

Flow-Insensitive Type Qualifiers¹

JEFFREY S. FOSTER

University of Maryland, College Park

and

ROBERT JOHNSON and JOHN KODUMAL

University of California, Berkeley

and

ALEX AIKEN

Stanford University

We describe flow-insensitive type qualifiers, a lightweight, practical mechanism for specifying and checking properties not captured by traditional type systems. We present a framework for adding new, user-specified type qualifiers to programming languages with static type systems, such as C and Java. In our system, programmers add a few type qualifier annotations to their program, and automatic type qualifier inference determines the remaining qualifiers and checks the annotations for consistency. We describe a tool CQUAL for adding type qualifiers to the C programming language. Our tool CQUAL includes a visualization component for displaying browsable inference results to the programmer. Finally, we present several experiments using our tool, including inferring *const* qualifiers, finding security vulnerabilities in several popular C programs, and checking initialization data usage in the Linux kernel. Our results suggest that inference and visualization make type qualifiers lightweight, that type qualifier inference scales to large programs, and that type qualifiers are applicable to a wide variety of problems.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications; D.2.4 [Software Engineering]: Software/Program Verification; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Design, Reliability, Experimentation, Languages, Theory, Verification

Additional Key Words and Phrases: Type qualifiers, types, security, constraints, *const*, *taint*, static analysis

1. INTRODUCTION

Software continues to increase in size and complexity, yet our ability to ensure its quality lags behind. Well-publicized software glitches have led to failures such as the Mars Climate Orbiter crash [Mars Climate Orbiter Mishap Investigation Board 1999], and security vulnerabilities in software have paved the way for attacks such

¹To appear in Transactions on Programming Languages and Systems. This is a DRAFT version before copy editing.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

as the Code Red Worm [CERT 2001]. The potentially staggering cost of software quality problems [NIST 2002] has led to a renewed call to increase the safety, reliability, and maintainability of software [Gates 2002; PITAC 1999].

In this paper, we propose using *type qualifiers* to improve software quality. Type qualifiers are a lightweight, practical mechanism for specifying and checking properties not captured by traditional type systems. A type qualifier is an atomic property that “qualifies” the standard types (examples below). Many programming languages have a few special-purpose type qualifiers. In contrast, we have developed a general framework for adding new, user-specified qualifiers to languages with static type systems, such as C and Java. In our framework the programmer adds a few key qualifier annotations, and then the system performs *type qualifier inference* to automatically infer the remaining qualifiers and check the consistency of qualifier annotations.

As one example, we can use type qualifiers to detect potential security vulnerabilities (Section 5.2). Security-conscious programs need to distinguish untrusted values read from the network from trusted values the program itself creates. We can model this property by using qualifiers *tainted* and *untainted* to mark the types of untrusted and trusted data, respectively. Type qualifier errors occur when a value of type *tainted T* is used where a value of type *untainted T* is expected, where *T* is a standard unqualified type such as *string*. Any such type qualifier error indicates a potential security vulnerability.

In addition to studying *tainted* and *untainted*, we explore two other qualifiers in this paper. In Section 5.1 we study the ANSI C qualifier *const*, which is used to mark *l*-values that cannot be updated [ANSI 1999]. In Section 5.3, we treat a special annotation `__init` in the Linux kernel as a type qualifier and use our system to check that it is used correctly. Although these three (*tainted* and *untainted*, *const*, and `__init`) are the focus of the experiments in this paper, our system can be used for other qualifiers, for example, *user* and *kernel* [Johnson and Wagner 2004], or *sensitive* and *unsensitive* [Broadwell et al. 2003].

Type qualifiers have a number of advantages as a mechanism for specifying and checking properties of programs:

- Of the multitude of proposals for statically-checked program annotations, types are arguably the most successful. In many languages, programmers must already include type annotations in their source code. Thus the machinery of types is familiar to the programmer, and we believe it is natural for a programmer to specify additional properties with a type qualifier. This bodes well for the adoption of type qualifiers in practice, since a key concern about any specification language is whether programmers are willing to use it.
- Type qualifiers are additional annotations layered on top of the standard types. As such, they can be safely ignored by conventional tools (such as standard compilers) that do not understand them. This natural backward compatibility lowers the barrier to adopting type qualifiers.
- Static type qualifier systems conservatively model all runs of a program. This is especially valuable for finding bugs that are hard to replicate and for finding security vulnerabilities, and both are exactly the problems that are most difficult to identify with dynamic techniques such as testing.

—Type qualifiers support efficient inference, which reduces the burden on the programmer by requiring fewer annotations. Efficient inference also allows us to apply type qualifiers to large bodies of legacy code; we can sprinkle in a few type qualifier annotations, and inference determines the remaining qualifiers automatically.

In our framework, type qualifiers are added to every level of the standard types. For example, a source language pointer type $ref(int)$ is extended to $q_1\ ref(q_2\ int)$, where the q_i are qualifiers. The key technical property of type qualifiers is that they do not affect the underlying standard type structure. That is, a program with type qualifier annotations should type check only if the same program with the annotations removed type checks according to the underlying standard type system. Aside from this restriction, type qualifiers could potentially affect the type system arbitrarily. In this paper, however, we focus on a particularly useful subclass of type qualifiers, those that introduce *subtyping*.

In our system, each set of related type qualifiers is in a partial order. For example, consider the qualifiers *tainted* and *untainted*. While it is an error for *tainted* data to be used in *untainted* positions, the reverse is perfectly fine—presumably positions that accept *tainted* data can accept any kind of data. Thus we choose $untainted < tainted$ as the partial order. The partial order among qualifiers is extended in the natural way to a subtype relation among *qualified types*, which are simply types with qualifiers.

We also provide a generic mechanism for programmers to specify and check type qualifiers in their source code. *Type qualifier annotations* specify the initial type qualifiers when program values are created. *Type qualifier checks* test the qualifiers on a value, succeeding only if the actual qualifiers on the value are compatible (in the partial order) with the specified qualifier.

Given a partial order among the type qualifiers and a program with some type qualifier annotations and checks, we can efficiently perform *flow-insensitive type qualifier inference* to infer the remaining qualifiers and check consistency. Our inference algorithm is designed using *constraint-based analysis*. To infer qualifiers in a source program, we scan the program text and generate a series of constraints $q_1 \leq q_2$ among qualifiers and qualifier variables, which stand for as-yet-unknown qualifiers. We solve the constraints for the qualifier variables and warn the programmer if the constraints have no solution, which indicates a type qualifier error. Note that although we have also studied flow-sensitive type qualifiers [Foster et al. 2002; Aiken et al. 2003] (see Section 6), this paper discusses exclusively flow-insensitive type qualifiers.

Type qualifiers can further come with well-formedness conditions that specify how qualifiers on different positions of the same type are related. For example, in the Linux kernel, functions and data marked with `__init` are garbage collected after kernel initialization time (Section 5.3). This optimization saves space, but the programmer must be careful never to use an `__init` function or pointer after the initialization phase is complete. We can interpret this annotation as a qualifier *init* and add an inverse qualifier *noninit* for data and functions that must be live after initialization.

Then there are two conditions to ensure that *init* is used safely. First, for data that

is marked with `_init`, we require that its type obey a *well-formedness constraint*: in a type of the form $q_1 \text{ ref } (q_2 T)$ (the type of a pointer, with qualifier q_1 , that points to an object of type T with qualifier q_2), if q_1 is a *noninit* pointer, then the data that is pointed to must also be live after initialization, and hence q_2 must also be *noninit*. Second, we also treat *init* and *noninit* as *effect qualifiers* [Gifford et al. 1987; Lucassen and Gifford 1988], and we require that any function that transitively calls a function with effect qualifier *init* also has effect qualifier *init*, i.e., the function can only be called during initialization time. We can enforce these restrictions by adding qualifiers to model effects and by adding additional constraints. For example, since *noninit* $<$ *init*, we could generate the constraint $q_2 \leq q_1$ for the type $q_1 \text{ ref } (q_2 T)$ to enforce well-formedness. However, if we added this constraint it would apply to all qualifiers, even though not all qualifiers should propagate in this way. Hence instead we introduce *gated qualifier constraints* $q_2 \leq_S q_1$. Here S is a set of qualifiers (for our example *noninit* and *init*), and such a constraint is satisfiable if $q_1 \in S \wedge q_2 \in S \Rightarrow q_2 \leq q_1$ (Section 3.3). In general these gated constraints allow us to restrict the set of qualifiers that propagate along certain edges, and are also used in handling type casts.

In order to improve the precision of type qualifier inference further, we also support *parametric type qualifier polymorphism*. For example, the C standard library function `strcat`, which destructively appends its second argument to its first argument and returns the result, can be given the type

$$\forall \kappa, \kappa' [\kappa' \leq \kappa]. \text{ref } (\kappa \text{ char}) \times \text{ref } (\kappa' \text{ char}) \longrightarrow \text{ref } (\kappa \text{ char})$$

Here the brackets contain constraints on the polymorphic variables. This is especially important in modeling library functions precisely, so that the many calls to the same library function are not conflated. We perform polymorphic qualifier inference by translating it into a context-free language reachability problem on the constraint graph among the qualifiers (Section 4.1).

To test our ideas in practice, we built a tool called CQUAL² for adding user-defined flow-insensitive type qualifiers to C (Section 4). CQUAL has been used both in our own research and by others [Zhang et al. 2002; Broadwell et al. 2003]. A key feature of CQUAL is that it includes a user interface that shows programmers not only what type qualifiers were inferred but why they were inferred. After inference, the program source code is presented to the user with each identifier colored according to its inferred qualifiers. For each error message, the user can browse qualifier constraints that exhibit the error. For example, if an error occurs because *tainted* data is used in an *untainted* position, the user is shown a set of constraints (roughly corresponding to a program path) that shows step-by-step how *tainted* was propagated to *untainted*. From our own personal experience, such an interface, while often neglected in the research literature, is one of the most important and visible features of any program analysis tool, and we found the interface invaluable in our research. In Section 4 we discuss some of the heuristics we use to make this visualization tool more useful.

Type qualifier systems applied to type-safe languages can be made *sound*, meaning that programs with valid qualifier annotations do not violate an operational

²Freely available under the GNU General Public License at <http://cqual.sourceforge.net>.

semantics of the qualifiers (Section 2.3). For example, the addition of type qualifiers to Java can be made sound. However, since C is not type safe, neither are type qualifiers directly applied to the C type system. Instead, there is a range of design choices that trade off soundness for simplicity and fewer false positive results. Our tool CQUAL allows the user to select from among several soundness options for particular qualifiers. Another alternative would be to combine CQUAL with a system for enforcing memory safety such as CCured [Necula et al. 2002].

We have performed a number of experiments with CQUAL (Section 5). We have used CQUAL to infer *const* qualifiers, and we have found that we were able to infer many additional *consts*, even in programs that already make a significant effort to use *const*. We have used *tainted* and *untainted* qualifiers to check for format-string bugs, a particular kind of security vulnerability, in several popular C programs, and we have found format-string vulnerabilities that were not known to us. We have also used CQUAL to find bugs in the Linux kernel involving invalid uses of *init* data and functions in *noninit* functions.

In summary, the contributions of this paper are as follows:

- We present a framework for adding type qualifiers to almost any language with standard types, and we show that flow-insensitive type qualifier inference can be carried out efficiently.
- We introduce a natural notion of *qualifier polymorphism* that allows types to be polymorphic in their qualifiers. We present examples from existing C programs to show that qualifier polymorphism is useful and in fact necessary in some situations.
- We describe a practical tool CQUAL that adds type qualifiers to the C programming language, and we discuss some key implementation details. We believe many of the lessons learned in developing CQUAL are applicable to other languages and other static analysis tools, as well.
- We describe a novel visualization component that gives users of CQUAL a natural interface for understanding the results of type qualifier inference.
- We present empirical evidence that type qualifiers are useful in practice by describing a number of experiments with type qualifier systems. In the process, we show that our algorithms scale to large programs.

An earlier version of this work was presented in conference publications [Foster et al. 1999; Shankar et al. 2001]. New contributions of this paper include polymorphism via CFL reachability (Section 3.1), gated qualifier constraints for effects and well-formedness constraints (Section 3.3), better handling of type casts (Section 4.2), and additional experiments.

2. QUALIFIERS AND QUALIFIED TYPES

We will present our type qualifier system using lambda calculus extended with updatable references, as shown in Figure 1. In general, type qualifiers can be added to any language with a standard type system, but showing the system on the language in Figure 1 will illustrate most of the important points. In this Section we present a basic type qualifier system that will need some extensions, described in Section 3, to be more useful in practice and to handle certain classes of qualifiers.

$e ::= v$	values
$e_1 e_2$	application
$\text{let } x = e_1 \text{ in } e_2$	name binding
$\text{ref } e$	allocation
$*e$	dereference
$e_1 := e_2$	assignment
$v ::= x$	variable
n	integer
$\lambda x:s.e$	function
$s ::= \text{int}$	integer type
$\text{ref } (s)$	pointer to type s
$s \longrightarrow s'$	function from type s to type s'

Fig. 1. Source Language

$$\begin{array}{c}
\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash_s x : \Gamma(x)} \text{ (Var}_s\text{)} \qquad \frac{}{\Gamma \vdash_s n : \text{int}} \text{ (Int}_s\text{)} \\
\\
\frac{\Gamma[x \mapsto s] \vdash_s e : s'}{\Gamma \vdash_s \lambda x:s.e : s \longrightarrow s'} \text{ (Lam}_s\text{)} \qquad \frac{\Gamma \vdash_s e : s}{\Gamma \vdash_s \text{ref } e : \text{ref } (s)} \text{ (Ref}_s\text{)} \\
\\
\frac{\Gamma \vdash_s e_1 : s \longrightarrow s' \quad \Gamma \vdash_s e_2 : s}{\Gamma \vdash_s e_1 e_2 : s'} \text{ (App}_s\text{)} \qquad \frac{\Gamma \vdash_s e_1 : s_1 \quad \Gamma[x \mapsto s_1] \vdash_s e_2 : s_2}{\Gamma \vdash_s \text{let } x = e_1 \text{ in } e_2 : s_2} \text{ (Let}_s\text{)} \\
\\
\frac{\Gamma \vdash_s e : \text{ref } (s)}{\Gamma \vdash_s *e : s} \text{ (Deref}_s\text{)} \qquad \frac{\Gamma \vdash_s e_1 : \text{ref } (s) \quad \Gamma \vdash_s e_2 : s}{\Gamma \vdash_s e_1 := e_2 : s} \text{ (Assign}_s\text{)}
\end{array}$$

Fig. 2. Standard Type Checking System

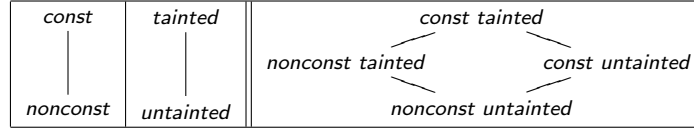


Fig. 3. Example Qualifier Partial Order

Section 4 contains a discussion of how to address some of the issues that come up in applying type qualifiers to the C programming language.

We will assume for the remainder of this section that our input programs are type correct with respect to the standard type system, shown in Figure 2 for completeness, and that function definitions have been annotated with standard types s . If that is not the case, we can always perform a preliminary standard type inference pass.

In our framework, the user specifies a set of qualifiers Q and a partial order \leq among the qualifiers. In practice, the user may wish to specify several sets (Q_i, \leq_i) of qualifiers that do not interact, each with their own partial order. But then we can form $(Q, \leq) = (Q_1, \leq_1) \times \cdots \times (Q_n, \leq_n)$, so without loss of generality we can assume a single partial order of qualifiers. For example, Figure 3 gives two independent partial orders and their equivalent combined, single partial order (in this case the partial orders are lattices). In Figure 3, as in the rest of this paper,

we write elements of Q using *slanted text*. We sometimes refer to elements of Q as *type qualifier constants* to distinguish them from type qualifier variables introduced in Section 2.2.

For our purposes, *types* Typ are terms over a set Σ of n -ary type constructors. Grammatically, types are defined by the language

$$Typ ::= c(Typ_1, \dots, Typ_{arity(c)}) \quad c \in \Sigma$$

In our source language, the type constructors are $\{int, ref, \longrightarrow\}$ with arities 0, 1, and 2, respectively. We construct the *qualified types* $QTyp$ by pairing each standard type constructor in Σ with a type qualifier (recall that a single type qualifier in our partial order may represent a set of qualifiers in the programmer's view). We allow type qualifiers to appear on every level of a type. Grammatically, our new types are

$$QTyp ::= Q c(QTyp_1, \dots, QTyp_{arity(c)}) \quad c \in \Sigma$$

For our source language, the qualified types are

$$\begin{aligned} \tau &::= Q \nu \\ \nu &::= int \mid ref(\tau) \mid \tau \longrightarrow \tau \end{aligned}$$

To avoid ambiguity, when writing down qualified function types we parenthesize them as $Q(\tau \longrightarrow \tau)$. Some example qualified types in our language are *tainted int* and *const ref(untainted int)*. We define the *top-level qualifier* of type $Q \nu$ as its outermost qualifier Q .

2.1 Assertions, Annotations, and Type Checking

So far we have types with attached qualifiers and a partial order among the qualifiers. A key idea behind our framework is that the partial order on type qualifiers induces a *subtyping* relation among qualified types. In a subtyping system, if type B is a subtype of type A , which we write $B \leq A$ (note the overloading on \leq), then wherever an object of type A is allowed an object of type B may also be used. (Subclassing in object-oriented programming languages such as Java and C++ is closely related to subtyping.)

Figure 4a shows how a given qualifier partial order is extended to a subtyping relation for our source language. These rules are standard, and a discussion of them can be found elsewhere [Mitchell 1991]. In general, for any $c \in \Sigma$ the rule

$$\frac{Q \leq Q' \quad \tau_i \leq \tau'_i \quad \tau'_i \leq \tau_i \quad i \in [1..n]}{Q c(\tau_1, \dots, \tau_n) \leq Q' c(\tau'_1, \dots, \tau'_n)}$$

should be sound. Whether the equality (here expressed by two \leq constraints) can be relaxed for any particular position depends on the meaning of the type constructor c .

