Type Qualifiers

CS195
Lecture 14

Motivation

The Problem

Software is buggy

The Solution

Find the bugs, and find them sooner

Idea

Build tools to do this!

A Good Idea

<table>
<thead>
<tr>
<th>MetaCompilation</th>
<th>Older</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford</td>
<td>LClint</td>
</tr>
<tr>
<td></td>
<td>AST</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SLAM</th>
<th>Toolkit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>NIL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CQual</th>
<th>Newer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>BLAST</td>
</tr>
</tbody>
</table>

|                 | ESP    |

This Lecture

- Type-based approaches
- Goal: Prevent bugs
  - Different from finding bugs after the fact
- Philosophy
  - Some programmer intervention is desirable and necessary
  - Full automation is hopeless

Specifications

- Typically, check specifications vs. code
  - But most programs are specification-poor

- Get more specifications into programs...
  - In a way programmers will accept
  - In a way that scales to large programs
  - And enables useful new checking for correctness

Idea: Extend Standard Type Systems

- What is the smallest interesting extension?

- Add new type qualifiers
  
  const int
  locked spinlock_t
  open FILE *
Type Qualifiers

- Already in mainstream languages
  - Special cases
- Opens up the type system
  - User-definable type qualifiers
- Expressive
  - Lots of control properties can be written this way

Framework

- Pick some qualifiers
  - and a partial order on them
- Add a few qualifier annotations to a program

This is a Good Idea

- Evidence
  - CQual
  - Project at Berkeley
  - Involving yours truly
- But more convincing
  - Sparse
  - A checker for user/kernel qualifiers in Linux
  - By Linus Torvalds

Sparse

- Two qualifiers
  - user
  - kernel
- Programmer declares the types of pointers
  \[ \text{foo(user int }^*x) \]
- Sparse restricts uses of user pointers
  - Assignment to another user pointer \( y = x \)
  - \text{copy\_from\_user}(x)
  - Nothing else is allowed

Good News and Bad News

- Good news
  - This works
  - Though not quite as simple as suggested
    - Lots of details to handle in C
    - Casting is yucky
- Bad news
  - This is a lot of work for programmers
  - Since qualifiers must match in all uses, programmers must add many qualifiers

A Quote About Sparse

"Most people... actually like how it forces you to make it very _explicit_ whether you’re using a user pointer or not. But I have to admit that I’ve grown tired of trying [to check] which sparse warnings are valid and which aren’t."

- Linus Torvalds, May 2003
Qualifier Inference

- Type (qualifier) checking is tedious and boring
  - Programmer must supply the types
  - Tool simply confirms the types are correct
- Much better is type (qualifier) inference
  - Programmer supplies little or no explicit types
  - Tool infers the types (qualifiers) itself
  - Type errors => possible bugs in program
  - No type errors => correct program

Application: Format String Vulnerabilities

- Adversary-controlled format strings
  - $s = \text{data from network}$
  - `printf($s)`
- Lots of these bugs in the wild

Tainting Analysis

- Qualifiers: tainted and untainted
  - tainted may be controlled by adversary
  - untainted = must not be controlled by adversary
  - untainted = tainted
- Annotate functions
  - `printf(untainted char *fmt,...)`
  - `tainted char *getenv(const char *)`
- Type error = possible format string bug

Types

- Standard types
  - $\tau ::= \text{int} | \text{ptr}()$ $\tau \rightarrow \tau$
- Qualified types
  - $\tau ::= \text{Q} \sigma$
  - $\sigma ::= \text{int} | \text{ptr}()$ $\tau \rightarrow \tau$

Type Rules

- Standard rule:
  $e_1 : \tau \rightarrow \tau' \quad e_2 : \tau$
  $e_1(e_2) : \tau$
- Qualified rule:
  $e_1 : \text{Q} (\tau \rightarrow \tau') \quad e_2 : \tau' \quad \tau' \leq \tau$
  $e_1(e_2) : \tau'$

Qualified Type Inference

- Two-pass algorithm
  - Collect $\epsilon \leq \epsilon$ constraints
  - Solve the constraints
- Are the constraints consistent?
  - Inconsistent constraints => possible bug
    - tainted $\leq a_0 \leq \ldots \leq a_n \leq$ untainted
Types are Trees

int

Types are Trees

ptr(int) or int *

ptr

int

Types are Trees

int → ptr(int)

