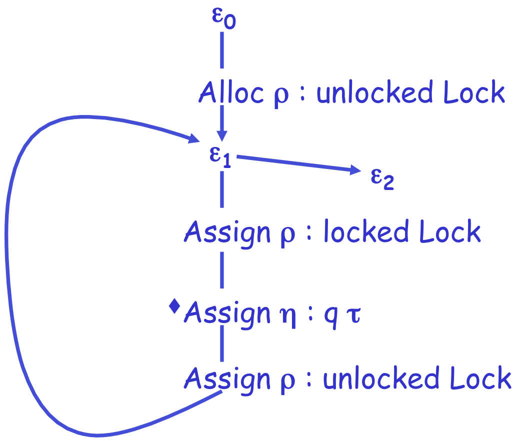
$\mathrm{Alloc}(\varepsilon_0, \rho : \mathrm{unlocked}) \le \varepsilon_1$

$\mathrm{Assn}(\mathrm{Assn}(\mathrm{Assn}(\varepsilon_1, \rho : \mathrm{locked}), \eta : q\ \tau), \rho : \mathrm{unlocked}) \le \varepsilon_1$

$\varepsilon_1 \le \varepsilon_2$

check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock



## Lazy Constraint Resolution

- We don't care about most locations
  - only those that may be locked or unlocked
  - In this case, we will only do work for ρ

- Key to efficiency:

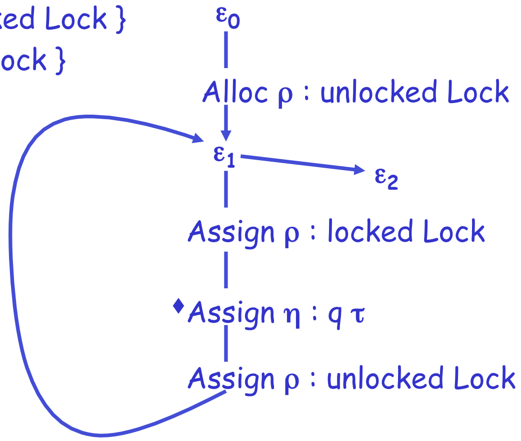
  When solving for store variables, only represent the minimum necessary

## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }

check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock

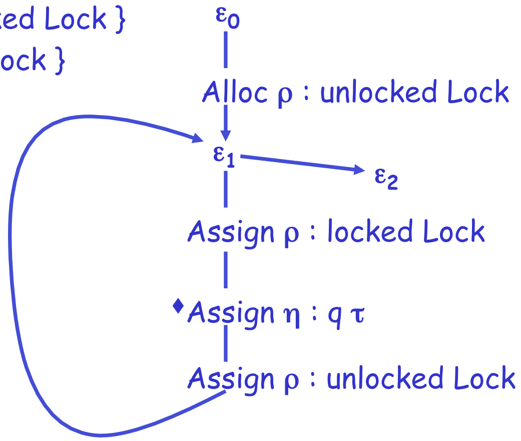


## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }

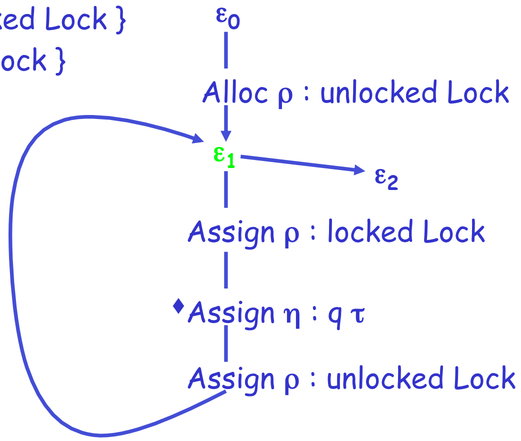
check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock



## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

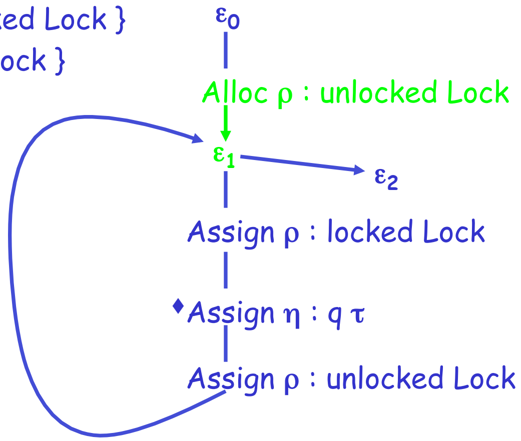
Assign ρ : unlocked Lock

check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

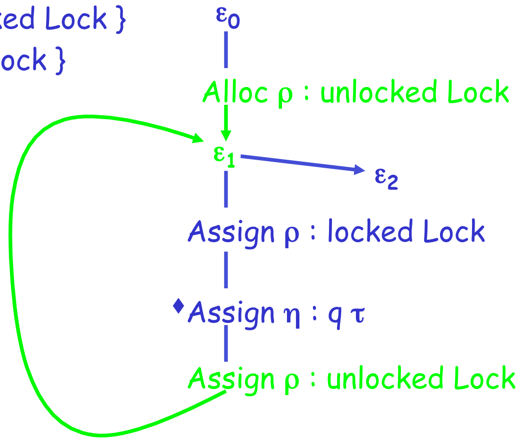
Assign ρ : unlocked Lock

check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

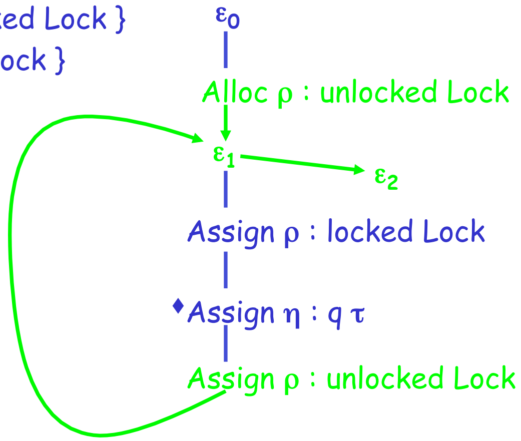
Assign ρ : unlocked Lock

check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }
red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

Assign ρ : unlocked Lock

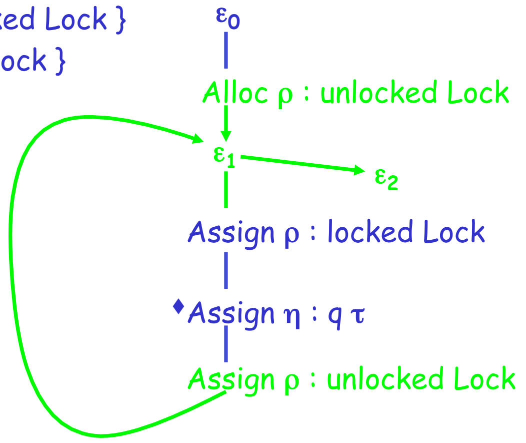
check $\varepsilon_1(\rho)$ :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }

red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

Assign ρ : unlocked Lock

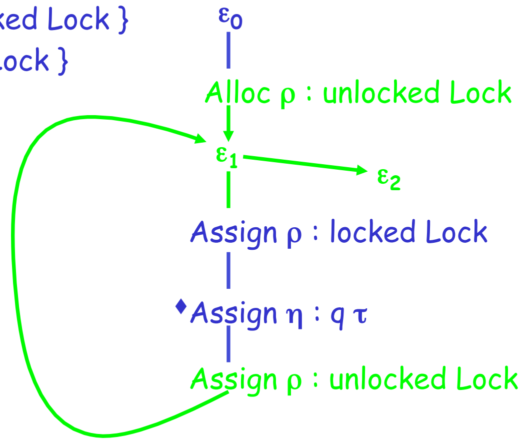
check $\varepsilon_1$(ρ) :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }

red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

Assign ρ : unlocked Lock

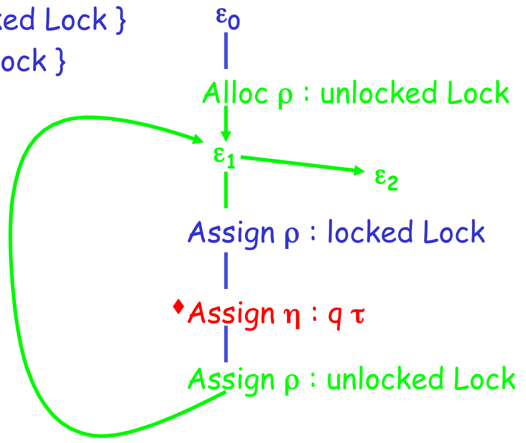
check $\varepsilon_1$(ρ) :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }

red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

Assign ρ : unlocked Lock

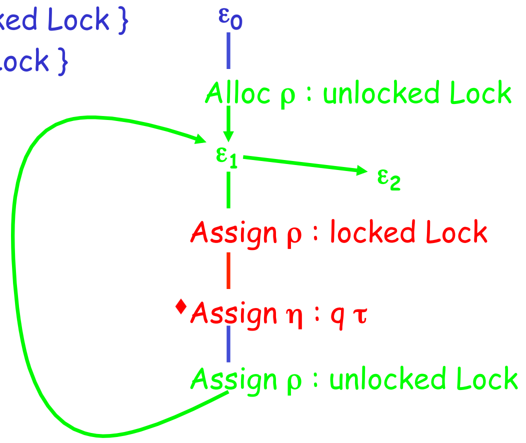
check $\varepsilon_1$(ρ) :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }

red = { ρ : locked Lock }



$\varepsilon_0$

Alloc ρ : unlocked Lock

$\varepsilon_1$

$\varepsilon_2$

Assign ρ : locked Lock

♦Assign η : q τ

Assign ρ : unlocked Lock

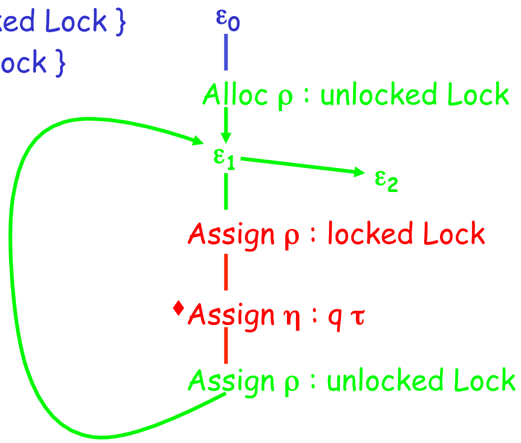
check $\varepsilon_1$(ρ) :
unlocked Lock

check ♦(ρ) :
locked Lock

## Constraint Resolution Example

green = { ρ : unlocked Lock }

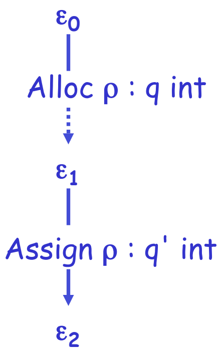
red = { ρ : locked Lock }



check $\varepsilon_1(\rho)$ :
unlocked Lock

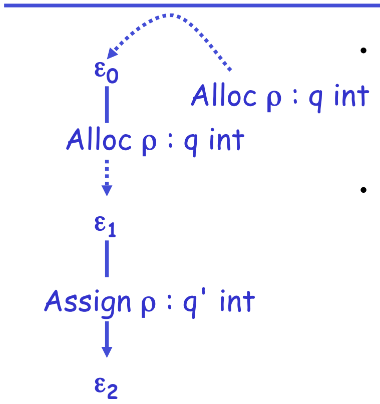
check ♦(ρ) :
locked Lock

## Strong Updates



- In $\varepsilon_2$, location ρ has qualifier q'
  - We've replaced ρ's qualifier
  - This is called a *strong update*
  - Location ρ is linear

## Weak Updates



- What if ρ allocated twice?
  - Only one is actually updated
- In $\varepsilon_2$, location ρ has qualifier q □ q'
  - We've merged ρ's new and old qualifiers
  - This is called a *weak update*
  - Location ρ is non-linear

## Recovering Linearity

- What do we do when aliasing too imprecise?
  - Can't strongly update non-linear locations
- New construct restrict
  - Programmer adds restrict to help the alias analysis
- restrict x = e1 in e2
  - Roughly: within e2, accesses to *e1 must use x

## Restrict Example

```
Lock locks[n];

lock(&locks[i]);
...
unlock(&locks[i]);
```

## Restrict Example

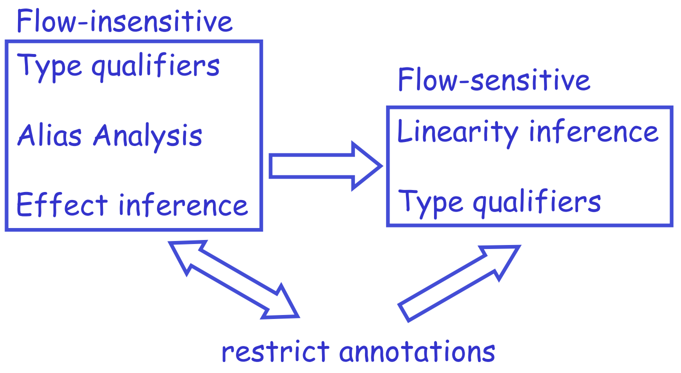
```
Lock locks[n];

restrict mylock = &locks[i] in
  lock(mylock);
   ...
  unlock(mylock);
```