Next we wish to extend our standard type system to work with qualified types. Thus far, however, we have supplied no mechanism that allows programmers to talk about the qualifiers used in their programs. One place where this issue comes up is when constructing a qualified type during type checking. For example, if we see an occurrence of the integer 0 in the program, how do we decide which qualifier Q to pick for its type $Q int$? We wish to have a generic solution for this problem,

$$\frac{Q \leq Q'}{Q \text{ int} \leq Q' \text{ int}} \text{ (Int}_{\leq}) \quad \frac{Q \leq Q' \quad \tau \leq \tau' \quad \tau' \leq \tau}{Q \text{ ref}(\tau) \leq Q' \text{ ref}(\tau')} \text{ (Ref}_{\leq})$$

$$\frac{Q \leq Q' \quad \tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{Q(\tau_1 \rightarrow \tau_2) \leq Q'(\tau'_1 \rightarrow \tau'_2)} \text{ (Fun}_{\leq})$$

(a) Subtyping Qualified Types

$$\frac{}{\Gamma \vdash n : \text{int}} \text{ (Int)} \quad \frac{\Gamma[x \mapsto \tau] \vdash e : \tau' \quad \text{strip}(\tau) = s}{\Gamma \vdash \lambda x : s. e : \tau \rightarrow \tau'} \text{ (Lam)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref} e : \text{ref}(\tau)} \text{ (Ref)} \quad \frac{\Gamma \vdash e : \nu}{\Gamma \vdash \mathbf{annot}(e, Q) : Q \nu} \text{ (Annot)}$$

(b) Rules for Unqualified Types ν

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \text{ (Var)} \quad \frac{\Gamma \vdash e_1 : Q(\tau \rightarrow \tau') \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau'} \text{ (App)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} x = e_1 \mathbf{in} e_2 : \tau_2} \text{ (Let)} \quad \frac{\Gamma \vdash e : Q \text{ ref}(\tau)}{\Gamma \vdash *e : \tau} \text{ (Deref)}$$

$$\frac{\Gamma \vdash e_1 : Q \text{ ref}(\tau) \quad \Gamma \vdash e_2 : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e_1 := e_2 : \tau} \text{ (Assign)} \quad \frac{\Gamma \vdash e : Q' \nu \quad Q' \leq Q}{\Gamma \vdash \mathbf{check}(e, Q) : Q' \nu} \text{ (Check)}$$

(c) Rules for Qualified Types τ

Fig. 4. Qualified Type Checking System

so in our system, we extend the syntax with two new forms:

$$e ::= \dots$$

	$\mathbf{annot}(e, Q)$	qualifier annotation
	$\mathbf{check}(e, Q)$	qualifier check

A *qualifier annotation* $\mathbf{annot}(e, Q)$ specifies the outermost qualifier Q to add to e 's type. Annotations may only be added to expressions that construct a term, and whenever the user constructs a term our type system requires that they add an annotation. Clearly this last requirement is not always desirable, and in Section 2.2 we describe an inference algorithm that allows programmers to omit these annotations if they like. Dually, a *qualifier check* $\mathbf{check}(e, Q)$ tests whether the outermost qualifier of e 's type is compatible with Q . Notice that if we want to check a qualifier deeper in a non-function type, we can do so by first applying the language's deconstructors. (For example, we can check the qualifier on the contents of a reference x using $\mathbf{check}(*x, Q)$).

Finally, we wish to extend the original type checking system to a *qualified type system* that checks programs with qualified types, including our new syntactic forms $\mathbf{annot}(\cdot, \cdot)$ and $\mathbf{check}(\cdot, \cdot)$. Intuitively this extension should be natural, in the sense that adding type qualifiers should not modify the underlying type structure (we make this precise below). We also need to incorporate subsumption [Mitchell 1991] into our qualified type system to allow subtyping.

We define a pair of translation functions between standard and qualified types and expressions. For a qualified type $\tau \in QTyp$, we define $strip(\tau) \in Typ$ to be τ with all qualifiers removed. Analogously, $strip(e)$ is e with any qualifier annotations or checks removed. In the other direction, for a standard type $s \in Typ$ we define $embed(s, q)$ to be the qualified type with the same shape as s and all qualifiers set to q . Analogously, $embed(e, q)$ is e with $\mathbf{annot}(e', q)$ wrapped around every subexpression e' of e that constructs a term.

Figures 4b and c show the qualified type system for our source language. Judgments are either of form $\Gamma \vdash e : \nu$ (three rules in Figure 4b) or $\Gamma \vdash e : \tau$ (the remaining rules in Figure 4c), meaning that in type environment Γ , expression e has unqualified type ν or qualified type τ . Here Γ is a mapping from variables to qualified types.

The rules (Int) and (Ref) are identical to standard type checking rules, and (Lam) simply adds a check that the parameter's qualified type τ has the same shape as the specified standard type. (This check is not strictly necessary—see Lemma 1 below.) Notice that these three rules produce types that are missing a top-level qualifier. The rule (Annot) adds a top-level qualifier to such a type, which is produced in our qualified type grammar by non-terminal ν . Inspection of the type rules shows that judgments of the form $\Gamma \vdash e : \nu$ can only be used in the hypothesis of (Annot). Thus the net effect of the four rules in Figure 4b is that all constructed terms must be assigned a top-level qualifier with an explicit annotation.

The rules (Var) and (Let) are identical to the standard type checking rules. The rules (App), (Deref), and (Assign) are similar to the standard type checking rules, except that they match the types of their subexpressions against qualified types, and (App) and (Assign) allow subsumption. Notice that these three rules allow arbitrary qualifiers (denoted by Q) when matching a type. Only the rule (Check) actually tests a qualifier on a type.

LEMMA 1. *Let e be a closed term, and let \vdash_s be the standard type checking judgment.*

—If $\emptyset \vdash_s e : s$, then for any qualifier q we have $\emptyset \vdash embed(e, q) : embed(s, q)$.

—If $\emptyset \vdash e : \tau$, then $\emptyset \vdash_s strip(e) : strip(\tau)$.

PROOF. (Sketch) Both properties can be proven by structural induction on e . The proofs are straightforward, since each rule in the standard type system corresponds exactly to one rule in Figure 4 with the qualifiers removed, and vice-versa. For the first claim, we also must observe that consistently adding the same qualifier q to all types maintains the same equalities between types in the program, which is sufficient for maintaining typability because $embed(\cdot, \cdot)$ does not add any type qualifier checks. For the second statement, we must also observe that the subsumption rules in Figure 4a require type structures to be equal, and so if $\tau \leq \tau'$, it is always the case that $strip(\tau) = strip(\tau')$. \square

This lemma formalizes our intuitive requirement that type qualifiers do not affect the underlying type structure.

$$\begin{array}{c}
\frac{}{\Gamma \vdash' n : \text{int}} \text{ (Int')} \qquad \frac{\Gamma[x \mapsto \tau] \vdash' e : \tau' \quad \tau = \text{embed}'(s)}{\Gamma \vdash' \lambda x : s. e : \tau \longrightarrow \tau'} \text{ (Lam')} \\
\frac{\Gamma \vdash' e : \tau}{\Gamma \vdash' \text{ref } e : \text{ref } (\tau)} \text{ (Ref')} \qquad \frac{\Gamma \vdash' e : \nu}{\Gamma \vdash' \text{annot}(e, Q) : Q \nu} \text{ (Annot')} \\
\frac{\Gamma \vdash' e : \nu \quad \kappa \text{ fresh}}{\Gamma \vdash' e : \kappa \nu} \text{ (Fresh')} \\
\text{(a) Rules for Unqualified Types } \nu \\
\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash' x : \Gamma(x)} \text{ (Var')} \qquad \frac{\Gamma \vdash' e_1 : Q (\tau \longrightarrow \tau') \quad \Gamma \vdash' e_2 : \tau_2 \quad \tau_2 \leq \tau}{\Gamma \vdash' e_1 e_2 : \tau'} \text{ (App')} \\
\frac{\Gamma \vdash' e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash' e_2 : \tau_2}{\Gamma \vdash' \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (Let')} \qquad \frac{\Gamma \vdash' e : Q \text{ref } (\tau)}{\Gamma \vdash' *e : \tau} \text{ (Deref')} \\
\frac{\Gamma \vdash' e_1 : Q \text{ref } (\tau) \quad \Gamma \vdash' e_2 : \tau' \quad \tau' \leq \tau}{\Gamma \vdash' e_1 := e_2 : \tau'} \text{ (Assign')} \qquad \frac{\Gamma \vdash' e : Q' \nu \quad Q' \leq Q}{\Gamma \vdash' \text{check}(e, Q) : Q' \nu} \text{ (Check')} \\
\text{(b) Rules for Qualified Types } \tau
\end{array}$$

Fig. 5. Qualified Type Inference System

2.2 Inference

As described so far, type qualifiers place a rather large burden on programmers wishing to use them: programmers must add explicit qualifier annotations to all constructed terms in their programs. We would like to reduce this burden by performing *type qualifier inference*, which is analogous to standard type inference. As with standard type inference, we introduce *type qualifier variables* $QVar$ to stand for unknown qualifiers that we need to solve for. We write qualifier variables with the Greek letter κ . In the remainder of this paper we use *type qualifier constants* to refer to elements of the given qualifier partial order, and we use *type qualifiers* to refer to either a qualifier constant or a qualifier variable. We define a function $\text{embed}'(s)$ that maps standard types to qualified types by inserting fresh type qualifier variables at every level:

$$\begin{array}{ll}
\text{embed}'(\text{int}) = \kappa \text{int} & \kappa \text{ fresh} \\
\text{embed}'(\text{ref } (s)) = \kappa \text{ref } (\text{embed}'(s)) & \kappa \text{ fresh} \\
\text{embed}'(s \longrightarrow s') = \kappa (\text{embed}'(s) \longrightarrow \text{embed}'(s')) & \kappa \text{ fresh}
\end{array}$$

The type qualifier inference rules for our source language are shown in Figure 5. In this system, Q stands for either a qualifier constant or a qualifier variable κ . These rules are essentially the same as the rules in Figure 4, with two differences. First, we allow qualifier annotations to be omitted in the source program. If they are, rule (Fresh') is used to introduce a fresh type qualifier variable to stand for the unknown qualifier on the term. Although this rule is not syntax-driven, it could be made so, since either rule (Annot') or rule (Fresh') must be used before applying any of the rules in part (b). Second, we use embed' in (Lam') to map the given standard type to a type with fresh qualifier variables. To simplify the rules slightly

$$\begin{aligned}
C \cup \{Q \text{ int} \leq Q' \text{ int}\} &\Rightarrow C \cup \{Q \leq Q'\} \\
C \cup \{Q \text{ ref } (\tau) \leq Q' \text{ ref } (\tau')\} &\Rightarrow C \cup \{Q \leq Q'\} \cup \{\tau \leq \tau'\} \cup \{\tau' \leq \tau\} \\
C \cup \{Q (\tau_1 \longrightarrow \tau_2) \leq Q' (\tau'_1 \longrightarrow \tau'_2)\} &\Rightarrow C \cup \{Q \leq Q'\} \cup \{\tau'_1 \leq \tau_1\} \cup \{\tau_2 \leq \tau'_2\}
\end{aligned}$$

(a) Resolution Rules for Subtyping Constraints

$$\begin{aligned}
C \cup \{q \leq Q\} \cup \{Q \leq Q'\} &\cup\Rightarrow \{q \leq Q'\} \\
C \cup \{Q \leq Q'\} \cup \{Q' \leq q\} &\cup\Rightarrow \{Q \leq q\}
\end{aligned}$$

(b) Resolution Rules for Qualifier Constraints

Fig. 6. Constraint Resolution

we use our assumption that the program is correct with respect to the standard types to avoid some shape matching constraints. For example, in (App') we know that e_1 has a function type, but we do not know its qualifier.

Given a solution to the constraints generated by inference (see below), we believe it is straightforward to show that the inference system in Figure 5 is sound and complete with respect to the checking system in Figure 4, although we have not proven it formally.

After we have applied the type rules in Figure 5, we have a typing derivation that contains qualifier variables to be solved for. The qualifier variables must satisfy two kinds of constraints that form the side conditions of the inference rules: *subtyping constraints* of the form $\tau_1 \leq \tau_2$ and *qualifier constraints* (from the rule (Check')) of the form $Q_1 \leq Q_2$. We say that these constraints are *generated* by the typing rules. In order to find a valid typing (if one exists), we must solve the generated constraints for the qualifier variables.

The first step is to apply the rules of Figure 6a to reduce the subtyping constraints to qualifier constraints. Notice that because we assume that the program we are analyzing type checks with respect to the standard types, we know that none of the structural matching cases in Figure 4a can fail.

After exhaustively applying the rules in Figures 5 and 6a, we are left with qualifier constraints of the form $Q_1 \leq Q_2$, where the Q_i are type qualifier constants from Q or type qualifier variables κ . We need to solve these qualifier constraints to complete type qualifier inference.

DEFINITION 2. *A solution S to a system of qualifier constraints C is a mapping from type qualifier variables to type qualifier constants such that for each constraint $Q_1 \leq Q_2$, we have $S(Q_1) \leq S(Q_2)$.*

We write $S \models C$ if S is a solution to C . Note that there may be many possible solutions to C . We say that C is *satisfiable* if there exists an S such that $S \models C$. For many uses of type qualifiers, we are interested in satisfiability rather than the actual solution. If we are looking for a solution there are two in particular that we may be interested in.

DEFINITION 3. *If $S \models C$, then S is a least (respectively greatest) solution if, for any other S' such that $S' \models C$, and for all $\kappa \in \text{dom}(S)$, we have $S(\kappa) \leq S'(\kappa)$ (respectively $S(\kappa) \geq S'(\kappa)$).*

For example, the *const* inference of Section 5.1 finds a greatest solution, to add as many *consts* as possible. For the format-string vulnerability experiment in Section 5.2, we are mostly interested in satisfiability. However, if we wanted to compute a full solution, we would find a least solution to require as little data to be tainted as possible.

A system of qualifier constraints is also known as an *atomic subtyping constraint system*. In general, checking satisfiability and/or solving atomic subtyping constraints over an arbitrary partial order is NP-hard, even with fixed Q [Pratt and Tiuryn 1996]. However, there are well-known linear-time algorithms for solving such constraints efficiently if Q is a semilattice [Rehof and Mogensen 1996]. In this case, given a system of constraints C of size n and a fixed set of qualifiers, we can check satisfiability of the constraints and find a solution in $O(n)$ time [Rehof and Mogensen 1996; Foster 2002].

For example, if Q is a lattice, we can repeatedly apply the transitive closure rules in Figure 6b to the qualifier constraints until we reach a fixpoint. These rules effectively propagate qualifier constants q through the constraints. Here $\cup \Rightarrow$ means the constraint on the right-hand side is added given the constraints on the left-hand side. After reaching a fixpoint, the constraint system C is satisfiable if and only if there is no constraint $q \leq q' \in C$ where $q \not\leq q'$ in the lattice. For a qualifier variable κ , its least solution is $\bigsqcup_{\{q \leq \kappa\} \in C} q$ and its greatest solution is $\bigsqcap_{\{\kappa \leq q\} \in C} q$.

2.3 Semantics and Soundness

We can prove that our qualified type system (applied to our extended lambda calculus), including parametric polymorphic type qualifiers (Section 3.1), is sound under a natural semantics for type qualifiers. Figure 7 gives our semantic reduction rules. In these semantics a store S is a mapping from locations l to values v . In order to type check programs with locations in them, we treat locations l as free variables and type them using rule (Var). Thus in the proof of soundness, as reduction proceeds and new locations are allocated we keep track of their types in the type environment. We use \emptyset for the empty store. Values are standard values (integers, locations, or functions) paired with uninterpreted qualifiers, written as a superscript. Notice that in these semantics the rules that deconstruct values ([App], [Deref], and [Assign]) throw away the outermost qualifier. Only rule [Check] tests the top-level qualifier of a value.

Any program to which none of the rules in Figure 7 apply is *stuck*, which we express by reducing the program to a special symbol **err**. The symbol **err** is not a value and has no valid type. We can prove that our system is sound by showing that no well-typed program reduces to **err**. The proof is omitted, since it uses standard techniques [Wright and Felleisen 1994; Eifrig et al. 1995; Odersky et al. 1997]; a proof excepting qualifier polymorphism is available in the first author’s thesis [Foster 2002]. In the theorem below, we use r to stand for a reduction result, either a value v^Q or **err**.

THEOREM 4. *If $\emptyset \vdash e : \tau$ and $\langle \emptyset, e \rangle \rightarrow \langle S', r \rangle$, then r is not **err**.*

$$\begin{array}{c}
\frac{l \in \text{dom}(S)}{\langle S, l^Q \rangle \rightarrow \langle S, l^Q \rangle} \text{ [Var]} \\
\\
\frac{}{\langle S, \text{annot}(n, Q) \rangle \rightarrow \langle S, n^Q \rangle} \text{ [Int]} \\
\\
\frac{}{\langle S, \text{annot}(\lambda x:s.e, Q) \rangle \rightarrow \langle S, (\lambda x:s.e)^Q \rangle} \text{ [Lam]} \\
\\
\frac{\langle S, e_1 \rangle \rightarrow \langle S_1, (\lambda x.e)^{Q_1} \rangle \quad \langle S_1, e_2 \rangle \rightarrow \langle S_2, v_2^{Q_2} \rangle \quad \langle S_2, e[x \mapsto v_2^{Q_2}] \rangle \rightarrow \langle S_3, v_3^{Q_3} \rangle}{\langle S, e_1 e_2 \rangle \rightarrow \langle S_3, v_3^{Q_3} \rangle} \text{ [App]} \\
\\
\frac{\langle S, e_1 \rangle \rightarrow \langle S_1, v_1^{Q_1} \rangle \quad \langle S_1, e_2[x \mapsto v_1^{Q_1}] \rangle \rightarrow \langle S_2, v_2^{Q_2} \rangle}{\langle S, \text{let } x = e_1 \text{ in } e_2 \rangle \rightarrow \langle S_2, v_2^{Q_2} \rangle} \text{ [Let]} \\
\\
\frac{\langle S, e \rangle \rightarrow \langle S_1, v^Q \rangle \quad l \notin \text{dom}(S_1)}{\langle S, \text{annot}(\text{ref } e, Q') \rangle \rightarrow \langle S_1[l \mapsto v^Q], l^{Q'} \rangle} \text{ [Ref]} \\
\\
\frac{\langle S, e \rangle \rightarrow \langle S_1, l^Q \rangle \quad l \in \text{dom}(S_1)}{\langle S, *e \rangle \rightarrow \langle S_1, S_1(l) \rangle} \text{ [Deref]} \\
\\
\frac{\langle S, e_1 \rangle \rightarrow \langle S_1, l^Q \rangle \quad \langle S_1, e_2 \rangle \rightarrow \langle S_2, v^{Q'} \rangle \quad l \in \text{dom}(S_2)}{\langle S, e_1 := e_2 \rangle \rightarrow \langle S_2[l \mapsto v^{Q'}], v^{Q'} \rangle} \text{ [Assign]} \\
\\
\frac{\langle S, e \rangle \rightarrow \langle S_1, v^{Q'} \rangle \quad Q' \leq Q}{\langle S, \text{check}(e, Q) \rangle \rightarrow \langle S_1, v^{Q'} \rangle} \text{ [Check]}
\end{array}$$

Fig. 7. Big-Step Operational Semantics with Qualifiers

3. REFINEMENTS

3.1 Parametric Polymorphism

There is a well-known problem with standard monomorphic type systems, like the one we have presented so far: multiple calls to the same function are conflated, leading to a loss of precision. For example, suppose we have qualifier constants \mathbf{a} and \mathbf{b} with partial order $\mathbf{b} < \mathbf{a}$, and consider the following two function definitions of the identity function, where we have annotated the argument with a qualified type:

```

let  $id_1 = \lambda x:(\mathbf{a} \text{ int}).x$ 
let  $id_2 = \lambda x:(\mathbf{b} \text{ int}).x$ 

```

We would like to have only a single copy of this function, since both versions behave the same and in fact compile to the same code. Unfortunately, without polymorphism we need both. The return type of id_1 must be $\mathbf{a} \text{ int}$, and thus an object of type $\mathbf{b} \text{ int}$ can be passed to id_1 , but the return value has qualifier \mathbf{a} . The argument type of id_2 must be $\mathbf{b} \text{ int}$, and thus an object of type $\mathbf{a} \text{ int}$ cannot be passed to id_2 . The problem here is that the type of the identity function on integers is $Q \text{ int} \longrightarrow Q \text{ int}$ with Q appearing both covariantly (to the right of the arrow) and contravariantly (to the left of the arrow).

