Composing Polymorphic Information Flow Systems with Reference Immutability

Ana Milanova
Rensselaer Polytechnic Institute
Troy, NY, USA
milanova@cs.rpi.edu

Wei Huang
Rensselaer Polytechnic Institute
Troy, NY, USA
huangw5@cs.rpi.edu

ABSTRACT

Information flow type systems, such as EnerJ (a type system for energy efficiency), and integrity and confidentiality, are unsound if subtyping for references is allowed because of the presence of mutable references. The standard approach is to disallow subtyping for references, or in other words, replace subtyping constraints with equality constraints. Unfortunately, this often leads to imprecision, causing the type system to reject valid programs.

We observe that subtyping is safe when the left-hand-side of the assignment is immutable. Therefore, we compose information flow systems with reference immutability, which allows for limited subtyping for references. We infer types with the standard approach (i.e., no subtyping for references), and with the composition approach on 13 Java web applications. The composition approach achieves at least 20% precision improvement over the standard approach.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, Theory

Keywords

information flow, reference immutability, inference

1. INTRODUCTION

We consider a class of type systems, which we call *polymorphic information flow systems*. The general structure of these systems is as follows. The universe of type qualifiers is

$$U = \{\mathsf{neg}, \mathsf{poly}, \mathsf{pos}\}$$

with subtyping hierarchy

$$\mathsf{neg} <: \mathsf{poly} <: \mathsf{pos}$$

Here neg is the "negative" qualifier and pos is the "positive" qualifier. The goal of the type system is to ensure that there is no flow from a "positive" variable x to a "negative" variable y. poly is a polymorphic qualifier, which is interpreted as pos in some contexts, and as neg in other contexts.

The best examples of information flow systems are confidentiality and integrity systems. A confidentiality system instantiates neg to public and pos to secret. The goal of the system is to ensure that there is no flow from secret *sources* to public *sinks*. Intuitively, it is safe to assign a public variable to a secret one, but it is not safe to assign a secret variable to a public one; hence the direction of the subtyping relation: public <: secret. Note that this is the desired subtyping, but unfortunately, as we show in Section 3.3, allowing such subtyping for references is not always safe. The standard solution has been to disallow subtyping for references [17, 16], which unfortunately leads to loss of precision.

This paper proposes a principled approach for composing information flow systems with reference immutability [23, 12]. Our approach allows for polymorphism in both the information flow system and the reference immutability system. The composition allows for limited subtyping, but even this limited subtyping improves precision significantly.

The rest of the paper is organized as follows. Section 2 describes language syntax and other preliminaries. Section 3 describes the information flow system and Section 4 describes reference immutability, which we contend, is a special case of an information flow system. Section 5 describes the composition of information flow systems with reference immutability. Section 6 and Section 7 describe our implementation and empirical results. Section 8 briefly discusses related work, and Section 9 concludes with an outline of future work.

2. PRELIMINARIES

This section describes our language syntax, the notion of viewpoint adaptation and the generalized typing rules.

2.1 Syntax

We restrict our formal attention to a core calculus in the style of Vaziri et al. [24] whose syntax appears in Figure 1. The language models Java with a syntax in a "named form", where the results of field accesses, method calls, and instantiations are immediately stored in a variable. Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter. Features not strictly

$$\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} & \textit{class} \\
fd & ::= t\ \mathsf{f} & \textit{field} \\
md & ::= t\ \mathsf{m}(t\ \mathsf{this},\ t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}}\ \mathsf{s};\ \mathsf{return\ y}\ \} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s};\mathsf{s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} & \textit{statement} \\
 & \mid \mathsf{x} = \mathsf{y.f} \mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.m(z)} & \\
t & ::= q\ \mathsf{C} & \textit{qualified type}
\end{array}$$

Figure 1: Syntax. C and D are class names, f is a field name, m is a method name, and x, y, z are names of local variables, formal parameters, or parameter this, and q is type qualifier. As in the code examples, this is explicit.

necessary are omitted from the formalism, but they are handled correctly in the implementation. We write $\overline{t\ \mathsf{y}}$ for a sequence of local variable declarations.