→

int  ptr

int

Qualifiers decorate every level of the tree

Types are Trees

Q₄((Q₃ int) → Q₂ ptr(Q₄ int)) ≤ Q₃ ((Q₂ int) → Q₁ ptr(Q₃ int))

Q₂ →

Q₃ int

Q₄ ptr

Q₅ int

Q₆ →

Q₄ ptr

Q₅ int

Q₆ int

Constraint Resolution as Graph Closure

Q₄ ((Q₃ int) → Q₂ ptr(Q₄ int)) ≤ Q₃ ((Q₂ int) → Q₁ ptr(Q₃ int))

⇒ (Q₄ int ≤ Q₃ int) ∧ (Q₄ ptr(Q₄ int) ≤ Q₃ ptr(Q₃ int)) ∧ Q₃ ≤ Q₄
**Constraints as Graphs**

\[ F(x\text{int} \rightarrow \text{ptr(int)}) = \{ \ldots \} \]

\[ \{ \ldots F(y) \ldots \} \]

**Satisfiability via Graph Reachability**

Do constant lower bounds reach consistent upper bounds?

**Satisfiability in Linear Time**

At joints take upper bound in lattice order \(
\text{constant size information per node} \)

**Visualization**

Arrows in constraint graph bewreated through user interface

**Part Way There**

- So far, so good
  - Useful for finding bugs
    - Tainting bugs and others
  - But insufficient to show correctness
- The big problem is aliasing
  - Present one solution
    - This is more speculative than type qualifiers alone
Locking

spin_lock(f-a[i]);
  do_work();
spin_unlock(f-a[i]);

Locked and Unlocked Qualifiers

spin_lock(f-a[i]);
  do_work();
spin_unlock(f-a[i]);

spin_lock: unlocked lock -> locked lock
spin_unlock: locked lock -> unlocked lock

The Example Again

f-a[i] = b[k]?

spin_lock(f-a[i]);
  do_work();
spin_unlock(f-a[i]);

f = h?

Alias Queries

- f-a = b?
- f = h?

- These are alias queries
  - Can two pointers refer to the same object?
- An alias analysis A answers alias queries

A(expression1, expression2) = No or I don’t know

One-Slide Summary of Alias Analysis

- Enormous progress in recent years
  - New techniques enable scalable alias analysis
  - Several techniques analyze 1M line programs in seconds

- But
  - Conservative
  - Size of alias sets grows with program size

The Example Yet Again

spin_lock(f-a[i]);
  do_work();
spin_unlock(f-a[i]);
Aliasing Specifications

- Need programmer intervention
  - This problem has arisen before
  - Solution allow declaration of non-dating information
  - An unchecked pragma

- FORTRAN
  - SUBROUTINE Matmul(A,B)
  - May assume A,B are undated

- C
  - int *foo(int *restrict x)
  - May assume x is undated
  - Promise not to use aliases of x in foo

Restrict

- Recall let \( x = e_1 \) in \( e_2 \)
  - New local variable \( x \), initialized to \( e_2 \), scope is \( e_2 \)

- Introduce restrict \( x = e_1 \) in \( e_2 \)

- Restricts use of aliases of \( x \)
  - OK to use \( x \) and aliases created from \( x \) in \( e_2 \)
  - Error to use \( e_2 \) in other aliases of \( e_2 \)

Problem: Can we verify restrict?