Notice, however, that the identity function behaves the same for any qualifier Q . We specify this in type notation with the *parametric polymorphic* type signature [Milner 1978] $\forall \kappa. \kappa \text{ int} \longrightarrow \kappa \text{ int}$. When we apply a function of this type to an argument, we first *instantiate* its type at a particular qualifier, in our case either as $a \text{ int} \longrightarrow a \text{ int}$ or $b \text{ int} \longrightarrow b \text{ int}$.

The traditional way to add polymorphism to a constraint-based type inference system is to use *polymorphically constrained types* [Eifrig et al. 1995; Odersky et al. 1997]. In this approach, we modify our type grammar as follows:

$$\begin{aligned} \sigma &::= \forall \vec{\kappa}[C].\tau \\ \tau &::= Q \nu \\ \nu &::= \text{int} \mid \text{ref } (\tau) \mid \tau \longrightarrow \tau \\ C &::= \emptyset \mid \{Q \leq Q'\} \mid C \cup C \end{aligned}$$

The type $\forall \vec{\kappa}[C].\tau$ represents all types of the form $\tau[\vec{\kappa} \mapsto \vec{Q}]$ for any \vec{Q} that satisfies the constraints $C[\vec{\kappa} \mapsto \vec{Q}]$. Note that polymorphism only applies to the qualifiers and not to the underlying types. Adding polymorphism only serves to make type qualifier inference more precise, and it does not affect the operational semantics. We can prove soundness for such a system using standard techniques [Wright 1995; Eifrig et al. 1995; Odersky et al. 1997; Mossin 1996].

In practice, polymorphically constrained type inference systems are tricky to implement directly. Instead, we use an equivalent formulation based on instantiation constraints, due to Rehof et al. [Rehof and Fähndrich 2001]. In this approach, inferring polymorphic qualifiers is reduced to a context-free language (CFL) reachability problem [Reps et al. 1995] on the qualifier constraints viewed as a graph. The CFL reachability problem can be solved in cubic time. This formulation has the added advantage of naturally supporting polymorphic recursion, which our implementation does as well.

In this formulation, the nodes in the qualifier constraint graph are qualifier constants and variables. A qualifier constraint $Q \leq Q'$ generated by the rules in Figures 5 and 6a is represented by an unlabeled directed edge $Q \longrightarrow Q'$ from the node for Q to the node for Q' . Labeled edges will be used to represent qualifier instantiation, as discussed next.

Figure 8a extends the rules in Figure 5 to add polymorphism. As is standard, the rule (Let_{CFL}) introduces polymorphism, which is restricted to syntactic values (variables, integers, or functions in our language) [Wright 1995] in order to be sound in the presence of updatable references. In rule (Let_{CFL}), we bind variable x to a pair containing its type and $fv(\Gamma)$, the set of qualifier variables that *cannot* be quantified [Henglein 1993]. Note this is the inverse of standard polymorphic type inference. In particular, the type scheme $(\tau_1, fv(\Gamma))$ corresponds to the polymorphically constrained type $\forall \vec{\kappa}[C].\tau_1$ where $\vec{\kappa} = (fv(\tau_1) \cup fv(C)) - fv(\Gamma)$.

Each instantiation occurs at a particular syntactic location in the program, which we associate with an index i . Rule (Var_{CFL}) instantiates the type of variable x , which is labeled with index i . We create a type $\tau' = \text{fresh}(\tau)$, where $\text{fresh}(\tau)$ is a type with the same shape as τ but fresh qualifier variables. Then we make an instantiation constraint $\tau \xrightarrow{i} \tau'$, represented as a labeled directed edge (we write the edge as a hypothesis to the rule). Intuitively this corresponds to a substitution

$$\frac{\Gamma \vdash' v : \tau_1 \quad \Gamma[x \mapsto (\tau_1, fv(\Gamma))] \vdash' e : \tau_2}{\Gamma \vdash' \mathbf{let} \ x = v \ \mathbf{in} \ e : \tau_2} \text{ (Let}_{\text{CFL}}\text{)}$$

$$\frac{\Gamma(x) = (\tau, \vec{\kappa}) \quad \tau' = \mathit{fresh}(\tau) \quad \tau \xrightarrow{)}_i \tau' \quad \kappa \xrightarrow{(,)}_i \kappa \quad (\forall \kappa \in \vec{\kappa})}{\Gamma \vdash' x^i : \tau'} \text{ (Var}_{\text{CFL}}\text{)}$$

(a) Type Rules

$$\begin{aligned} C \cup \{Q \ \mathit{int} \xrightarrow{(,)}_i Q' \ \mathit{int}\} &\Rightarrow C \cup \{Q \xrightarrow{(,)}_i Q'\} \\ C \cup \{Q \ \mathit{int} \xrightarrow{)}_i Q' \ \mathit{int}\} &\Rightarrow C \cup \{Q' \xrightarrow{)}_i Q\} \\ C \cup \{Q \ \mathit{ref}(\tau) \xrightarrow{(,)}_i Q' \ \mathit{ref}(\tau')\} &\Rightarrow C \cup \{Q \xrightarrow{(,)}_i Q'\} \cup \{\tau \xrightarrow{(,)}_i \tau'\} \cup \{\tau' \xrightarrow{)}_i \tau\} \\ C \cup \{Q \ \mathit{ref}(\tau) \xrightarrow{)}_i Q' \ \mathit{ref}(\tau')\} &\Rightarrow C \cup \{Q \xrightarrow{)}_i Q'\} \cup \{\tau \xrightarrow{)}_i \tau'\} \cup \{\tau' \xrightarrow{(,)}_i \tau\} \\ C \cup \{Q(\tau_1 \longrightarrow \tau_2) \xrightarrow{(,)}_i Q'(\tau'_1 \longrightarrow \tau'_2)\} &\Rightarrow C \cup \{Q \xrightarrow{(,)}_i Q'\} \cup \{\tau'_1 \xrightarrow{)}_i \tau_1\} \cup \{\tau_2 \xrightarrow{(,)}_i \tau'_2\} \\ C \cup \{Q(\tau_1 \longrightarrow \tau_2) \xrightarrow{)}_i Q'(\tau'_1 \longrightarrow \tau'_2)\} &\Rightarrow C \cup \{Q \xrightarrow{)}_i Q'\} \cup \{\tau'_1 \xrightarrow{(,)}_i \tau_1\} \cup \{\tau_2 \xrightarrow{)}_i \tau'_2\} \end{aligned}$$

(b) Structural Edge Generation Rules

Fig. 8. Polymorphic Constraint Graph Construction

S_i , where $S_i\tau = \tau'$. (See [Rehof and Fähndrich 2001; Henglein 1993] for details.) We also add self loops labeled with both $(,)_i$ and $)_i$ for each $\kappa \in \vec{\kappa}$, i.e., for each qualifier variable that $(\text{Let}_{\text{CFL}})$ determined was not generalizable. Intuitively this corresponds to requiring $S_i\kappa = \kappa$, meaning κ was instantiated to itself, i.e., κ was not actually instantiated.

After we have generated qualifier constraints and instantiation constraints, we apply the rules in Figure 8b exhaustively to propagate instantiation constraints from types to qualifiers, taking contravariance into account by flipping the kind of parenthesis and the direction of the edge. Given this graph, the CFL reachability problem is to find all paths in the graph made up of edges whose labels do not contain any mismatched parentheses (ignoring the unlabeled edges), where only open and closed parentheses with the same index match [Rehof and Fähndrich 2001].

As an example, consider again the identity function, with two uses:

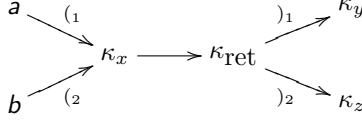
```

let id = λx: int .x in
let a = annot(0, a) in
let b = annot(1, b) in
  let y = id1 a in
  let z = id2 b in 42

```

In this program, a has qualifier a and b has qualifier b , and we have labeled the two uses of id with location 1 and location 2. Then the CFL reachability graph

generated by this example looks like the following:



Here κ_{ret} is the qualifier variable on the return type of id , and κ_x , κ_y , and κ_z are the qualifier variables for x , y , and z , respectively. Since we require paths with balanced parentheses, there are only two paths through this graph: from a to κ_y , indicating that y should be qualified with a , and from b to κ_z . The other paths, from a to κ_z and from b to κ_y , are *unrealizable* [Reps et al. 1995], since they correspond to a call at position 1 followed by a return at position 2 and vice-versa. The CFL reachability technique eliminates these unrealizable paths from inference, giving us polymorphism.

This system is equivalent to one that uses polymorphically constrained types. This is important, because CQUAL allows the user to specify polymorphically constrained types by hand (Section 4.1). In particular, for a polymorphically constrained type $\forall \vec{\kappa}[C].\tau$, the constraints C are added directly to the constraint graph, and the type is instantiated as usual, with self edges for non-quantified variables.

Optimizations. Our CFL reachability algorithm has several important optimizations that improve on the cubic time bound in practice for C programs. When the algorithm discovers a non-trivial matched parenthesis path through the graph, it creates a transitive summary edge from the start to the end of the path and flags the path so that it will not be explored again. This optimization was originally described by Horwitz, Reps and Sagiv, and improves the asymptotic running time of matched parenthesis CFL reachability to $O(n)$ [Horwitz et al. 1995].

Qualifier variables associated with global program variables are made monomorphic by treating them as if they have self-edges with every parenthesis in the language. As a consequence, the CFL reachability relation is transitive at globals. To see why, consider CFL paths $p_1 : x \rightarrow \dots \rightarrow y$ and $p_2 : y \rightarrow \dots \rightarrow z$. There must not be any mismatched parentheses along these paths, but there may be unmatched parentheses [Rehof and Fähndrich 2001]. For example, p_1 may end with an unmatched $(_1$ and p_2 may begin with an unmatched $)_2$. For this reason, it is not always possible to concatenate two valid CFL paths to obtain a new valid CFL path. If y is global, though, then we can use the self-edges on y to match all the unmatched opening parentheses in p_1 and all the unmatched closing parentheses in p_2 . We can then concatenate p_1 and p_2 to obtain a valid CFL path.

We exploit this fact to speed up the CFL reachability computation. Since CFL paths are not normally concatenable, the CFL reachability algorithm often must explore the same region of the graph several times—once for each entry point to that region. CFL paths are concatenable at global nodes, though, so the CFL reachability algorithm can compute the reachability set of a global node once and reuse that information whenever a reachability query encounters that global node. This optimization provides several orders of magnitude speed improvement and was originally described by Das et al. [Das et al. 2001].

$$\frac{Q \leq Q' \quad \tau \leq \tau' \quad \mathit{const} \leq Q'}{Q \mathit{ref}(\tau) \leq Q' \mathit{ref}(\tau')} \text{ (Ref}'_{\leq}\text{)}$$

Fig. 9. Subtyping Non-Writable References

We also prune regions of the graph that have no outgoing edges with closing parenthesis labels. Searching for matched parenthesis paths through these regions is futile. Since we perform many CFL reachability queries, stopping as soon as we reach one of these regions provides another order of magnitude speedup.

3.2 Subtyping under Non-Writable Pointer Types

As is standard, in Figure 4a we use a conservative rule (Ref_≤) for pointer subtyping: the constraint $\mathit{ref}(\tau) \leq \mathit{ref}(\tau')$ is satisfiable only if $\tau = \tau'$. This rule can often lead to non-intuitive “backward” qualifier propagation. For example, consider the following code:

```
let f = λx: ref(int). *x in
  f y;
  f z
```

Ignoring the outermost qualifier, inference assigns the domain of f type $\mathit{ref}(\kappa \mathit{int})$. Assume that the types of y and z are $\mathit{ref}(\kappa' \mathit{int})$ and $\mathit{ref}(\kappa'' \mathit{int})$, respectively. Then by (Ref_≤), the first application requires $\kappa' = \kappa$, and the second application requires $\kappa'' = \kappa$. Putting the two together yields the rather counter-intuitive $\kappa' = \kappa''$. In other words, y 's qualifier κ' is propagated from x into f and then backward to κ'' and z .

While we could solve this problem with polymorphism, in many cases there is a far simpler solution. Since f does not write through its parameter x , we know that y and z cannot be modified by f , and thus we can soundly weaken our constraints to $\kappa' \leq \kappa$ and $\kappa'' \leq \kappa$ [Pierce 2002] (Section 15.5). Think of an updatable reference x containing data of type τ_x as an object with two methods $\mathit{get}_x : \mathit{void} \rightarrow \tau_x$ and $\mathit{set}_x : \tau_x \rightarrow \mathit{void}$ to read and write the reference, respectively [Fähndrich et al. 1998]. Here void is a placeholder meaning “no parameter” or “no result.” Notice that τ_x appears both co- and contravariantly (on the left and right sides of the function arrow). When we apply f to y in the above code, we generate two constraints:

$$\begin{aligned} \mathit{void} \rightarrow \tau_y &\leq \mathit{void} \rightarrow \tau_x && (1) \text{ get compatibility} \\ \tau_y \rightarrow \mathit{void} &\leq \tau_x \rightarrow \mathit{void} && (2) \text{ set compatibility} \end{aligned}$$

These constraints correspond to (Fun_≤) in Figure 4a: the constraint (1) yields $\tau_y \leq \tau_x$ and (2) yields $\tau_x \leq \tau_y$, which put together produce $\tau_y = \tau_x$. But if f does not write through x , then intuitively x does not have a set method. Thus, analogous to standard width subtyping in object-oriented type systems, we do not generate constraint (2), and the result is that we only require $\tau_y \leq \tau_x$.

Thus we can use a new subtyping rule for references, as shown in Figure 9 for a system where const marks non-updatable reference types. We refer to this as *deep subtyping*. There are a number of techniques for checking whether a particular name is used to write to an updatable reference, and any one could be used to decide

where to use deep subtyping. In Section 4.1 we describe our implementation, which uses explicitly specified *const* annotations.

3.3 Gated Qualifier Constraints, Effect Qualifiers, and Well-formedness Constraints

The qualifier inference rules described in Section 2 generate subtyping constraints following the standard “data flow” of the program, which allows qualifiers to model how values are propagated during evaluation. However, checking some program properties requires modeling other aspects of computation. Two examples that we have encountered in practice are *effect qualifiers* that interact with control flow, rather than data flow, and qualifiers that place structural *well-formedness constraints* on types. Both of these features are used in our experiment checking initialization in the Linux kernel (Section 5.3).

Before discussing effect qualifiers and well-formedness constraints further, we need to introduce some new machinery into our type system. To support these additional qualifiers, our type system will generate extra constraints during the analysis. Depending on whether the qualifier is a standard qualifier, an effect qualifier, or a well-formedness qualifier, it may or may not interact with these new constraints (examples below). To model this behavior, we introduce *gated qualifier constraints* of the form $\tau \leq_S^p \tau'$, where S is a set of qualifier constants (which we call the *gate*), and p is a *polarity*, either \rightarrow or \leftarrow . Following structural decomposition rules analogous to Figure 6a (with the polarity flipping for contravariant positions), this constraint reduces to a set of constraints $Q \leq_S^p Q'$ (not shown). The constraint resolution algorithm mentioned in Section 2.2 includes the transitive closure rules shown in Figure 6b, which propagate bounds from Q to Q' and vice-versa. In contrast, for a constraint $Q \leq_S^p Q'$, only qualifiers $q \in S$ are propagated through this constraint, and then only in the direction specified by p :

$$\begin{aligned} C \cup \{q \leq Q\} \cup \{Q \leq_S^{\rightarrow} Q'\} \cup \{q \leq Q'\} & \quad q \in S \\ C \cup \{Q \leq_S^{\leftarrow} Q'\} \cup \{Q' \leq q\} \cup \{Q \leq q\} & \quad q \in S \end{aligned}$$

We write $Q \leq_S Q'$ as a shorthand for $Q \leq_S^{\rightarrow} Q'$ and $Q \leq_S^{\leftarrow} Q'$.

One use of gated qualifier constraints is to model effects [Gifford et al. 1987; Lucassen and Gifford 1988]. The effect of a function is traditionally the set of locations the invocation of the function may access. In our type qualifier system, the effect of a function is a qualifier q , which must be the least upper bound of the set of qualifiers on types that may be affected when the function is executed. Formally, we extend our typing judgments from Figure 5 to the form $\Gamma \vdash e : \tau; Q$, meaning that in type environment Γ , expression e has type τ , and evaluating e has effect Q . We then modify our type rules to track effects, e.g., the (App') rule becomes

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : Q(\tau \longrightarrow \tau'); Q_1 \quad \Gamma \vdash e_2 : \tau_2; Q_2 \quad \tau_2 \leq \tau \\ Q_1 \leq_{S_e} \kappa \quad Q_2 \leq_{S_e} \kappa \quad Q \leq_{S_e} \kappa \quad \kappa \text{ fresh} \end{array}}{\Gamma \vdash e_1 e_2 : \tau'; \kappa} \text{ (App')}$$

where S_e is the set of effect qualifiers. In words, this rule means that the effect of applying function e_1 to argument e_2 is the least upper bound of the effect of evaluating e_1 , the effect of evaluating e_2 , and the effect of invoking e_1 . Notice that by using constraints with gate S_e , only qualifiers that are specified in the

configuration file to be effects will be propagated over the edges $Q_i \leq_{S_e} \kappa$ and $Q \leq_{S_e} \kappa$.