- Within scope of restrict, only mylock used
  - Can perform strong updates
- After restrict ends, weak update from mylock to locks[]

## More Features

- Low-cost polymorphism
  - Use effects to avoid merging stores at fn calls
- Some path-sensitivity
  - Different types on if-then-else branches

## Qualifier Inference Architecture



## Applications

Published experiments:

- const Inference [Foster, Fahndrich, Aiken, PLDI99]
- Y2K bug detection [Elsman, Foster, Aiken, 1999]
- Format-string vulnerabilities [Shankar, Talwar, Foster, Wagner, Usenix Sec 01]
- Locking and stream operations [Foster, Terauchi, Aiken, PLDI 02]
- Linux Security Modules (IBM Watson) [Zhang, Edwards, Jaeger, Usenix Sec 02]

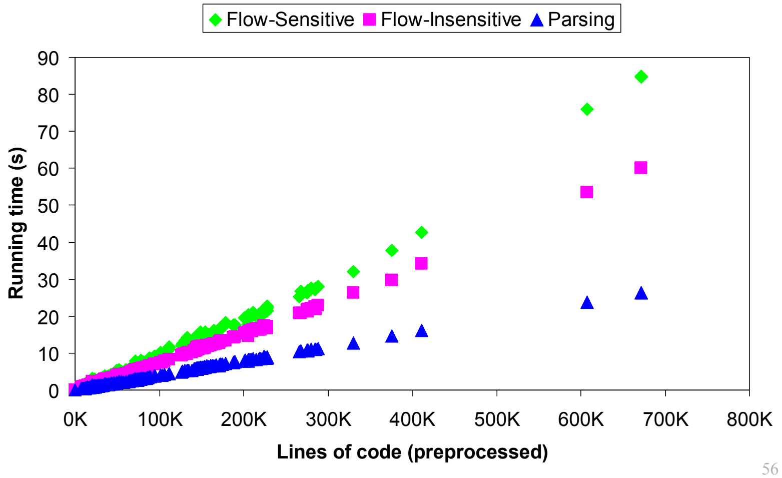
## Results: Format String Vulnerabilities

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

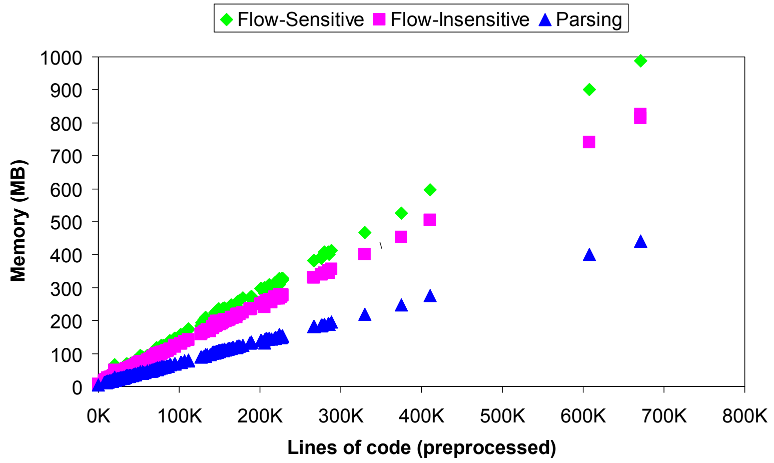
## Results: Locking

- Looked for simple deadlocks in Linux 2.4.9
  - Double acquires/releases
- Analyzed 892 files in linux/drivers individually
- Analyzed 513 modules (all linked files)
  - 14 type errors ⇒ deadlocks
  - ~41/892 fail to typecheck but appear correct
  - ~196/513 fail to typecheck
    - added restrict by hand to remove type errors due to aliasing for 64/196

## Running Time: Locking



## Memory Usage:  Locking



## Main Contributions

- Type qualifiers as specifications
  - With applications
- Scalable flow-sensitive qualifier inference
  - Lazy, constraint-based
  - Built with alias analysis, effect inference
  - Linearities for strong/weak updates
- restrict construct

## (Some) Related Work

- Dataflow Analysis
- Bug-finding Tools
  - AST Toolkit [Weise, Crew]
  - Meta-Level Compilation [Engler et al]
- Type Systems
  - Label flow [Mossin]
  - Typestate [Strom, Yemini, Yellin]
  - Vault [Fähndrich, DeLine]
  - Cyclone [Grossman et al]

## Conclusion

- Type qualifiers are specifications that...
  - Programmers will accept
    - Lightweight
    - Easy to use -- inference and visualization
  - Scale to large programs
  - Solve many different problems

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

Includes source code and web demo of cqual