Security via Type Qualifiers

Jeff Foster
University of Maryland

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

Introduction

- Ensuring that software is secure is hard
- Standard practice for software quality:
  - Testing
    - Make sure program runs correctly on set of inputs
  - Code auditing
    - Convince yourself and others that your code is correct

Drawbacks to Standard Approaches

- Difficult
- Expensive
- Incomplete
- A malicious adversary is trying to exploit anything you miss!

Tools for Security

- What more can we do?
  - Build tools that analyze source code
    - Reason about all possible runs of the program
  - Check limited but very useful properties
    - Eliminate categories of errors
    - Let people concentrate on the deep reasoning
  - Develop programming models
    - Avoid mistakes in the first place
    - Encourage programmers to think about security

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)	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 := &lt;data-from-network&gt;
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
- 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);
	void g(untainted int);
	tainted int b;
	f(b);

OK	Error
f accepts tainted or untainted data	g accepts only untainted data
untainted ≤ tainted	tainted ≰ untainted

untainted < tainted

Demo of cqual

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

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 $\to^Q$ 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):$int^{tainted}$ = …$42^{tainted}$
  - fun printf (x:qt):qt' = check(untainted, x), …

Typing Rules: Qualifier Introduction

- Newly-constructed values have "bare" types

$$\frac{}{G \mid\text{--}\ n : int}$$

$$\frac{}{G \mid\text{--}\ true : bool} \qquad \frac{}{G \mid\text{--}\ false : bool}$$

- Annotation adds an outermost qualifier

$$\frac{G \mid\text{--}\ e1 : s}{G \mid\text{--}\ annot(Q, e) : Q\ s}$$

Typing Rules: Qualifier Elimination

- By default, discard qualifier at destructors

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

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

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

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?

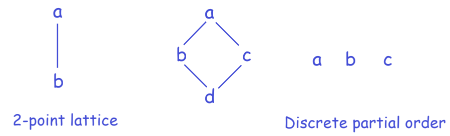
Partial Orders

- 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

Example Partial Orders

a
|
b

2-point lattice



a b c

Discrete partial order

- Lower in picture = lower in partial order
- Edges show $\leq$ relations

Combining Partial Orders

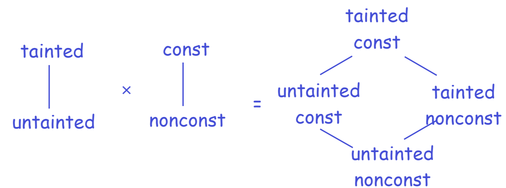
- Let $(Q_1, \leq_1)$ and $(Q_2, \leq_2)$ be partial orders
- We can form a new partial order, their cross-product:

$$(Q_1, \leq_1) \times (Q_2, \leq_2) = (Q, \leq)$$

where
  - $Q = Q_1 \times Q_2$
  - $(a, b) \leq (c, d)$ if $a \leq_1 c$ and $b \leq_2 d$

Example

tainted | untainted × const | nonconst =



- Makes sense with orthogonal sets of qualifiers
  - Allows us to write type rules assuming only one set of qualifiers

Extending the Qualifier Order to Types

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

- Add one new rule *subsumption* to type system

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

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

Use of Subsumption

$$\frac{\dfrac{\dfrac{|\text{-- } 42 : \text{int}}{|\text{-- annot(untainted, 42) : untainted int}} \quad \text{untainted} \leq \text{tainted}}{|\text{-- annot(untainted, 42) : tainted int}}}{|\text{-- check(tainted, annot(untainted, 42)) : tainted int}}$$

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:

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

  - 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)

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 ::= ... | ref^Q qt
    - Note: In ML the ref appears on the right

$$\frac{G \mid\!-\!- e : qt}{G \mid\!-\!- ref^Q\ e : ref^Q\ qt}$$

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

Dynamic Semantics and Copying

- Our previous semantics implemented everything with substitution and copying
  - This works OK for values...

(fun f x = x + x) 3 →

3 + 3 →

6

Dynamic Semantics and Copying

- But it doesn't make sense for references...

(fun f x = (x:=3); !x + !x) (ref 4) →

(ref 4) := 3; !(ref 4) + !(ref 4) →

8??

The Solution

- Add a level of indirection to semantics
  - Add locations to set of values
    - v ::= … | loc
  - A *store* S is a mapping from locations to values
  - New reduction relation $\langle S, e\rangle \rightarrow \langle S', e'\rangle$
    - In initial store S, expression e evaluates to e', resulting in new store S'

Adding Stores to Old Rules

- Most rules just carry the stores along

$$\frac{}{\langle S, \text{if true}^{Q} \text{ then e1 else e2}\rangle \rightarrow \langle S, \text{e1}\rangle}$$

- Ordering rules need to thread the store

$$\frac{\langle S, \text{e1}\rangle \rightarrow \langle S', \text{e1}'\rangle}{\langle S, \text{e1 e2}\rangle \rightarrow \langle S', \text{e1}' \text{ e2}\rangle}$$

Dynamic Semantics for Allocation

$$\frac{\langle S, e\rangle \rightarrow \langle S', e'\rangle}{\langle S, \text{ref } e\rangle \rightarrow \langle S', \text{ref } e'\rangle}$$

$$\frac{\text{loc fresh in } S}{\langle S, \text{ref } v\rangle \rightarrow \langle S[\text{loc}:v], \text{loc}\rangle}$$

Dynamic Semantics for Dereference

$$\frac{\langle S, e\rangle \rightarrow \langle S', e'\rangle}{\langle S, !e\rangle \rightarrow \langle S', !e'\rangle}$$

$$\frac{\text{loc in } S}{\langle S, !\text{loc}\rangle \rightarrow \langle S, S(\text{loc})\rangle}$$

Dynamic Semantics for Assignment

$$\frac{\langle S, \text{e1}\rangle \rightarrow \langle S', \text{e1}'\rangle}{\langle S, \text{e1}:=\text{e2}\rangle \rightarrow \langle S', \text{e1}:=\text{e2}\rangle} \qquad \frac{\langle S, \text{e2}\rangle \rightarrow \langle S', \text{e2}'\rangle}{\langle S, \text{loc}:=\text{e2}\rangle \rightarrow \langle S', \text{loc}:=\text{e2}\rangle}$$

$$\frac{\text{loc in } S}{\langle S, \text{loc}:=v\rangle \rightarrow \langle S[\text{loc}:v], v\rangle}$$

Subtyping References

- The *wrong* rule for subtyping references is

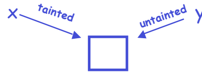
$$\frac{Q \le Q' \quad qt \le qt'}{\text{ref}^{Q}\, qt \le \text{ref}^{Q'}\, qt'}$$

- Counterexample

  let x = ref $0^{\text{untainted}}$ in
  let y = x in
  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 → qt
    - set : qt → unit
  - Notice that qt occurs both co- and contravariantly
- The right rule:

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

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?
- We want an algorithm to solve this problem

Type Checking vs. Type Inference

- Let's think about C's type system
  - C requires programmers to annotate function types
  - …but not other places
    - E.g., when you write down 3 + 4, you don't need to give that a type
  - So all type systems trade off programmer annotations vs. computed information
- Type checking = it's "obvious" how to check
- Type inference = it's "more work" to check

Why Do We Want Qualifier Inference?

- Because our programs weren't written with qualifiers in mind
  - They don't have qualifiers in their type annotations
  - In particular, functions don't list qualifiers for their arguments
- Because it's less work for the programmer
  - …but it's harder to understand when a program doesn't type check