We also use gated qualifier constraints to enforce well-formedness conditions on types, which require that qualifiers on different positions of the same type are related [Henglein 1991]. For example, a pointer qualifier may propagate (or *flow*) to the pointed-to qualifier or vice-versa, or a qualifier on an aggregate may flow to qualifiers on the fields of the aggregate or vice-versa (Section 5.3). To enforce these conditions, we implicitly require that all types in the program be well-formed, and we place gated qualifier constraints on well-formed types.

For example, let S_{down*} be the set of all qualifiers that flow from pointer qualifiers to pointed-to qualifiers (“down pointers”), and let S_{up*} be the set of all qualifiers that flow in the opposite direction (“up pointers”). Then a type $Q \text{ ref } (Q' \ \tau)$ is well-formed if $Q' \ \tau$ is well-formed and the following constraints are satisfied:

$$Q \leq_{S_{down*}}^{\rightarrow} Q' \quad Q' \leq_{S_{down*}}^{\leftarrow} Q \quad Q \leq_{S_{up*}}^{\leftarrow} Q' \quad Q' \leq_{S_{up*}}^{\rightarrow} Q$$

In other words, these constraints will effectively add the following reductions (in order):

$$\begin{array}{ll} C \cup \{q \leq Q\} \cup \{Q \leq_{S_{down*}}^{\rightarrow} Q'\} \cup \Rightarrow \{q \leq Q'\} & q \in S_{down*} \\ C \cup \{Q \leq q\} \cup \{Q' \leq_{S_{down*}}^{\leftarrow} Q\} \cup \Rightarrow \{Q' \leq q\} & q \in S_{down*} \\ C \cup \{Q' \leq q\} \cup \{Q \leq_{S_{up*}}^{\leftarrow} Q'\} \cup \Rightarrow \{Q \leq q\} & q \in S_{up*} \\ C \cup \{q \leq Q'\} \cup \{Q' \leq_{S_{up*}}^{\rightarrow} Q\} \cup \Rightarrow \{q \leq Q\} & q \in S_{up*} \end{array}$$

Thus using this machinery, qualifiers in S_{down*} propagate from pointers to pointed-to types, and qualifiers in S_{up*} propagate in the opposite direction. Notice that ordinary qualifiers not in S_{down*} or S_{up*} are unaffected by this constraint. For example, the tainting analysis used to find format-string bugs in Section 5.2 has no well-formedness conditions. Our implementation also provides support for two other well-formedness conditions: in a similar manner as above, the user may specify that qualifiers on pointers-to-aggregates (structures and unions) constrain the corresponding pointers-to-fields, and separately, the user may specify that qualifiers on aggregates (structures and unions) constrain their fields. For symmetry, our implementation supports well-formedness constraints that flow up and down, though for most applications one direction is sufficient. For example, in an analysis to track untrusted pointers in operating system kernels [Johnson and Wagner 2004], we observed that any pointer loaded through an untrusted pointer must also be untrusted, which we modeled using the downward pointer well-formedness condition. This could also be implemented via a dual constraint that trusted pointers can only be loaded through other trusted pointers, but this would be redundant.

Gated qualifier constraints are a particular example of *conditional constraints*, which are of the form $C_1 \Rightarrow C_2$, where the C_i are qualifier constraints. In particular, $Q \leq_{\vec{S}} Q'$ can be reduced to the conditional constraints

$$\{q \leq Q\} \Rightarrow \{q \leq Q'\} \quad \forall q \in S$$

and $Q \leq_{\overleftarrow{S}} Q'$ can be reduced to the conditional constraints

$$\{Q' \leq q\} \Rightarrow \{Q \leq q\} \quad \forall q \in S$$

CQUAL includes support for general conditional constraints, although none is included in our examples. A solution S solves a conditional constraint $C_1 \Rightarrow C_2$ if either $S \not\models C_1$ or $S \models C_2$.

We believe that the system with effect qualifiers and well-formedness conditions added is still sound, using a small variation on the subject-reduction style proof mentioned in Section 2.3.

4. CQUAL

To test our ideas in practice, we have built a tool called CQUAL that adds flow-insensitive type qualifiers to the C programming language.³ To use CQUAL, programmers annotate their C programs with a few type qualifiers, and then CQUAL performs type qualifier inference, as discussed in Sections 2 and 3.

Rather than computing a full solution to the qualifier constraints, CQUAL computes the transitive closure of the constraints and checks satisfiability. CQUAL includes some extra closure rules to handle the discrete partial order and some semilattices. The results, which include the computed closure and warnings for unsatisfiable constraints, are presented to the user with an Emacs-based user interface [Harrelson 2001]. An Eclipse plug-in interface is also available [Greenfieldboyce and Foster 2004].

The correspondence between the core part of C and the lambda calculus-based formalism of the previous sections is fairly straightforward: primitive values such as integers, characters, and so on are treated like *int* values, pointers are modeled with *ref* types, C functions correspond to lambda expressions, and local variable bindings correspond to **let**- or lambda-bound variables. In the rest of this section, we discuss extending C types to include qualifiers, to take the place of annotations and checks, as well as how we analyze some of the interesting features of C and some of the choices we made in designing CQUAL. Appendix A gives a complete description of CQUAL's configuration files and surface syntax. We believe that the lessons learned while developing CQUAL are applicable to other languages as well. Section 5 describes a series of experiments using CQUAL.

4.1 Modeling C Types

In this section we discuss some of the issues in handling C types as they are used in C programs, which is somewhat more complicated than the idealized types previously presented.

L-Types and R-Types. In C there is an important distinction between *l*-values, which correspond to memory locations, and *r*-values, which are ordinary values like integers. In the C type system, *l*-values and *r*-values are given the same type. For example, consider the following code:

```
int x;
x = ...;
... = x;
```

The first line defines the variable **x** as a location containing an integer. On the second line **x** is used as an *l*-value: it appears on the left-hand side of an assignment,

³CQUAL also supports flow-sensitive type qualifiers [Foster et al. 2002].

meaning that the location corresponding to x should be updated. On the third line x is used as an r -value. Here when we use x as an r -value we are not referring to the location x , but to x 's contents. In the C type system, x is given the type `int` in both places, and the syntax distinguishes integers that are l -values from integers that are r -values.

In our formal type checking system (Figure 4), types are used to distinguish l -values and r -values, and this is what CQUAL actually implements. CQUAL gives the variable x (ignoring qualifiers for a moment) the type $ref(int)$, meaning that the name x is a location containing an integer. When x is used as an l -value its type stays the same—the left-hand side of an assignment is always a ref type. When x is used as an r -value the outermost ref is removed, i.e., x as an r -value has the type int .

Qualifier Annotations and Checks. In Section 2, we introduced type qualifier annotations and checks to specify and test the qualifier on a value. However, C source code already contains type information, and in fact allows a limited set of qualifiers to appear on types. Thus rather than add new syntactic forms for qualifier annotations and checks, CQUAL simply extends the set of qualifiers that can appear in types. To make lexing and parsing simple, CQUAL requires that all user-defined qualifiers begin with a dollar sign (not shown in this paper). The partial order among qualifiers, as well as other information described below, is supplied in a separate configuration file, fully described in Appendix A.

The next question that arises is how to interpret a qualifier appearing in a type in the surface syntax of C. There are two main issues. First, suppose we see a declaration `a int x`. Then we assign x the type $x \text{ ref}(x' \text{ int})$, where x and x' are fresh qualifier variables, which we then constrain by a . But should we interpret the occurrence of a as constraining x or as constraining x' ? In CQUAL, the user must specify this for each qualifier (in the configuration file). The qualifier a may be declared to constrain x , i.e., the ref level, or it may be declared to constrain x' , i.e., the int or $value$ level. Most qualifiers constrain the value level of a type; for example, *tainted* and *untainted* behave this way. The canonical example of a qualifier that constrains the ref level of a type is *const* (Section 5.1).

The second issue in interpreting a qualifier in the surface syntax is to decide whether it corresponds to a qualifier annotation, a qualifier check, or both. For example, if we see *tainted* `int x` in the source code, the qualifier occurrence should be equivalent to `annot(x, tainted)`. On the other hand, if we see *untainted* `int x`, that should be equivalent to `check(x, untainted)`. In our experience with the qualifiers discussed in Section 5, we have found that all occurrences of the same qualifier should be treated consistently, so in CQUAL the programmer specifies in the configuration file how to treat each qualifier.

Suppose there is declaration `a int x` where a is a value-level qualifier and x' is the value-level qualifier variable on x . Then if a is specified as *positive*, then the constraint $a \leq x'$ is generated, i.e., a is a qualifier annotation. If a is *negative*, then the constraint $x' \leq a$ is generated, i.e., a is a qualifier check. Otherwise, a may be *non-variant*, in which case both constraints are generated, corresponding to a check and an annotation.

Finally, it is worth mentioning that arguments in C are passed by value, and

hence function types contain the r -types of their declared parameters, even though within the body of a function the parameter is treated as having an l -type. For example, given the declaration `void f(int x)`, we assign `f` the type $x' \text{ int} \rightarrow \text{void}$ (ignoring qualifiers except on `x`), and within the body of `f` the variable `x` has type $x \text{ ref } (x' \text{ int})$.

Structures. One of the key considerations in any whole-program analysis of C code is how structures (record types) are modeled [Chandra and Reps 1999; Heintze and Tardieu 2001; Yong et al. 1999]. Suppose that the user declares a structure for `foo`:

```
struct foo {
    int x;
    int *y;
    ...
}
```

Then in theory, if we see two definitions `struct foo a` and `struct foo b` we can simply assign `a` and `b` two distinct copies of the type `struct foo`. Unfortunately, in practice this turns out to be prohibitively expensive. If `struct foo` has m fields and we assign each of n instances of `struct foo` fresh copies of the types of its fields, then we are doing $O(mn)$ work. Since many C programs contain extremely long structure type declarations and many instances of the same `struct`, m and n can be relatively large, causing a large slowdown in type qualifier inference. In this worst case, m and n may be linear in the size of the program, resulting in a quadratic algorithm.

We solve this problem by associating a mapping F_a from field names to qualified types with each occurrence `struct foo a` of type `struct foo`. This mapping is initially empty. When the programmer references `a.x`, we check for an entry $x \in F_a$ and, if such an entry is found, we use $F_a(x)$ as the qualified type for `a.x`. Otherwise, we create such an entry based on the declaration of field `x` in `struct foo` and add it to F_a . When we discover a subtyping relation between two structs, say `a` and `b`, we scan F_a and F_b and unify $F_a(x) = F_b(x)$ for any shared fields `x`. Finally, we set $F_a = F_b$. This approach has two advantages:

- The total number of qualified types created for structure fields is guaranteed to be linear in the size of the input program.
- Equating the field types does not lose much precision because most structures in C programs are referenced through pointers, and thus most subtyping relations between structure fields reduce to type equality.

Multiple Files. Very few C programs are contained within a single source file, and thus CQUAL is designed to perform type qualifier inference on multiple files simultaneously. We require that globals declared in multiple files have the same type, which can be achieved by unifying their types. Aggregates must be handled carefully. Suppose `struct foo` and `struct bar` are declared with exactly the same fields. In ANSI C equivalence of `struct` types is by name, and thus if `struct foo` and `struct bar` are declared in the same file, then they are considered different even though they are the same structurally. However, if they are declared in separate

files then a `struct foo` can be passed to a `struct bar` and vice-versa. Shared header files tend to reduce the incidence of this, but it does happen occasionally in practice. Thus CQUAL performs structural matching on `struct` and `union` types to determine equivalence. We also handle the special case where `struct foo` is declared with no fields in some files, a trick used to hide implementation details. When we unify two aggregates and one has no fields, we replace the empty aggregate with the other one.

Note that we do not require that the programmer analyze all files of a program together. However, to get safe results when analyzing a single file CQUAL must be supplied with full qualified type declarations for any undefined globals.

Parametric Qualifier Polymorphism. As discussed in Section 3.1, CQUAL includes parametric polymorphic recursive qualifier inference. In addition, we allow the programmer to directly assign a polymorphic type signature to a function. The given type signature is assumed to be correct, and any function with a polymorphic type signature is not type checked. As an example type, consider the C standard library function `char *strcat(char *dest, char *src)`, which appends `src` to the end of `dest` and returns `dest`. We can assign `strcat` the polymorphic type

$$\forall \kappa, \kappa' [\kappa' \leq \kappa]. \text{ref}(\kappa \text{ char}) \times \text{ref}(\kappa' \text{ char}) \longrightarrow \text{ref}(\kappa \text{ char})$$

which means that the qualifier on `strcat`'s second argument must be a subtype of its first argument, and that its first argument is returned. We omit the top-level `ref` qualifiers for clarity.

In the surface syntax, we declare this function with

```
$_1_2 char *strcat($_1_2 char *, $_1 char *);
```

The `$_1_2` and `$_1` are explicit qualifier variables representing the unordered sets $\{1, 2\}$ and $\{1\}$, respectively. We use the names of the qualifier variables to encode the subtyping constraints. We generate the constraint $\kappa \leq \kappa'$ if the set encoded in the name of κ is a subset of the set encoded in the name of κ' . The existence of the Dedekind-MacNeille Completion [Davey and Priestley 1990] implies that any set of subtyping constraints can be encoded this way. While this is not the most transparent representation of subtyping constraints, it has the advantage of requiring no changes to the surface syntax.

Subtyping Under Pointer Types with const. In Section 3.2 we argued that if there are no updates through a reference, we can use the deep subtyping rule (Ref'_{\leq}) in Figure 9 rather than the conservative (Ref_{\leq}) in Figure 4a. In ANSI C, programmers use `const` to annotate `l`-values that are never written. Thus in CQUAL we perform deep subtyping on locations explicitly annotated with `const` by the programmer. This is sound up to the unsafe features of C such as type casts. Although we could do so, we do not use the `const` inference described later to infer additional `l`-values that are not written to.

Arrays, Address-of, Function Pointers, and Memory Allocation. Finally, we discuss some additional language constructs and features. All elements of an array are given the same type qualifier, and we allow conversions between arrays and pointers as is standard [ANSI 1999]. Given our choice of distinguishing `l`- and `r`-values at

the type level, handling the address-of operation is straightforward: if a variable defined `int x` has *l*-type $x \text{ ref } (x' \text{ int})$, then `&x` is an *r*-value of exactly the same type. We assign function types to C functions just as in the language in Section 2. In ANSI C, function types are almost always immediately promoted to pointers to functions [ANSI 1999], and CQUAL behaves similarly. Since function pointers cannot be written through, we allow subtyping under function pointer types as discussed in Section 3.2. Note that we need not do anything special to handle calls through function pointers; the standard typing rules in Figure 5 infer types for a language with full higher-order functions, which subsumes ANSI C’s more limited function pointers. Finally, in C memory allocation is done via a library call to `malloc()`. CQUAL does not handle this function call in any special way, but our standard prelude file gives `malloc()` the type (ignoring the qualifiers on the function arrow and pointer)

$$\forall \kappa[\emptyset]. \text{size_t} \longrightarrow \text{ref } (\kappa \text{ void})$$

This is a slight abuse of parametric polymorphism, since κ is not bound in a parameter type, but it has the effect of giving each call to `malloc()` a fresh qualifier variable.

4.2 Unsafe Features of C

The C programming language contains many features that allow the programmer to violate memory and type safety. Some of the major holes are type casts, unions, variable-argument functions, and arbitrary pointer arithmetic. CQUAL incorporates a range of techniques in order to track qualifiers even through unsafe language constructs, although the techniques are neither sound nor complete.

Type Casts. Type casts allow a C programmer to treat a value as having any type they choose, which lets the programmer bypass limitations of the C type system. For example, a pointer to any type can legally be cast to and from a pointer to the special type *void*. Such casts are commonly used for generic functions on data structures. For example, a programmer may define a list data structure whose elements have type pointer to *void*, and then the same code for list operations can be used for lists of pointers to objects of any type.

By default, CQUAL assumes that type casts cast away the qualifiers as well as the types. For example, some type casts are added to programs exactly to cast away *const* qualifiers, and so it would be a bad idea to ignore such a cast in general. For other qualifiers, the programmer can tell CQUAL to model type casts by propagating qualifiers “through” the cast. For example, consider the following code:

```
a char *y;
void *x = (void *) y;
```

This code declares `y` to be a pointer to character, where the character has qualifier `a`, and then initializes `x`, which is a pointer to *void*, with `y`. If the programmer tells CQUAL to propagate the qualifier `a` through type casts, `x` is inferred to have type `a void *`.

More formally, consider a type cast $(\tau) e$, which has type τ , and let τ' be the type of expression e . For this type cast, CQUAL generates the special constraint

$$\begin{array}{c}
\frac{Q \leq_{S_{cast}} Q'}{Q \text{ int} \leq_c Q' \text{ int}} \qquad \frac{Q \leq_{S_{cast}} Q' \quad \tau \leq_c \tau' \quad \tau' \leq_c \tau}{Q \text{ ref}(\tau) \leq_c Q' \text{ ref}(\tau')} \\
\\
\frac{Q \leq_{S_{cast}} Q' \quad \tau'_1 \leq_c \tau_1 \quad \tau_2 \leq_c \tau'_2}{Q(\tau_1 \longrightarrow \tau_2) \leq_c Q'(\tau'_1 \longrightarrow \tau'_2)} \\
\\
\frac{\tau \leq_c Q_2 \nu \quad Q_2 \nu \leq_c \tau \quad (\nu \neq \text{ref}(\cdot))}{Q_1 \leq_{S_{cast}} Q_2 \quad Q_2 \leq_{S_{cast}} Q_1} \qquad \frac{\tau \leq_c Q_1 \nu \quad Q_1 \nu \leq_c \tau \quad (\nu \neq \text{ref}(\cdot))}{Q_1 \leq_{S_{cast}} Q_2 \quad Q_2 \leq_{S_{cast}} Q_1} \\
\frac{Q_1 \text{ ref}(\tau) \leq_c Q_2 \nu}{Q_1 \text{ ref}(\tau) \leq_c Q_2 \nu} \qquad \frac{Q_1 \nu \leq_c Q_2 \text{ ref}(\tau)}{Q_1 \nu \leq_c Q_2 \text{ ref}(\tau)} \\
\\
\frac{Q_1 \leq_{S_{cast}} Q_2 \quad Q_2 \leq_{S_{cast}} Q_1 \quad (\nu \neq \cdot \longrightarrow \cdot)}{Q_1(\tau \longrightarrow \tau') \leq_c Q_2 \nu} \qquad \frac{Q_1 \leq_{S_{cast}} Q_2 \quad Q_2 \leq_{S_{cast}} Q_1 \quad (\nu \neq \cdot \longrightarrow \cdot)}{Q_1 \nu \leq_c Q_2(\tau \longrightarrow \tau')}
\end{array}$$

Fig. 10. Rules for Handling Casts, including Shape Mismatches.