Examples

```
restrict p = q in {
    *p: /*OK*/
    *q: /*Error*/
}

restrict p = q in {
    *p: /*OK*/
    *q: /*OK*/

    restrict r = p in {
        *p: /*OK*/
    }

    restrict p = q in {
        *q: /*Error*/
    }
}
```

Checking Restrict

- Need to track use of (abstract) memory locations
  - So we can check whether certain locations are used or not
  - Assign each pointer an abstract location \( p \)
  - Record locations used by an expression in an abstract set

\[
e_1 : \text{ptr}(\cdot) : L_1 \quad e_2 : \text{ptr}(\cdot) : L_2 \quad e_3 : \text{ptr}(\cdot) : L_3
\]

Let Rule

```
e_1 : \text{ptr}(\cdot) : L_1
x : \text{ptr}(\cdot) : e_2 : \tau_2 : L_2
\]

let \( x = e_1 \) in \( e_2 \) : \( \tau_2 : L_1 \cup L_2 \)
```

Restrict Rule

\[
e_1 : \text{ptr}(\cdot) : L_1
x : \text{ptr}(\cdot) : e_2 : \tau_2 : L_2
\rho \in L_2
\rho' \in \tau_2
\]

restrict \( x = e_1 \) in \( e_2 \) : \( \tau_2 : L_1 \cup L_2 \)

Note we have two names for the different alias sets; this is the advantage over standard alias analysis.
Experience with Restrict Checking

- Experiment with 513 device drivers
  - locked/unlocked qualifier inference
  - 307 passed
  - Added restrict by hand to 64/196 others
- All 64 passed with hand annotation

Conclusion
- This works
- But it is a lot of work
- Can we do better?

An Idea

- Instead of adding restrict by hand, could the tool do it automatically?
- Could we do restrict inference?

Let and Restrict Side-by-Side

\[
\begin{align*}
e_1: \text{ptr}(t); L_1 & \quad x: \text{ptr}(t) \leftarrow e_2 : \tau_2; L_2 \quad \text{let } x = e_1 \text{ in } e_2 : \tau_2; L_1 \cup L_2 \\
e_1: \text{ptr}(t); L_1 & \quad x: \text{ptr}(t) \leftarrow e_2 : \tau_2; L_2 \quad \text{restrict } x = e_1 \text{ in } e_2 : \tau_2; L_1 \cup L_2
\end{align*}
\]

Let and Restrict: A Bit More Alike

\[
\begin{align*}
e_1: \text{ptr}(t); L_1 & \quad x: \text{ptr}(t) \leftarrow e_2 : \tau_2; L_2 \quad p \equiv p' \quad \text{let } x = e_1 \text{ in } e_2 : \tau_2; L_1 \cup L_2 \\
e_1: \text{ptr}(t); L_1 & \quad x: \text{ptr}(t) \leftarrow e_2 : \tau_2; L_2 \quad p \equiv p' \quad \text{restrict } x = e_1 \text{ in } e_2 : \tau_2; L_1 \cup L_2
\end{align*}
\]

Combined Rule

\[
\begin{align*}
e_1: \text{ptr}(t); L_1 & \quad x: \text{ptr}(t) \leftarrow e_2 : \tau_2; L_2 \quad p \equiv p' \quad \text{let-restrict } x = e_1 \text{ in } e_2 : \tau_2; L_1 \cup L_2
\end{align*}
\]

Collect all constraints, solve for the least solution, which restrict's the maximum # of variables
### One More Step

- Recall example
  ```c
  spin_lock();
  do_work();
  spin_unlock();
  ```
- Need a version of `restrict` for expressions
  - called `confine`
- `restrict doesn't help!`
  - `&-[]` is an expression
  - `restrict` works on variables
- `restrict the location`
  - Locations used in accessing restricted location must be read-only
  - Tracking read-only an exercise in effects and constraints

### Experiment

- Same 513 device drivers
- Tried to confine all `e` in `spin_lock(e)`
- What scope do we confine?
  - Many possible choices, want largest possible
  - Solution: Infer scope
  - Use constraints again to express all possibilities

### Confinement Inference: Before

- Graph showing line of code vs running time before confinement

### Confinement Inference: After

- Graph showing line of code vs running time after confinement

### Confinement Inference

- Makes a huge difference
  - Hand annotation time drops from hours to minutes for worst device driver
- Why does it sometimes fail?
  - Something other than `spin_lock(e)` needs confining
  - Analysis is still conservative

### Running Time: Locking

- Graph showing lines of code vs running time for locking operations
Summary

- Verification is possible
  - At least for limited properties

- Specifications needed from programmer
  - Few in practice
  - Puts documentation in the code

- Two forms of specification useful:
  - Type qualifiers
  - Aliasing declarations
  - Programmer ultimately needs control