$\tau' \leq_c \tau$. When the shapes of τ' and τ match, this reduces to a set of gated qualifier constraints $Q \leq_{S_{cast}} Q'$ (Section 3.3), where S_{cast} is the set of qualifiers that propagate through casts. In order to handle type casts like the example above, the constraint \leq_c also allows matching between base types like *void* and *char*. Further, the constraint \leq_c allows matching between structurally dissimilar types. In particular, in C any pointer type may legally be cast to *void **. For example, a programmer might write

```

a char **s;
char **t;
void *v = (void *) s;
t = (char **) v;

```

Here *s* and *t* are pointers to pointers to character. Notice that the type structure of *v* and the type structures of *s* and *t* do not even have the same shape. To model these kinds of casts, CQUAL “collapses” the mismatched levels at type casts by equating their qualifiers. Figure 10 gives rules to conservatively equate qualifiers at shape mismatches. In these rules, for pointers, we match up as many levels of the pointer qualifiers as possible, then the qualifiers on any extra pointer levels of one type are equated with the qualifier on the base of the other. For our example above, inference determines that *v* has type *a void **, and both *s* and *t* have type *a char ** ** (both levels of pointers get qualifier *a*). When collapsing at casts, we do not collapse function argument qualifiers.

C programs cast pointers to and from integral types often enough that it is worth having special support for this programming idiom. We handle this case by treating every integral program variable as if it were a void pointer. For example, *int a* is given type *a ref(a' void)*. All the standard type inference rules for void pointers are applied to *a*. This technique captures most of the casts between pointers and integral types, especially when combined with the special handling of casts between void pointers and *structs* described below.

Another common C idiom is to cast structure pointers to void pointers. To improve the precision of our analysis, we let void pointers “masquerade” as pointers to any structure, and have the void pointer don the appropriate mask when interacting with a structure. More concretely, with each void pointer *p* we associate a map-

ping $T_p : \{\text{structure declarations}\} \rightarrow \{\text{qualified structure types}\}$. Initially, $T_p(x)$ is undefined for all x . Upon discovering a type relation between \mathbf{p} and some `struct foo *a` with type $a \text{ ref } (\tau_a)$, we first look up $\tau = T_p(\text{struct foo})$. If τ is defined, we generate the constraint $\tau = \tau_a$. Otherwise, we set $T_p(\text{struct foo}) = \tau_a$. When we discover a subtyping relation between two void pointers $\mathbf{p1}$ and $\mathbf{p2}$, we unify the maps T_{p_1} and T_{p_2} in the obvious way. This approach is not always safe, since programmers can convert from one struct type to another via a void pointer, and no constraints between the fields of the struct types will be generated. This is a relatively rare operation in many coding styles, though, and so we believe that this choice balances safety and precision.

C also permits casts between structures that do not have identical definitions. One structure may have more fields than another, or corresponding fields may differ in their type and, importantly, their size in memory. For these kinds of casts, CQUAL matches up as many fields as possible and ignores any extra fields that only appear in one of the structs. Also, the fields are matched by their index in the structure, i.e., the first fields are paired, then the second fields, etc., instead of matching by byte-offset.

Sometimes casts to discard qualifiers are useful. CQUAL assumes that any cast to a type that contains an explicit qualifier should stop propagation of any other qualifiers in the same partial order. For example, in the following code

```
a char *y;
void *x = (b void *) y;
```

the variable `x` is inferred to have qualifier `b` but not `a`. Such “trusted casts” are essential for making CQUAL usable in practice by allowing the programmer to suppress false warnings. For example, in our experiments in Section 5.2, we needed to use trusted casts in a few places to remove warnings that did not correspond to format-string vulnerabilities. In our experience, trusted casts typically do not need to be used heavily, and of course great care must be taken when they are used.

Unions and Pointer Arithmetic. We make the same assumptions as the C standard about unions and pointer arithmetic. Namely, we model unions in the same way we model structures, and we assume that values of a union type are always accessed at the correct type with the correct qualifiers. We assume that pointer arithmetic does not violate object bounds and is only used to access array elements, rather than jump to a value of a different type. In other words, if p is a pointer to type τ , then we assume $p + i$ for any integer i also has type τ .

Libraries. Most C programs make some use of the extensive set of standard C libraries. Unfortunately, we do not necessarily have source code for library functions. Thus we require that the programmer supply a model for any library function that has an effect on the qualifiers. This model is usually a small stub function that mimics the behavior of the library function with respect to the qualifiers. Alternately, programmers may also supply polymorphic type signatures for functions in lieu of a stub function. In order to make it easy to identify library functions, CQUAL provides the programmer with a list of all globals that are used but never defined.

Variable-Argument Functions. In C, functions can be declared to take a variable number of arguments using the *varargs* language feature. One major problem with varargs functions is that there is no way to specify types for the variable arguments. CQUAL extends the grammar for C types to allow a qualifier constant or a polymorphic qualifier variable (Section 4.1) to be associated with the variable arguments. When the varargs function is called, we make constraints between that qualifier constant or polymorphic qualifier variable and all qualifiers on all levels of the actual arguments. For example, for the tainting analysis described in Section 5.2, we give the `sprintf` function the polymorphic type

```
int sprintf(char $_1.2 * str, const char untainted * format, $_1 ...);
```

meaning that for any qualifier κ on any varargs parameter, $\kappa \leq \kappa'$ where κ' is the qualifier on the contents of `str`. To avoid unnecessary conservatism, we only generate such constraints for varargs functions that have explicitly marked varargs qualifiers. CQUAL provides a list of all undefined varargs functions to the user.

4.3 Presenting Qualifier Inference Results

Unlike traditional optimizing compiler technology, in order to be useful the results of the analysis performed by CQUAL must be presented to the user. We have found that in practice this often-overlooked aspect of program analysis is critically important—a user of CQUAL needs to know not only what was inferred but why it was inferred, especially when the analysis detects an error. To address this issue, CQUAL presents type qualifier inference results to the user via Program Analysis Mode (PAM) for Emacs [Harrelson 2001]. PAM was developed concurrently with CQUAL, based on an earlier version that was part of the BANE toolkit [Fähndrich 1999]. PAM is a generic system for adding color markups and hyperlinks to program source code in Emacs. The ideas behind PAM, inspired by the MrSpidey system [Flanagan et al. 1996], can be adapted to many environments, and an Eclipse interface is also available [Greenfieldboyce and Foster 2004].

After CQUAL analyzes the source programs, the user is presented with a buffer containing a list of the files that were analyzed and a list of any errors. Each file name in the buffer is a hyperlink to the start of the source file. Each error lists an inconsistent path through the qualifier constraint graph (see below) and has a hyperlink to the line and column in the source code where the error was discovered. When the user clicks on a hyperlink to bring up a file, the preprocessed source code of the file is colored according to the inferred qualifiers, with colors specified in the configuration file. If an identifier is inferred to have a particular qualifier, it is given that qualifier's color. If a qualifier variable is unconstrained by the program (or has constraints that are meaningless, such as $\kappa \leq \top$), then it is not colored. CQUAL presents preprocessed source code because otherwise, due to C preprocessor macro expansions, jumping to particular line and column positions and marking up identifiers would not always be possible (for example, macro expansion can introduce new identifiers not present in the original source).

For each identifier in the program, CQUAL tries to show the user how its qualifiers were inferred. Clicking on an identifier brings up its type and qualifier variables. and clicking on a qualifier variable shows a path through the qualifier constraint graph that entails the inference result. Figure 11 shows a screen shot of CQUAL

The screenshot shows a window titled "File Edit Options Buffers Tools PAM Help". The top section displays C code for a program named "taint1.c":

```
# 1 "/home/rtjohnso/projects/cquals/fields2/cqual/examples/taint1.c"
$tainted char *getenv(const char *name);
int printf($untainted const char *fmt, ...);

int main(void)
{
  char *s, *t;
  s = getenv("LD_LIBRARY_PATH");
  t = s;
  printf(t);
}
```

The bottom section shows a constraint graph for the variable `*t`:

```
*t: $tainted $untainted

$tainted <= *getenv_ret
           <= *getenv_ret@8
           <= *s
           <= *t
           <= *printf_arg1@10
           <= *printf_arg1
           <= $untainted
```

At the bottom, there is a section labeled `** *Types*` with the text "(Fundamental PAM)--L10--A11--".

Fig. 11. Sample Run of CQUAL

displaying such a path. In this example the result of `getenv` is annotated as *tainted*, and `printf` is annotated as taking an *untainted* first argument (see Section 5.2 for a discussion of these particular qualifiers). The result of `getenv` is passed to `s`, which is copied to `t`, which is passed as the first argument to `printf`. The screen shot in Figure 11 shows what happens when the user clicks on one of `t`'s qualifier variables `*t`: CQUAL presents the user with a path from *tainted* to `*t` and from `*t` to *untainted*. In this particular case the path indicates an error, since *tainted* $\not\leq$ *untainted* in the partial order. This path is also available on the initial screen when CQUAL is first run. To make the paths even more useful, in CQUAL each element of the path, which represents a constraint, is hyperlinked to the position in the source code where that constraint was generated. In this way the programmer can step through a path one constraint at a time, viewing each line of source code that led to a particular inference result.

In general, for a given qualifier variable x , CQUAL presents the user with the shortest transitive paths (possibly bidirectional for non-variant qualifiers) from x to any qualifier constants appearing in x 's solution. Clearly there could be many paths, some of which may be cyclic, from x to its bounds. We settled on presenting the shortest path as a way of reducing the burden on the user. In our experience, this heuristic is very important for usability.

As explained above, CQUAL presents a list of type qualifier errors to the user for evaluation. A naive algorithm for generating type qualifier errors would be to solve the generated constraints on-line, signaling an error whenever newly generated constraints are unsatisfiable. As we have discovered, using this approach a single program error can result in thousands of type errors. The problem is that the original error can “leak out” to rest of the program and pollute the inferred types, because once a portion of the constraint graph is unsatisfiable, any constraints that interact with that portion of the graph typically also become unsatisfiable. Displaying these extraneous errors would not only be overwhelming, but would

Name	Warnings	
	Unfiltered	Filtered
bftpd-1.0.11	4	1
cfengine-1.5.4	5261	3
muh-2.05d	20	2

Fig. 12. Tainting Analysis Error Filtering Results

also make it difficult for the programmer to find the root cause of the error, since most of the warnings would be only remotely related to the original programming mistake.

CQUAL reduces the number of error messages by solving the constraints off-line, after all constraints have been generated, and then using heuristics to decide where to report error messages. For *derivative* errors, which result when one error spreads through a large portion of the constraint graph, CQUAL uses the following heuristic. Let x be a type variable with error path $l \leq l_1 \leq \dots \leq l_n \leq x \leq u_1 \leq \dots \leq u_m \leq u$, where l and u are inconsistent constant qualifier bounds on x . Then CQUAL considers the type error on x to be *derivative* if there exists some $x' = l_i = u_j$. In this case, the error on x is probably just a side-effect of the error on x' , and so CQUAL only reports an error involving x' and not one involving x .

CQUAL also contains a heuristic for eliminating *redundant* error messages. Even if the path $l \leq l_1 \leq \dots \leq l_n \leq x \leq u_1 \leq \dots \leq u_m \leq u$ does not satisfy the derivative condition, printing out a warning message for each l_i, u_j , and for x would give the programmer little new information. Thus CQUAL suppresses these useless additional warnings, and only reports one error for this path. Lastly, CQUAL flags type variables corresponding to intermediate values in the program as *anonymous*, and CQUAL will initially only report warnings for positions involving non-anonymous variables. In the unlikely event that all type errors involve only anonymous variables, CQUAL disables this heuristic and reports warnings involving anonymous variables.

Despite their ad-hoc nature, these heuristics have proven extremely effective. Figure 12 shows that these error filtering techniques can reduce the number of warnings by over three orders of magnitude. (See Section 5.2 for more information on the tainting analysis used in this benchmark.) Furthermore, in our experience the warnings that are produced have pointed us directly to the source of the error. In most cases, we have found that fixing the errors that CQUAL displays has resulted in a program that type checks. Thus these heuristics seem to do a very good job of making CQUAL display exactly one useful error message for each programming error. Overall, we have found these heuristics to be indispensable to making CQUAL much more usable.

One of the main problems in presenting analysis results is that for a large input program, there is a correspondingly large amount of information we may wish to present to the user. This information is usually represented compactly during analysis, but if represented textually it becomes extremely unwieldy. Clearly this is the case here: the constraint graph is relatively compact, but writing out all paths from qualifier variables to their bounds would be prohibitively expensive.

PAM sidesteps this problem completely by using a client-server architecture. PAM runs CQUAL as a subprocess. As the user clicks on hyperlinks in PAM buffers,

PAM passes the click events to the CQUAL subprocess, which then sends commands to PAM to move the cursor position, display additional screens of information, and so on. In this way CQUAL maintains the inference results in its internal, compact form, and the results are presented verbosely only on demand by the user.

5. EXPERIMENTS

In this section we describe three experiments using CQUAL. All experiments were performed on a dual processor (though CQUAL is single-threaded) Pentium Xeon 2GHz machine with 2GB of physical memory and hyperthreading enabled.

In the first experiment, we infer *const* qualifiers for C programs. In this case, we treat assignment statements as if they had the form `check(e_1 , nonconst) := e_2` , to require that the left-hand side of every assignment is not *const*, and then we infer the maximum number of *consts*. In the second experiment, we find security vulnerabilities by checking whether *tainted* (untrusted) data might ever flow to *untainted* positions. In this case, we constructed a header file that marks the return values of untrusted functions with *tainted*, corresponding to `annot(\cdot , tainted)`, and marks trusted argument positions with *untainted*, corresponding to `check(\cdot , untainted)`. Finally, in the last experiment, we check that after initialization the Linux kernel does not use functions or data whose memory space has been reclaimed. In this case, we treat existing annotations marking initialization data as qualifier annotations `annot(\cdot , init)`, and for any function live after initialization time we add constraints equivalent to `check(\cdot , noninit)`.

5.1 Const Inference

In ANSI C, the *const* qualifier can be added to types to specify that certain updatable references cannot, in fact, be updated. For example, if the programmer defines

```
const int x = 42;
```

then any assignment to `x`, such as `x = 3`, is forbidden. In this section we discuss checking and inferring ANSI C *const* qualifiers with CQUAL.

In ANSI C, locations that can be written are simply unannotated. In our framework, we add a qualifier *nonconst* to make this explicit. As in ANSI C, we also allow *nonconst* types to be used where *const* types are expected, i.e., the partial order among these two qualifiers is *nonconst* < *const*. For purposes of this experiment, we added a small amount of code to CQUAL to count how many positions (see below) are not reachable from *nonconst*, i.e., how many positions can be made *const*.

As mentioned earlier, in our system *const* and *nonconst* are ref-level qualifiers. To understand why, consider the following C program:

```
const int x = 42;
int y;
y = x;
```

This program is valid: we read the value of `x`, which is allowed, and we write its value to `y`, which is allowed because `y` is *nonconst*. In the language of Section 3 the assignment would be written `y := *x`, where `y` has type *nonconst ref(int)* and `x` has type *const ref(int)* (omitting the qualifier on *int*). The assignment requires

that y is *nonconst* (which it is), but also that the type of the right-hand side (*int*) is a subtype of the left-hand side contents type (also *int*). If instead we had opted to put *const* and *nonconst* on the value level, then our type rules would reject this program because $\text{const } int \not\leq \text{nonconst } int$. Thus in our system, the *const* qualifier never appears on anything other than a *ref* constructor. Expressions that are only *r*-values, like the integer 3, do not have a *ref* type, and thus *const* does not apply to them.

The main use of *const* is annotating the types of pointer-valued function parameters. Below is a table listing which assignments are allowed by the four possible placements of *const* on the type pointer to integer. Recall that C types are most easily read from right to left; thus, for instance, the second example below can be read as defining a pointer to a constant integer.

Definition	$\mathbf{p} = \dots$	$\mathbf{*p} = \dots$
<code>int *p</code>	valid	valid
<code>const int *p</code>	valid	invalid
<code>int *const p</code>	invalid	valid
<code>const int *const p</code>	invalid	invalid

Suppose that the programmer declares a function `void f(const int *p)`. Looking at our table, we see that the caller of `f` knows that, up to casting, `f` does not write through its argument `p`. This annotation is quite useful, since it means that one may freely pass pointers as arguments to `f` without fearing that the data they point to will be modified through `p`. Note that *const* does not guarantee that `*p` is not modified through other aliases, but only that it is not modified through `p`.

Making *const* and *nonconst* ref-level qualifiers also illuminates the suspicious-looking subtyping among pointers in C. Consider again our function `f(const int *p)`. With subtyping, both a *nonconst int ** and a *const int ** may be passed to `f`, and neither can be written to by `f` via `p`. This is exactly what we want, but in the C syntax it looks like we are performing subtyping under a *ref*, which would be unsound. However, in our notation we are passing type *nonconst ref(int)* to *const ref(int)*, and that is allowed by the standard rule for subtyping references from Figure 4a.

To enforce the semantics of *const*, we can transform the input program by replacing each assignment statement $e_1 := e_2$ with `check(e_1 , nonconst) := e_2` , which requires that the left-hand side of every assignment is not *const*. Alternatively, we can modify the rule for assignment from Figure 5 to require the same thing:

$$\frac{\Gamma \vdash e_1 : Q \text{ ref } (\tau) \quad \Gamma \vdash e_2 : \tau' \quad \tau' \leq \tau \quad Q \leq \text{nonconst}}{\Gamma \vdash e_1 := e_2 : \tau}$$

The latter is in fact what we do in CQUAL. We also make *const* a positive qualifier, so that occurrences of it in the program correspond to qualifier annotations.

Given an input program, we can assume that every position without a *const* qualifier has an implicit *nonconst* qualifier, just like a C compiler, and using our new rule for assignment CQUAL can check that a program uses *const* correctly. Unlike an ordinary C compiler, however, CQUAL can do better: we can use flow-insensitive type qualifier inference to infer *const* annotations.

Name	Description	Lines	Preproc.
woman-3.0a	Manual page viewer	1496	8611
patch-2.5	Apply a diff	5303	11862
m4-1.4	Macro preprocessor	7741	18830
diffutils-2.7	Find diffs between files	8741	23237
ssh-1.2.26 ⁵	Secure shell	18620	127099
uucp-1.04	Unix to unix copy	36913	272680

Fig. 13. Const Inference Benchmarks

A system that performs *const* inference has many benefits for the programmer. Although use of *const* is considered good programming style, it is well-known folklore that *const* is difficult to use in practice. Often an attempt to add a single *const* annotation to a program requires adding many other *consts* throughout the code. For the same reason, it can be difficult to mix code that uses *const* with code that does not. The result is that it is often easiest to simply omit *const* annotations altogether.

To perform *const* inference using CQUAL, we do not assume that any position without a *const* qualifier is *nonconst*. Instead we make no constraint on the qualifier variables in such positions. Then, given our new rule for assignment, we infer which qualifier variables must be *nonconst*. All of the remaining qualifier variables may be set to *const*. Note that we are actually computing the greatest solution of the generated qualifier constraints, since *nonconst* < *const*.

In Section 4.2 we describe some techniques for handling unsafe features of C. For purposes of these experiments we simply model unsafe features unsafely. We allow type casts to remove *const* qualifiers—we must do this, since many such casts are added precisely to remove *const*—and we do not perform any type checking on the extra arguments passed to varargs functions. We supply program stubs for library functions, and we make the conservative assumption that positions not marked *const* are indeed *nonconst* for such functions, as are all fields of structures used by library functions. In general library functions are annotated with as many *consts* as possible,⁴ and so lack of *const* really does mean *nonconst*.

We ran *const* inference on six programs, listed in Figure 13, that make a significant effort to use *const*. Several of these “programs” are actually collections of programs that share a common code base. We list program size in terms of original (non-comment, non-blank) lines of code and preprocessed lines of code. We analyzed each set of programs simultaneously, which occasionally required renaming as distinct certain functions that were defined in several files. We measured the number of *const* pointers in function signatures, i.e., the number of *const* annotations inferred “under” pointer types, which are the annotations most likely to be useful for a programmer. For example, for function `int foo(int x, int *y)`, there is

⁴...and sometimes more. For example, the `strchr` function is declared `char *strchr(const char *s, int c)`. The call `strchr(s, c)` returns a pointer somewhere in `s`, and yet the return type lacks *const*. This implicit cast is a way to emulate parametric qualifier polymorphism.

⁵The ssh distribution also includes a compression library `zlib` and the GNU MP library (arbitrary precision arithmetic). We treated both of these as unanalyzable libraries; `zlib` contains certain structures that are inconsistently defined across files, and the GNU MP library contains inlined assembly code.

Name	Declared	Inferred		Max	Mono		Poly	
		Mono	Poly		Time	Mem	Time	Mem
woman-3.0a	50	64	69	95	0.31s	27M	0.34s	29M
patch-2.5	84	96	103	148	0.43s	35M	0.49s	39M
m4-1.4	88	239	251	370	0.77s	58M	0.85s	64M
diffutils-2.7	153	210	242	372	0.81s	61M	0.90s	66M
ssh-1.2.26	154	303	332	561	4.03s	272M	4.44s	298M
uucp-1.04	433	1161	1288	1773	16.37s	1031M	17.71s	1080M

Fig. 14. Const Inference Results

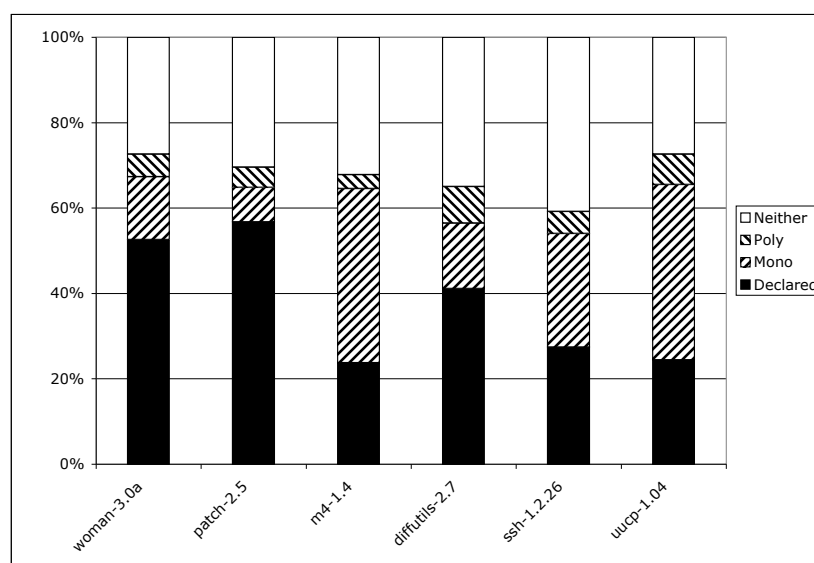


Fig. 15. Graph of Const Inference Results

one potential *const* pointer, namely *y*.

Figure 15 shows our results, which are tabulated in Figure 14. The second column of the table in Figure 14 lists the number of *const* pointers on function signatures declared by the programmer. The third and fourth columns list the number inferred by *const* inference (which includes the explicitly specified ones) and the number inferred by polymorphic *const* inference (with the library functions effectively monomorphic), and the fifth column lists the total number of possible *const* pointer positions on function signatures. The last columns list the running time and memory usage of the monomorphic and polymorphic analyses, which are the average of three runs. These measurements show that many more *consts* can be inferred than are typically present in a program, even one that makes a significant

effort to use *const*. For some programs the results are quite dramatic, notably for `uucp-1.04`, which can have more than 2.5 times more *consts* than are actually present. An inspection of the code suggests that *const* is used consistently only in certain portions of the code, and that other parts of the code make no use of *const*. Additionally, the program uses several `typedefs` to define new names for pointer types. Because we allow different instances of the same named type to have different qualifiers, we are able to infer that some uses of those pointer types can have *const* annotations. Clearly this is a case where *const* inference is very desirable. Faced with a program that heavily uses a single named type, few programmers would attempt to introduce a new type name with *const* annotations, but inference makes that process easy.

For this set of benchmarks polymorphic analysis allows 5-15% more *consts* than monomorphic analysis. These results show that qualifier polymorphism is both useful and already latent in C programs, although we believe that most of the benefit for polymorphism comes from allowing fewer type casts rather than more *consts*.

5.2 Format-String Vulnerabilities

Systems security is an ever more important problem as more critical services are connected to the Internet. Systems written in C are a particularly fruitful source of security problems, due to the tendency of C programmers to sacrifice safety for efficiency and the sometimes subtle interactions of C language and library features. One class of C security problems is the so-called format-string vulnerability, which arises from the combination of unchecked variable argument (`varargs`) functions and standard C library implementations.

The standard ANSI C libraries contain a number of `varargs` functions that take as an argument a format specifier that gives the number and types of the additional arguments. For example, the standard printing function is declared as

```
int printf(const char *format, ...);
```

When `printf(format, a1, a2, ...)` is called, the string `format` is displayed with the *i*th format specifier replaced by extra argument `ai`. For example, here is the typical, correct way to print a string `buf`:

```
printf("%s", buf);
```

But for simply printing a string, the above construction appears at first to be unnecessarily verbose. A programmer can save themselves five characters—and possibly some whitespace—if they instead write

```
printf(buf);    /* may be incorrect */
```

Unfortunately, this innocuous-looking change may lead to security problems. If `buf` contains a format specifier (for example, `%s` or `%d`), perhaps supplied by a malicious adversary, `printf` attempts to read the corresponding argument off of the stack. Since there is no corresponding argument, `printf` will mostly likely crash, either when reading off the end of the stack or when it attempts to treat the garbage off the end of the stack if as it were a pointer to a null-terminated string.

```

char *getenv(const char *name);
int printf(const char *fmt, ...);

int main(void)
{
    char *s, *t;
    s = getenv("LD_LIBRARY_PATH");
    t = s;
    printf(t);
}

```

Fig. 16. Program with a Format-String Vulnerability

It turns out that format-string vulnerabilities are even worse than they first appear. Many implementations of the C standard libraries support the `%n` format specifier, which is now part of the ANSI C standard [ANSI 1999]. When a `printf`-like function encounters a `%n` format specifier, it writes through the corresponding argument, which must be a pointer, the number of characters printed so far. Given the ability to write to memory, a clever adversary can often exploit format-string vulnerabilities to completely compromise security—for example, to gain remote root access [Newsham 2000]. Since the ability to exploit format-string vulnerabilities was discovered in 2000, security experts and malicious attackers have discovered many such vulnerabilities in widely-deployed, security-critical systems. Unfortunately, it is too restrictive to merely forbid non-constant format-strings, and clearly the `%n` specifier cannot be eliminated, given that it is part of the standard.

Format-string vulnerabilities are one of a wider class of security bugs that can occur in any language. When programmers write security-conscious programs, they should distinguish two different classes of data: *untrusted* data read from the external environment should never be passed unchecked to functions requiring *trusted* data. In our case, untrusted data should never be used directly as a format specifier. We can track the trust level of data in CQUAL by introducing the qualifier *tainted* to mark untrusted data and *untainted* to mark trusted positions. It is safe to interpret untainted data as tainted but not vice-versa, hence we choose *untainted* < *tainted* as our partial order. Furthermore, we choose that *tainted* is a positive qualifier (so that occurrences of it correspond to `annot(·, tainted)`), and we make *untainted* a negative qualifier (so that occurrences of it correspond to `check(·, untainted)`).

As an example use of these qualifiers, consider the simple program shown in Figure 16. This program calls `getenv` to return the value of an environment variable, which is then stored successively in `s` and then `t`, and finally is passed as a format specifier to `printf`. Assuming we do not trust the user’s environment variables, this program has a format-string vulnerability. Indeed, on one system we tried, setting `LD_LIBRARY_PATH` to a string of eight `%s`’s causes this program to have a segmentation fault.

To detect this format-string vulnerability, we annotate the program as shown in the top half of Figure 11 on page 28. Marking the result of `getenv` with *tainted* produces the constraint $tainted \leq getenv_ret'$. Marking the format-string argument of `printf` as *untainted* produces the constraint $printf_arg0' \leq untainted$. Notice

Name	Description	Lines	Preproc.
identd-1.0.0	Network id service	223	1224
mingetty-0.9.4	Remote terminal controller	270	1599
bftpd-1.0.11	FTP server	2323	6032
muh-2.05d	IRC proxy	3039	19083
cfengine-1.5.4	Sysadmin tool	26852	141863
imapd-4.7c	UW IMAP4 server	21796	78049
ipopd-4.7c	UW POP3 server	20159	78056
mars_nwe-0.99	Novell Netware emulator	21199	72954
apache-1.3.12	HTTP server	32680	135702
openssh-2.3.0p1 ⁶	Secure shell	25907	218947

Fig. 17. Format-String Vulnerability Detection Benchmarks

Name	Warn	Bugs	Mono		Poly	
			Time	Mem	Time	Mem
identd-1.0.0	0	0	0.05s	7M	0.05s	7M
mingetty-0.9.4	0	0	0.06s	8M	0.06s	9M
bftpd-1.0.11	1	1	0.20s	18M	0.21s	20M
muh-2.05d	2	~ 2	0.56s	42M	0.56s	45M
cfengine-1.5.4	5	3	6.89s	464M	8.26s	501M
imapd-4.7c	0	0	6.44s	404M	8.28s	478M
ipopd-4.7c	0	0	6.57s	404M	8.66s	471M
mars_nwe-0.99	0	0	2.90s	186M	3.12s	207M
apache-1.3.12	0	0	7.12s	478M	8.15s	533M
openssh-2.3.0p1	0	0	14.13s	955M	14.17s	970M

Fig. 18. Format-String Vulnerability Detection Results

that we need not annotate the types of \mathbf{s} or \mathbf{t} . When CQUAL performs inference on this program, the generated qualifier constraints are inconsistent, meaning that *tainted* data is passed to an *untainted* argument, i.e., that this program may have a format-string vulnerability. The bottom half of Figure 11 displays the set of inconsistent constraints, and as mentioned in Section 4.3 the user can explore this error path to discover why type qualifier inference failed.

We used CQUAL to check for format-string vulnerabilities in ten popular C programs. For this experiment, we enable flow of qualifiers through casts (Section 4.2) to model taint propagation conservatively. We add *tainted* and *untainted* to programs by supplying a header file that contains declarations of the standard C library functions with the appropriate qualifiers and the appropriate parametric polymorphic types (Section 4.1). We use the same file of annotated library functions for all of our benchmarks. For one benchmark we also annotated two application-specific memory allocation and deallocation functions as polymorphic.

Figure 17 lists our benchmarks. All of these programs read data from the network, possibly controlled by a malicious adversary, and hence all could potentially have format-string vulnerabilities. For each program we list the numbers of lines of source code, both before and after preprocessing. The results of applying CQUAL are shown in Figure 18. The second and third columns list the number of warnings reported by monomorphic CQUAL and the number of actual format-string bugs

⁶We checked for vulnerabilities in the SSH daemon portion of the code.

discovered.⁷ The results for the polymorphic analysis are discussed below. The remaining columns list the running time and memory usage for monomorphic and polymorphic qualifier inference.

For most of these programs CQUAL issues no warning, indicating that the presence of a format-string bug is unlikely. This is especially interesting for two of our test cases, `mars_nwe` and `mingetty`, which contain suspicious-looking calls to a function that accepts format-strings [Huuskonen 2000b; 2000c]. Since we originally studied these programs, the `mars_nwe` code has been patched, and the suspicious-looking call has been said to be fully exploitable [Frasunek 2001a]. Because CQUAL does not model internal compiler functions to read variable arguments,⁸ we believe CQUAL may be wrong in this case, though the patch for `mars_nwe` did not give any details about an exploit and stated there were no working exploits yet [Frasunek 2001b]. The `mingetty` program has also been patched in some distributions, although at least one patch says that the code cannot be abused “to the best of [the writer’s] knowledge” [Huuskonen 2000a].

CQUAL finds potential format-string vulnerabilities in three of the programs. For `bftpd`, the one warning corresponds to one format-string vulnerability, and after eliminating the vulnerability (by replacing a function call of the form `sendstrf(s, entry->d_name)` with a call `sendstrf(s, "%s", entry->d_name)`), CQUAL produces no more warnings. With polymorphism, again CQUAL reports one warning corresponding to the same vulnerability, with no additional warnings after we eliminate the vulnerability. (The displayed error path is changed slightly, as is the case whenever we use polymorphism.)

For `muh`, the first run of CQUAL produces one warning corresponding one format-string vulnerability, and then the second run, after eliminating the vulnerability, produces another warning. A brief inspection of the code at the second warning site suggests it may have a vulnerability, because according to the analysis it has a possibly-tainted string in a format string position (but see below). After eliminating the potential vulnerability, CQUAL produces no more warnings. Interestingly, with polymorphism CQUAL initially reports one error corresponding to the same initial format-string vulnerability. But after eliminating that first vulnerability, no more errors are reported—thus the second warning in the monomorphic case was a false positive. Investigating further reveals that the warning can be suppressed in the monomorphic case by annotating three functions as polymorphic. Thus the second bug, although it appeared to be a vulnerability, was in fact not one. However, the code could change in the future, and since the transformation to eliminate the vulnerability does not affect the semantics of the program, it seems worth doing even though it is not necessary.

For `cfengine`, three warnings appear in the first run, and one additional warning appears in the second and third runs. After eliminating the three format-string vulnerabilities, we needed to do a little more work. One warning was produced by an incorrect call to `sprintf` that was simply a bug (the first argument to `sprintf` had

⁷CQUAL also reports a warning about `const` for both `apache` (a false warning) and `openssh` (a correct warning the compiler also flags).

⁸In `gcc` the important function is `__builtin_next_arg`; in other compilers different techniques, such as pointer arithmetic, are used to access varargs.

been omitted), so to eliminate that warning we corrected the bug. To suppress the final warning, we needed to annotate two functions with *const* parameters, declare one function to be polymorphic, and add three typecasts removing tainting from a single character extracted from a *tainted* string. Without a deep understanding of the code we cannot be sure these typecasts are safe, though it is also not obvious that they are unsafe, and hence it may be better to leave these warnings in. With polymorphism, the first run reports two warnings (corresponding to the buggy call to `sprintf` and a vulnerability), and two successive runs each report one warning corresponding to one remaining format-string vulnerability. After these runs, there are no further warnings. Thus an alternative solution to the suppress the extra monomorphic warnings would be adding parametric polymorphism in more places instead of adding *const* and two type casts.

In the case of `muh`, we knew beforehand [Henrion 2000] that these vulnerabilities were present in the code. In the cases of `cfengine` [Savola 2000] and `bftpd` [Bailleux 2000], the vulnerabilities were unknown to us at the time, although we subsequently discovered that these bugs had been previously reported. Nevertheless, this suggests that our tool is effective in finding unknown format-string vulnerabilities.

5.3 Initialization in the Linux Kernel

In order to reduce its memory footprint, the Linux kernel frees much of its initialization code and data after the initialization phase has completed. To support this explicit garbage collection step, all functions that the programmer believes are used only during the kernel initialization phase are declared with a special flag `__init`. The compiler places the code for all `__init` functions (those that have been annotated with `__init`) in the `.text.init` section of the executable (instead of in the `.text` section), and after the kernel has finished all initialization it deallocates the `.text.init` section. Calling an `__init` function after this point is very dangerous, since the code implementing that function may have been overwritten. Similarly, global variables can be explicitly declared to reside in a `.data.init` section that is deallocated after initialization, and referencing these variables after the initialization phase has completed is a dangerous error.

To check for correct usage of `__init` code and data, we must reason about not just the parameter and return types of a function, but also what code and data it may access during execution. Thus we model `__init` using effect qualifiers, introduced in Section 3.3. Each variable and function in the Linux kernel is assigned one of two effect qualifiers, either *init* (corresponding to the `__init` annotation) or *noninit*, indicating whether or not it survives after the initialization phase. Since *noninit* functions and data are available for use at all times, *noninit* items can be used wherever an *init* item is expected. Thus *noninit* < *init*. When annotating a variable declaration, *init* and *noninit* describe a property of *l*-values, and thus they are ref-level qualifiers, i.e.,

```
init int y;
```

indicates that `y` has the qualified type *init ref (int)*. We require the user to declare that the effect of a dereference is the qualifier on the dereferenced pointer:

```
$$a _op_deref ($$a *$1 x) $1;
```

Here $\$a$ is a polymorphic type variable and $\$_1$ is a polymorphic qualifier variable (Section 4.1). Thus, this declaration states that the dereference operator takes a pointer to any type and returns an object of that type, and has as its effect the qualifier $\$_1$ on x . This use of standard type polymorphism is restricted to this special case, as well as a few other built-in C operators. We use a separate declaration for this fact, rather than hard-coding it into the type system, to maintain maximum flexibility. In addition to this analysis-specific annotation, this experiment also used polymorphic annotations for common kernel functions such as `memcpy`. The `_op_deref` annotation was the only *init*-specific annotation required for this analysis, since the kernel source already contains `__init` annotations.

We specify that *init* is a positive qualifier (corresponding to `annot(·, init)`) and *noninit* is a negative qualifier (corresponding to `check(·, noninit)`). The *init* analysis also makes use of two of the well-formedness constraints described in Section 3.3. First, observe that a structure and its fields must all reside in the same section. Thus, for the *init* analysis, the qualifier on a pointer to a structure must be the same as the qualifiers on all the pointers to its fields. Certain untypeable operations in C may result in an *init* or *noninit* qualifier on a structure. For example, if the programmer uses `memcpy` to copy an array of *init* pointers into a structure, then standard type-inference rules will conclude that the structure is *init*. Note that when this happens, the fields of that structure are *init* pointers. The *init* analysis captures these semantics by using a well-formedness constraint: if a structure is *init* or *noninit*, then so are all its fields.

With these annotations, CQUAL can infer the effect of every function in the Linux kernel. The only thing left to do is enforce the kernel policy regarding *init* functions: no *init* function is called after the initialization phase has ended. Although this rule is sufficient to ensure the kernel has no *init* bugs, the kernel developers tend to follow an even stricter policy: every function not explicitly annotated with *init* should be safe to call after the initialization phase. To implement this policy, for each function `foo` not explicitly marked with *init*, we create a dummy function `foo'` that calls `foo`, and assert that `foo'_effect ≤ noninit`. Notice that this technique allows `foo` itself to be polymorphic in its effect, rather than always being *noninit*. Also notice that this rule is slightly conservative, since it may be that `foo` is never actually called after initialization time. However, if that is the case then `foo()` could be explicitly declared *init* to save memory, so issuing a warning in this case is still useful to the programmer.

We ran CQUAL on each source file in Linux kernel 2.6.0-test6, one at a time. (Currently CQUAL runs out of memory if we try to analyze the whole kernel, including all device drivers, at once.) Unfortunately for separate analysis, in the kernel sources *init* annotations are placed only on function and data definitions, and not on declarations or on function pointer parameters. We optimistically assume that functions and data that are declared but not defined are *noninit*, as are any function pointers. This optimistic assumption is not always safe, meaning that we will not be able to find all misuses of *init*, but in practice the results are still very useful. In our experiments, the analysis produced 28 warnings, only 5 of which corresponded neither to real errors nor to missed opportunities to add *init*. The results are summarized in Figure 19. We examined the warnings by hand and divided them into the following categories:

Cause of warning	Comment	Count
Bug	An actual bug	3
<i>init</i> opportunity	Function can be <i>init</i>	6
<code>init_module</code> API	Common <i>init</i> omission	14
Flow-insensitivity	False positives	3
Structure modeling	False positives	2
All warnings		28

Fig. 19. Warnings generated by CQUAL *init* analysis on Linux kernel 2.6.0-test6

Actual bugs. These errors could crash the kernel.

Init opportunities. These warnings indicate one or more functions that could safely be declared *init*, saving memory in the kernel.

Init module API. Every dynamically loaded module in the Linux kernel should have an `init_module` routine, and this routine can be declared *init*. Omitting the *init* is not a bug since the routine is only called during initialization, but it is a deviation from the specification. It is also a missed *init* opportunity.

Flow-insensitivity. Three of the warnings that were not errors or missed *init* opportunities and thus were false positives resulted from the flow-insensitivity of the analysis. For example, some device drivers maintain a flag indicating whether or not they have been initialized and only call *init* functions when the flag is off.

Structure modeling. Two warnings resulted from the lack of subtyping and polymorphism in our analysis of fields in C data structures.

We have submitted reports on all these warnings to kernel developers, and so far we have received confirmation of a few of them.

6. DISCUSSION

In this paper, we have presented a framework for adding type qualifiers to a language. In our framework, the type qualifiers form a partial order and introduce subtyping. They may also interact with the structure of terms, via well-formedness constraints, and they may be used to model effects. Our system also includes parametric polymorphism over qualifiers.

We have discussed three example sets of qualifiers in the paper: *const* and *non-const*, *tainted* and *untainted*, and *init* and *noninit*. In all of the examples we have shown, the qualifiers form the two-element lattice. Others have used CQUAL’s flow-insensitive type qualifiers to check additional properties. Johnson (one of the authors of this paper) and Wagner introduce *user* and *kernel* qualifiers to ensure that the Linux kernel does not trust pointers passed from the user into system calls [Johnson and Wagner 2004]. In this case, the qualifiers are in the discrete partial order (no qualifier is related to any other), and the qualifiers have well-formedness constraints. Broadwell et al. introduce a *sensitive* qualifier to mark data that should not be revealed in a core dump file, and use CQUAL’s type qualifier inference to determine where *sensitive* propagates [Broadwell et al. 2003]. Data that can be public is marked *unsensitive*, and the qualifiers form the two-element lattice. In earlier unpublished work, we introduced *YY* and *YYYY* qualifiers to mark strings that contain two- and four-digit years, respectively, and *NONYEAR* for strings that do not contain years. We then used CQUAL to look for Y2K bugs where strings

containing years were passed to the wrong date-related functions. In this case, the qualifiers are in the discrete partial order. We have also used other partial orders for flow-sensitive qualifiers (see below).

We have focused on flow-insensitive qualifiers in this paper because their theory is simple yet we have found them to be very effective in practice. As suggested by the previous examples, flow-insensitive type qualifiers can be used to check a number of different properties. Additionally, because our formulation of type qualifiers supports efficient inference (even with context-sensitivity), we can perform whole-program analysis on relatively large programs with few annotations. Another benefit of this approach is conceptual simplicity for the programmer. It is already the case in many languages that each variable has a fixed type throughout the program, i.e., assignment statements do not change the (static) types of variables. Adding type qualifiers that obey the same flow-insensitive discipline is a small (in some sense, the smallest) natural extension that still allows us to check interesting new properties.

Clearly, however, there are properties of imperative programs that flow-insensitivity cannot model. As one example, we cannot track state changes. For example, if after an assignment `*p = x` we want to change the qualifier on `*p`, we cannot do that with the system in this paper. For this we need *flow-sensitive* analysis. In prior work, we have described a flow-sensitive type qualifier system [Foster et al. 2002; Aiken et al. 2003] that is part of an earlier version of CQUAL. In our flow-sensitive analysis each memory location can have different qualifiers at different program points. For example, if `x` has type $ref(\kappa \text{ int})$ at one program point, it may have type $ref(\kappa' \text{ int})$ at another. Intuitively, our flow-sensitive system generates and solves qualifier constraints just as described in this paper, except with more qualifier variables (different ones for different program points). However, it is not quite that simple, and sound, scalable flow-sensitive analysis in the presence of heap aliasing is considerably more complicated than what we have presented here.

As a side note, we have studied two flow-sensitive qualifier analyses, both with different partial orders than described in this paper. In a deadlock detection experiment [Foster et al. 2002], we introduced flow-sensitive qualifiers *locked* and *unlocked* and then made sure that no code tries to acquire a mutex it has already locked. For this analysis we also introduced another qualifier \top to stand for the unknown lock state, with partial order $locked < \top$ and $unlocked < \top$. In another experiment, we used qualifiers to check that file operations were applied to files that had been previously opened in the correct mode (for reading, writing, etc) [Foster 2002]. For this experiment we added qualifiers such as *read*, *write*, and *close* in a more complicated partial order (see [Foster 2002]).

Flow-insensitivity is of course not the only limitation. A finite set of type qualifiers cannot describe all possible properties to check in a program. For example, we could not precisely model a generative system in which we would like each call to a function to return a type with a new constant qualifier on it. We also cannot easily model correlations among data. For example, suppose we have a structure `struct foo { int n; int a[]; }` where `n` is the length of array `a`. Then there is no natural way to represent that association precisely with a fixed set of qualifiers for all instances of `struct foo`. Finally, our restriction to subtyping also limits the expressiveness of qualifiers somewhat. For example, if we wanted to reason about

the exact values of integers using qualifiers, we could not do it very precisely. In particular, we can give $+$ the type $\forall \kappa_i^i[C]. \kappa_1 \text{ int} \times \kappa_2 \text{ int} \longrightarrow \kappa_3 \text{ int}$ for some set of constraints C , but C is restricted to subtyping constraints. Extensions to handle all of these kinds of properties is an interesting area for future work.

Despite the limitations of flow-insensitive type qualifiers, we believe they are still valuable and effective.

7. RELATED WORK

There are three main threads of related work: prior systems that use type qualifiers, other systems similar to CQUAL that find and prevent general errors in programs, and particular systems for finding format-string vulnerabilities.

7.1 Flow-Insensitive Type Qualifiers

Specific examples of flow-insensitive type qualifiers have been proposed to solve a number of problems. For example, ANSI C contains the type qualifier *const* [ANSI 1999], discussed in Section 5.1. The *const* qualifier was added to the standard in 1989 [ANSI 1989], inspired by C++’s *const*, which Stroustrup “invented” [Stroustrup 2005]. Binding-time analysis [Dussart et al. 1995] can be viewed as associating one of two qualifiers with expressions, either *static* for expressions that may be computed at compile time or *dynamic* for expressions not computed until run-time. The Titanium programming language [Yelick et al. 1998] uses qualifiers *local* and *global* to distinguish data located on the current processor from data that may be located at a remote node [Liblit and Aiken 2000]. Solberg [Solberg 1995] gives a framework for understanding a particular family of related analyses as type annotation (qualifier) systems. Pratikakis et al. [Pratikakis et al. 2004] give a system for inferring qualifiers for transparent Java futures. In contrast to these systems, our approach is an extensible, general framework for adding new, user-specified qualifiers that are expressible in our system.

Chin et al. [Chin et al. 2005] develop a semantic type qualifier system that allows qualifiers to be associated with language operations. Their system uses theorem proving to check that qualifier specifications are correct, and they can verify richer properties than subtyping constraints allow (for example, *pos* and *neg* qualifiers for integers). Their system does not currently include inference, unlike CQUAL.

Several related techniques have been proposed for using qualifier-like annotations to address security issues. A major topic of recent interest is *secure information flow* [Abadi et al. 1999; Denning 1976; Smith and Volpano 1998; Volpano and Smith 1997], which associates *high* and *low* security levels with expressions and tries to prevent high-security data from “leaking” to low-security outputs. Other examples of security-related annotation systems are lambda calculus with trust annotations [Ørbæk and Palsberg 1997] and Java security checking [Skalka and Smith 2000]. These systems include checks for implicit flows from conditional guards to the body of the conditional. Section 7.3 below contains a discussion of why CQUAL does not include this feature.

Type qualifiers, like any type system, can be seen as a form of abstract interpretation [Cousot and Cousot 1977]. Flow-insensitive type qualifiers can be viewed as a label flow system [Mossin 1996] in which we place constraints on where labels may flow. Type qualifiers can also be viewed as refinement types [Freeman

and Pfenning 1991], which have the same basic property: refinement types do not change the underlying type structure. The key difference between qualifiers and Freeman and Pfenning’s refinement types is that the latter is based on the theory of intersection types, which is significantly more complex than atomic subtyping. Mandelbaum et al. [Mandelbaum et al. 2003] have developed a type system that incorporates a logic of type refinements to allow reasoning about state, corresponding to flow-sensitivity. There is currently a limited implementation of their type system, but as of yet its scalability and effectiveness in practice are unknown.

7.2 Error Detection and Prevention Systems

Many systems have recently been proposed that allow programmers to check more properties of their programs. Vault [DeLine and Fähndrich 2001; Fähndrich and DeLine 2002] and Cyclone [Grossman et al. 2001; Grossman et al. 2002] are two safe variants of C that allow a programmer to enforce conditions on how resources are used in programs. These systems allow flow-sensitive tracking of resources, which is not modeled in our flow-insensitive framework (but see Foster et al. [Foster et al. 2002]). However, Vault and Cyclone also require per-function annotations, whereas CQUAL only requires a few annotations on the entire program and performs whole-program inference. Additionally, to use Vault and Cyclone the programmer must rewrite their program into the new language (which may vary from trivial to difficult), whereas CQUAL is designed to work with a legacy language, namely C. Because CQUAL operates on C, it cannot be fully sound, unlike these new languages.

Several systems based on dataflow analysis have been proposed to statically check properties of source code. These systems are flow-sensitive, in contrast to the flow-insensitive qualifiers described in this paper. One such system is Evans’s Splint [Evans 1996], which introduces a number of additional qualifier-like annotations to C as an aid to debugging memory usage errors. Evans found Splint to be valuable in practice [Evans 1996], and Splint has also been used to check for buffer overruns [Larochelle and Evans 2001]. The main difference between Splint and CQUAL is annotations. Splint’s analysis is intraprocedural, relying on programmers-supplied annotations at function calls, whereas CQUAL can perform whole-program inference.

Another such system is meta-level compilation [Engler et al. 2000; Hallem et al. 2002], in which the programmer specifies a flow-sensitive property as a finite state automaton. Meta-level compilation includes an interprocedural dataflow component [Hallem et al. 2002] but does not model general aliasing, unlike CQUAL. Meta-level compilation has been used to find many different kinds of bugs in programs, including tainting bugs [Yang et al. 2003]. The key difference between CQUAL’s approach and meta-level compilation is soundness. While CQUAL is not fully sound (e.g., due to arbitrary pointer arithmetic in C), CQUAL strives for soundness up to the limitations of C. In contrast, the goal of meta-level compilation is bug finding, and features like aliasing are ignored in order to limit false positives (which there are more of in CQUAL).

A third dataflow-based system is ESP [Das et al. 2002], an error detection tool based on sound dataflow analysis. ESP incorporates a conservative alias analysis to model pointers, and uses path-sensitive symbolic execution to model predicates.

ESP has been used to check the correctness of C stream library usage in `gcc` [Das et al. 2002]. ESP is designed to soundly detect all errors with a minimum of false positives; as such, the algorithms it uses for tracking state are quite sophisticated, and it is not easy for a programmer to predict in advance whether their program will check successfully. In contrast, CQUAL produces more warnings, but gives the programmer a relatively simple, predictable, type-based discipline to avoid errors.

The Extended Static Checking (ESC) system [Detlefs et al. 1998; Leino and Nelson 1998; Flanagan et al. 2002] is a theorem-proving based tool for finding errors in programs. Programmers add extensive annotations, including preconditions, postconditions, and loop invariants to their program, and ESC uses sophisticated theorem proving technology to verify the annotations. ESC includes a rich annotation language; the Houdini assistant [Flanagan and Leino 2001] can be used to reduce the burden of adding annotations. ESC provides significantly more sophisticated checking than CQUAL, but at the cost of scalability, both in terms of annotations and efficiency.

SLAM [Ball and Rajamani 2001; 2002] and BLAST [Henzinger et al. 2002] verify software using model checking techniques. Both tools can track program state very precisely and are by their nature flow- and path-sensitive. They use predicate abstraction followed by successive refinement to make analysis more tractable, and they have been used to check properties of device drivers. SLAM includes techniques for producing small counterexamples to explain error messages [Ball et al. 2003]. While the scalability of these tools is promising, the systems' worst-case complexity is much higher than CQUAL.

A number of techniques that are less easy to categorize have also been proposed. The AST toolkit provides a framework for posing user-specified queries on abstract syntax trees annotated with type information. The AST toolkit has been successfully used to uncover many bugs [Weise 2001]. The PREFIX tool [Bush et al. 2000], based on symbolic execution, is also highly effective at finding bugs in practice [Pincus 2002]. Both of these tools are unsound, and are designed to catch bugs rather than show the absence of errors.

A number of systems have been proposed to check that implementations of data structures are correct. Graph types [Klarlund and Schwartzbach 1993; Møller and Schwartzbach 2001] allow a programmer to specify the shape of a data structure and then check, with the addition of pre- and postconditions and loop invariants, that the shape is preserved by data structure operations. Shape analysis with three-valued logic [Sagiv et al. 1999] can also model data structure operations very precisely. Both of these techniques are designed to run on small inputs, and neither in its current form scales to large programs.

7.3 Format-String Vulnerabilities

Our approach to finding format-string vulnerabilities (described in a previous publication by us [Shankar et al. 2001]) is conceptually similar to Perl's taint mode [Wall et al. 2000], but with a key difference: unlike Perl, which tracks tainting dynamically, CQUAL checks tainting statically without ever running the program. Moreover, CQUAL's results are conservative over all possible runs of the program. This gives us a major advantage over dynamic approaches for finding security flaws. Often security bugs are in the least-tested portions of the code, and a malicious

adversary is actively looking for just such code to exploit. Using static analysis, we conceptually analyze all possible runs of the program, providing complete code coverage.

Several lexical techniques have been proposed for finding security vulnerabilities. Pscan [DeKok] searches the source code for calls to `printf`-like functions with a non-constant format string. Thus pscan cannot distinguish between safe calls when the format string is variable and unsafe calls. Lexical techniques have also been proposed to find other security vulnerabilities [Bishop and Dilger 1996; Viega et al. 2000]. The main advantage of lexical techniques is that they are extremely fast and can analyze non-preprocessed source files. However, because lexical tools have no knowledge of language semantics there are many errors they cannot find, such as those involving aliasing or function calls.

Another approach to eliminating format-string vulnerabilities is to add dynamic checks. The `libformat` library intercepts calls to `printf`-like functions and aborts when a format string contains `%n` and is in a writable address space [Robbins 2001]. A disadvantage to `libformat` is that, to be effective, it must be kept in synchronization with the C libraries. Another dynamic system is FormatGuard, which injects code to dynamically reject bad calls to `printf`-like functions [Cowan et al. 2001]. The main disadvantage of FormatGuard is that programs must be recompiled with FormatGuard to benefit. Another downside to both techniques is that neither protect against denial-of-service attacks.

It is important to realize that while CQUAL is successful at finding format-string vulnerabilities, it can never find all such bugs. One reason is the unsafe features of C, as discussed in Section 4.2. However, there is a more fundamental reason. Suppose the programmer performs a branch based on a tainted value. Then conceptually the program counter has become tainted, and any result that is control-dependent on the branch is suspect. Secure information flow systems, dual to our tainting analysis, use implicit flows (see above) to try to prevent just these kinds of security problems. We have found that modeling all possible information flows often leads to a very conservative analysis. For example, the `sendmail` program is a network daemon that waits for data from the network and then performs various tasks depending on the data. If taint propagates to the program counter, then all of `sendmail`'s computation must be tainted, which, while safe, is not a useful result.

8. CONCLUSION

We have presented a framework for flow-insensitive type qualifiers, a lightweight, specification-based technique for improving software quality. To use our system, the programmer supplies a set of qualifiers, a partial order among the qualifiers, and a source program with a few key type qualifier annotations. Constraint-based type qualifier inference takes as input the source program, determines the remaining qualifiers, and checks for consistency. Any inconsistent qualifiers indicate potential bugs in the program.

To show that our framework is useful in practice, we have described a tool CQUAL that implements our algorithms. An important component of our tool is a user interface for presenting analysis results. In this interface, the source code is colored according to the inferred qualifiers, and users can browse through the inference results to see how the results were inferred.

Finally, we have presented a number of experiments using CQUAL. We performed *const* inference for C programs, and we discovered that inference can add many more *consts* to existing programs, even ones whose authors make a significant effort to use *const*. We also used CQUAL to find a number of format-string bugs in popular programs; several of these bugs were unknown to us at the time. Finally, we used CQUAL to check whether functions and data marked as being collectible after kernel initialization time are annotated correctly or not, and we found several bugs as well as missed opportunities to collect functions.

In conclusion, we believe we have shown that type qualifiers are lightweight and easy to use because they are natural extensions of type systems and because of constraint visualization; that type qualifiers are practical, because both monomorphic and polymorphic inference scale to large programs; and that type qualifiers are useful for a number of realistic applications.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their thorough reviews and helpful suggestions.

REFERENCES

- ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas, 147–160.
- AIKEN, A., FOSTER, J. S., KODUMAL, J., AND TERAUCHI, T. 2003. Checking and Inferring Local Non-Aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, California, 129–140.
- ANSI 1989. *Rationale for American National Standard for Information Systems—Programming Language—C*. ANSI. Associated with ANSI standard X3.159-1989.
- ANSI 1999. *Programming languages – C*. ANSI. ISO/IEC 9899:1999.
- BAILLEUX, C. 2000. More security problems in bftpd-1.0.12. BugTraq Mailing List. <http://www.securityfocus.com/archive/1/149977>.
- BALL, T., NAIK, M., AND RAJAMANI, S. K. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, Louisiana, USA, 97–105.
- BALL, T. AND RAJAMANI, S. K. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *The 8th International SPIN Workshop on Model Checking of Software*. Number 2057 in Lecture Notes in Computer Science. 103–122.
- BALL, T. AND RAJAMANI, S. K. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, 1–3.
- BISHOP, M. AND DILGER, M. 1996. Checking for Race Conditions in File Accesses. *Computing Systems 2*, 2, 131–152.
- BROADWELL, P., HARREN, M., AND SASTRY, N. 2003. Scrash: A System for Generating Secure Crash Information. In *Proceedings of the 12th Usenix Security Symposium*. Washington, DC.
- BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. 2000. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience 30*, 7 (June), 775–802.
- CERT. 2001. CERT Advisory CA-2001-19 “Code Red” Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. <http://www.cert.org/advisories/CA-2001-19.html>.
- CHANDRA, S. AND REPS, T. W. 1999. Physical Type Checking for C. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toulouse, France, 66–75.
- ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year.

- CHIN, B., MARKSTRUM, S., AND MILLSTEIN, T. 2005. Semantic Type Qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, Illinois.
- COUSOT, P. AND COUSOT, R. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 238–252.
- COWAN, C., BARRINGER, M., BEATTIE, S., AND KROAH-HARTMAN, G. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*. Washington, D.C.
- DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 57–68.
- DAS, M., LIBLIT, B., FÄHNDRICH, M., AND REHOF, J. 2001. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis, Eighth International Symposium*, P. Cousot, Ed. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Paris, France.
- DAVEY, B. A. AND PRIESTLEY, H. A. 1990. *Introduction to Lattices and Order*. Cambridge University Press.
- DEKOK, A. PScan: A limited problem scanner for C source files. <http://www.striker.ottawa.on.ca/~aland/pscan>.
- DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 59–69.
- DENNING, D. E. 1976. A Lattice Model of Secure Information Flow. *Communications of the ACM* 19, 5 (May), 236–243.
- DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended Static Checking. Tech. Rep. 159, Compaq Systems Research Center. Dec.
- DUSSART, D., HENGLEIN, F., AND MOSSIN, C. 1995. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In *Static Analysis, Second International Symposium*, A. Mycroft, Ed. Number 983 in Lecture Notes in Computer Science. Springer-Verlag, Glasgow, Scotland, 118–135.
- EIFRIG, J., SMITH, S., AND TRIFONOV, V. 1995. Type Inference for Recursively Constrained Types and its Application to OOP. In *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier.
- ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. 2000. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth symposium on Operating System Design and Implementation*. San Diego, California.
- EVANS, D. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, Pennsylvania, 44–53.
- FÄHNDRICH, M. 1999. BANE: A Library for Scalable Constraint-Based Program Analysis. Ph.D. thesis, University of California, Berkeley.
- FÄHNDRICH, M. AND DELINE, R. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 13–24.
- FÄHNDRICH, M., FOSTER, J. S., SU, Z., AND AIKEN, A. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, Canada, 85–96.
- FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., WEIRICH, S., AND FELLEISEN, M. 1996. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, Pennsylvania, 23–32.
- FLANAGAN, C. AND LEINO, K. R. M. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of*
- ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TBD, Month Year.

- Formal Methods*, J. N. Oliverira and P. Zave, Eds. Number 2021 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 500–517.
- FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 234–245.
- FOSTER, J. S. 2002. Type Qualifiers: Lightweight Specifications to Improve Software Quality. Ph.D. thesis, University of California, Berkeley.
- FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. 1999. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia, 192–203.
- FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 1–12.
- FRASUNEK, P. 2001a. format string vulnerability in mars_nwe 0.99pl19. <http://online.securityfocus.com/archive/1/158959>.
- FRASUNEK, P. 2001b. ports/24733: mars_nwe remote format string vulnerability. http://groups.google.com/groups?q=mars_nwe+vulnerability&hl=en&lr=&ie=UTF-8&oe=UTF-8&selm=9566si%24gpv%241%40FreeBSD.csie.NCTU.edu.tw&rnum=1.
- FREEMAN, T. AND PFENNING, F. 1991. Refinement Types for ML. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Toronto, Ontario, Canada, 268–277.
- GATES, B. 2002. Trustworthy computing. Microsoft internal memo. Available at <http://www.theregister.co.uk/content/4/23715.html>.
- GAY, D. AND AIKEN, A. 2001. Language Support for Regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 70–80.
- GIFFORD, D. K., JOUVELOT, P., LUCASSEN, J. M., AND SHELDON, M. A. 1987. FX-87 Reference Manual. Tech. Rep. MIT/LCS/TR-407, MIT Laboratory for Computer Science. Sept.
- GREENFIELDBOYCE, D. AND FOSTER, J. S. 2004. Visualizing Type Qualifier Inference with Eclipse. In *Workshop on Eclipse Technology eXchange*. Vancouver, British Columbia, Canada.
- GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 282–293.
- GROSSMAN, D., MORRISETT, G., WANG, Y., JIM, T., HICKS, M., AND CHENEY, J. 2001. Cyclone User’s Manual. Tech. Rep. 2001-1855, Department of Computer Science, Cornell University. Nov.
- HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. 2002. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany, 69–82.
- HARRELSON, C. 2001. Program Analysis Mode. <http://www.cs.berkeley.edu/~chrisrtr/pam>.
- HEINTZE, N. AND TARDIEU, O. 2001. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 254–263.
- HENGLEIN, F. 1991. Efficient Type Inference for Higher-Order Binding-Time Analysis. In *FPCA ’91 Conference on Functional Programming Languages and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Cambridge, MA, 448–472.
- HENGLEIN, F. 1993. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems* 15, 2 (Apr.), 253–289.
- HENRION, M. 2000. muh IRC bouncer remote vulnerability. FreeBSD-SA-00:57. <http://www.securityfocus.com/advisories/2741>.
- HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy Abstraction. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, 58–70.
- ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year.

- HORWITZ, S., REPS, T., AND SAGIV, M. 1995. Demand Interprocedural Dataflow Analysis. In *Third Symposium on the Foundations of Software Engineering*. Washington, DC, 104–115.
- HUUSKONEN, J. 2000a. Possibility for formatchar errors in syslog call. https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=17349.
- HUUSKONEN, J. 2000b. Some possible format string errors. Linux Security Audit Project Mailing List. <http://www2.merton.ox.ac.uk/~security/security-audit-200009/0118.html>.
- HUUSKONEN, J. 2000c. syslog(prio, buf) in mars.nwe. Linux Security Audit Project Mailing List. <http://www2.merton.ox.ac.uk/~security/security-audit-200009/0136.html>.
- JOHNSON, R. AND WAGNER, D. 2004. Finding User/Kernel Bugs With Type Inference. In *Proceedings of the 13th Usenix Security Symposium*. San Diego, CA.
- KLARLUND, N. AND SCHWARTZBACH, M. I. 1993. Graph Types. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, 196–205.
- LAROCHELLE, D. AND EVANS, D. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*. Washington, D.C.
- LEINO, K. R. M. AND NELSON, G. 1998. An Extended Static Checker for Modula-3. In *Compiler Construction, 7th International Conference*, K. Koskimies, Ed. Lecture Notes in Computer Science, vol. 1383. Springer-Verlag, Lisbon, Portugal, 302–305.
- LIBLIT, B. AND AIKEN, A. 2000. Type Systems for Distributed Data Structures. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Boston, Massachusetts, 199–213.
- LUCASSEN, J. M. AND GIFFORD, D. K. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 47–57.
- MANDELBAUM, Y., WALKER, D., AND HARPER, R. 2003. An Effective Theory of Type Refinements. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*. Uppsala, Sweden, 213–225.
- MARS CLIMATE ORBITER MISHAP INVESTIGATION BOARD. 1999. Phase I Report. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MC0_report.pdf.
- MILNER, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17, 348–375.
- MITCHELL, J. C. 1991. Type inference with simple subtypes. *Journal of Functional Programming* 1, 3 (July), 245–285.
- MØLLER, A. AND SCHWARTZBACH, M. I. 2001. The Pointer Assertion Logic Engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 221–231.
- MOSSIN, C. 1996. Flow Analysis of Typed Higher-Order Programs. Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen.
- NECULA, G., MCPPEAK, S., AND WEIMER, W. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, 128–139.
- NEWSHAM, T. 2000. Format String Attacks. <http://online.securityfocus.com/guest/3342>.
- NIST. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3. <http://www.nist.gov/director/prog-ofc/report02-3.pdf>.
- ODERSKY, M., SULZMANN, M., AND WEHR, M. 1997. Type Inference with Constrained Types. In *Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages*, B. Pierce, Ed.
- ØRNBÆK, P. AND PALSBERG, J. 1997. Trust in the λ -calculus. *Journal of Functional Programming* 3, 2, 75–85.
- PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press.
- PINCUS, J. D. 2002. Personal communication.
- PITAC. 1999. President’s Information Technology Advisory Committee Report to the President. <http://www.ccic.gov/ac/report>.

- PRATIKAKIS, P., SPACCO, J., AND HICKS, M. 2004. Transparent Proxies for Java Futures. In *Proceedings of the nineteenth annual conference on Object-oriented programming systems, languages, and applications*. 206–223.
- PRATT, V. AND TIURYN, J. 1996. Satisfiability of Inequalities in a Poset. *Fundamenta Informaticae* 28, 1-2, 165–182.
- REHOF, J. AND FÄHNDRICH, M. 2001. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. London, United Kingdom, 54–66.
- REHOF, J. AND MOGENSEN, T. Æ. 1996. Tractable Constraints in Finite Semilattices. In *Static Analysis, Third International Symposium*, R. Cousot and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Aachen, Germany, 285–300.
- REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, 49–61.
- ROBBINS, T. J. 2001. libformat—protection against format string attacks. <http://www.wiretapped.net/~fyre/software/libformat.html>.
- SAGIV, M., REPS, T., AND WILHELM, R. 1999. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas, 105–118.
- SAVOLA, P. 2000. Very probable remote root vulnerability in cfengine. BugTraq Mailing List. <http://www.securityfocus.com/archive/1/136751>.
- SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*. Washington, D.C.
- SKALKA, C. AND SMITH, S. 2000. Static Enforcement of Security with Types. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*. Montreal, Canada, 34–45.
- SMITH, G. AND VOLPANO, D. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 355–364.
- SOLBERG, K. L. 1995. Annotated Type Systems for Program Analysis. Ph.D. thesis, Aarhus University, Denmark, Computer Science Department.
- STROUSTRUP, B. 2005. C++ Style and Technique FAQ. http://www.research.att.com/~bs/bs_faq2.html#constplacement.
- VIEGA, J., BLOCH, J., KOHNO, T., AND MCGRAW, G. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference*. <http://www.acsac.org>.
- VOLPANO, D. AND SMITH, G. 1997. A Type-Based Approach to Program Security. In *Theory and Practice of Software Development, 7th International Joint Conference*, M. Bidoit and M. Dauchet, Eds. Lecture Notes in Computer Science, vol. 1214. Springer-Verlag, Lille, France, 607–621.
- WALL, L., CHRISTIANSEN, T., AND ORWANT, J. 2000. *Programming Perl*, 3rd Edition ed. O’Reilly & Associates.
- WEISE, D. 2001. Personal communication.
- WRIGHT, A. K. 1995. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation* 8, Vol. 4. 343–356.
- WRIGHT, A. K. AND FELLEISEN, M. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1, 38–94.
- YANG, J., KREMENEK, T., XIE, Y., AND ENGLER, D. 2003. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM Conference on Computer and Communication Security*. Washington, D.C., USA, 321–334.
- ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TDB, Month Year.

- YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. 1998. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*.
- YONG, S. H., HORWITZ, S., AND REPS, T. 1999. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia, 91–103.
- ZHANG, X., EDWARDS, A., AND JAEGER, T. 2002. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th Usenix Security Symposium*. San Francisco, CA.

A. CQUAL SURFACE SYNTAX AND CONFIGURATION FILES

CQUAL is implemented using the RC region compiler’s C front end [Gay and Aiken 2001]. In the surface syntax we write all constant qualifiers except built-in ones like *const* with a dollar sign, so that the lexer can unambiguously tokenize them. (While some C compilers do allow dollar signs in identifiers, their use is infrequent, so CQUAL reserves them for qualifiers.) We allow CQUAL qualifiers to appear in the same positions where *const* can occur. We also extend the syntax of C to allow qualifiers in two additional places: on the ... of varargs functions (as mentioned in Section 4.1) and after the right parenthesis of a function declaration’s parameter list. The latter places the qualifier on the function arrow itself, i.e., it is an effect. For example,

```
void f(void) q;
```

declares a function of type q ($void \rightarrow void$) (qualifiers on *void* omitted for clarity).

We also allow functions to be declared with polymorphic type signatures, as described in Section 4.1. Generalized qualifier variables as written as $\$_{n_1 \dots n_k}$ where $n_1 \dots n_k$ are numbers representing the set $\{n_1, \dots, n_k\}$. Using s_i as an abbreviation for such a set of numbers, a declaration of the form

```
$_{s_0} typ_0 f($_{s_1} typ_1, ..., $_{s_k} typ_k);
```

declares a function **f** of the type

$$\forall \vec{\kappa}_i[C]. \kappa_1 \text{ typ}_1 \times \dots \times \kappa_k \text{ typ}_k \rightarrow \kappa_0 \text{ typ}_0$$

where $\kappa_i \leq \kappa_j \in C$ if $s_i \subseteq s_j$. (The qualifier on the function arrow has been omitted for clarity.)

As mentioned in Section 5.3, CQUAL allows the user to give declarations of the qualified types of built-in operations such as the dereferencing operator *****. These are given in the form of special function declarations like

```
$$a _op_deref($$a *$_1 x) $_1;
```

This declaration states that the dereference operator, which takes a pointer to any type **\$\$a** (this is a type variable that ranges over standard types) and returns an object of that type, has as its effect the qualifier **\$_1**.

Qualifier partial orders are specified using a special configuration file. Figure 20 gives the partial order configuration file for the qualifiers discussed in Section 5. The complete grammar for partial order configuration files is shown in Figure 21.

```

partial order {
  const [level = ref, sign = pos]
  $nonconst [level = ref, sign = neg]

  $nonconst < const
}

partial order {
  $untainted [level = value, color = "pam-color-untainted", sign = neg]
  $tainted [level = value, color = "pam-color-tainted", sign = pos]

  $untainted < $tainted
}

partial order [effect] {
  $init [level = ref, color = "pam-color-tainted", sign = pos,
        fieldflow = down, fieldptrflow = all]

  $noninit [level = ref, color = "pam-color-untainted", sign = neg,
           fieldflow = down, fieldptrflow = all]

  $noninit < $init
}

```

Fig. 20. Example Partial Order Configuration File

```

po-defn ::= partial order [ po-opt* ]? { po-entry* }
po-opt  ::= flow-insensitive
           | flow-sensitive
           | nonprop
           | effect
           | casts-preserve
po-entry ::= qual-name [ qual-opt* ]?
           | qual-name < qual-name
qual-opt ::= color = "color-name"
           | level = ref | level = value
           | sign = pos | sign = neg | sign = eq
           | ptrflow = dir
           | fieldflow = dir
           | fieldptrflow = dir
dir     ::= up | down | all

```

Fig. 21. Partial Order Configuration File Grammar

In this grammar, x^* means zero or more occurrences of x , and $[x]^?$ means either zero or one occurrence of $[x]$.

Each partial order can be declared to contain either flow-insensitive qualifiers, flow-sensitive qualifiers [Foster et al. 2002], or non-propagating qualifiers (which should not be inferred). Additionally, partial orders may also be marked as containing effect qualifiers (Section 5.3) and/or qualifiers that are preserved through casts (Section 4.2). The canonical example of a non-propagating qualifier is *restrict*, which has a special meaning in C [Aiken et al. 2003]. For each partial order the user lists the qualifiers and their options. The **sign** option specifies the variance

of a qualifier: positive (**pos**), negative (**neg**), or non-variant (**eq**) (Section 4.1). The **level** options are explained in Section 4.1, and the **color** option is used in the visualization described in Section 4.3. Each qualifier also has well-formedness conditions (Section 3.3), specified as flowing between pointer and pointed-to data (**ptrflow**), between aggregate and contents (**fieldflow**), and between aggregate and pointer-valued contents (**fieldptrflow**). Finally, the partial order is specified by declarations $a < b$ for each pair of qualifiers so related in the partial order. We compute the reflexive transitive closure of the specified relations to yield the final partial order.

Received Month Year