A type t has two orthogonal components: type qualifier q and Java class type C. A pluggable type system is *orthogonal* to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

2.2 Viewpoint Adaptation and Typing Rules

Viewpoint adaptation is a concept from Universe Types [5], which applies to other ownership type systems as well [4, 24]. For example, the type of x.f is not just the declared type of field f — it is the type of f adapted from the viewpoint of x. In Universe Types, rep x denotes that the current this object is the owner of the object i referenced by x. If field f has type peer, this means that the object i and the object j referenced by field f have the same owner. Thus, the type of x.f, or the type of f adapted from the viewpoint of x, is rep — the object j's owner is the current this object as well.

Ownership type systems make use of a viewpoint adaptation operation. This viewpoint adaptation operation is performed at field accesses and method calls. It is written $q \rhd q'$, which denotes that type q' is adapted from the viewpoint of type q, to the viewpoint of the current object this. Traditional viewpoint adaptation always adapts from the viewpoint of the *receiver* at the corresponding field access or method call.

The typing rules for all systems in this paper fit into the framework for ownership-like types we developed in previous work [11]. The rules are shown in Figure 2. Explicit assignments ((TNEW), (TASSIGN), (TREAD), (TWRITE)) create the expected subtyping constraints from the right-hand-side of the assignment to the left-hand-side. So do implicit assignments at (TCALL): there are subtyping constraints that link actual arguments to formal parameters, and return value to the left-hand-side of the call assignment. Rules for field access and method calls make use of viewpoint adaptation.

We use a generalization of traditional viewpoint adaptation. Specifically, we allow for adaptation from *different viewpoints*, not only from the viewpoint of the receiver. Essentially, viewpoint adaptation encodes context sensitivity directly in the typing rules. Varying the viewpoint adaptation operation and/or the choice of viewpoint adapter at (TCALL), allows for encoding of *different kinds* of context sensitivity (e.g., CFL-reachability, object sensitivity, etc.). Returning to Figure 2, rule (TCALL) is parameterized by context of adaptation a, where a instantiates to some combination of the types at the method call (i.e., $q_\mathsf{x}, q_\mathsf{y}$ and q_z) and the types at the method definition ($q_\mathsf{ret}, q_\mathsf{this}$ and q). For now on, we refer to q in $q \rhd q'$ as the *context of adaptation*, or simply as the *context*.

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad q <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}()}\ \text{(TNEW)}$$

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ \text{(TASSIGN)}$$

$$\frac{\Gamma(\mathsf{x}) = q_\mathsf{x} \quad \textit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{y}) = q_\mathsf{y} \quad q_\mathsf{y} <: q_\mathsf{x} \rhd q_\mathsf{f}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ \text{(TWRITE)}$$

$$\frac{\Gamma(\mathsf{y}) = q_\mathsf{y} \quad \textit{typeof}(\mathsf{f}) = q_\mathsf{f} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ \text{(TREAD)}$$

$$\frac{\begin{array}{c}\Gamma(\mathsf{y}) = q_\mathsf{y} \quad \textit{typeof}(\mathsf{m}) = q_\mathsf{this}, q \to q_\mathsf{ret} \quad \Gamma(\mathsf{x}) = q_\mathsf{x} \quad \Gamma(\mathsf{z}) = q_\mathsf{z} \\ q_\mathsf{y} <: a \rhd q_\mathsf{this} \quad q_\mathsf{z} <: a \rhd q \quad a \rhd q_\mathsf{ret} <: q_\mathsf{x}\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ \text{(TCALL)}$$

Figure 2: Typing rules. Function *typeof* retrieves the types of fields and methods. Γ is a type environment that maps references to qualifiers. a is the context of adaptation.

3. POLYMORPHIC INFORMATION FLOW SYSTEM N

The type qualifiers and subtyping hierarchy in N is

$$\mathsf{neg} <: \mathsf{poly} <: \mathsf{pos}$$

as described in Section 1. We now elaborate on this system.

Instance fields in N are interpreted in the context of the receiver object. Field types are restricted to poly or pos. Thus, a field f that is typed pos, is guaranteed to be pos in all x.f, while a field that is poly is interpreted depending on the type of x. We disallow neg qualifiers on fields. If they were allowed, the meaning of a pos reference with a neg field becomes difficult to interpret. Essentially, this would amount to excluding the neg field from the state of the object, analogously to the way Javari [23] excluded assignable fields from the state of the object. This complicates the semantics of the system and for this reason, we have chosen to disallow neg fields in the current version of our system. Other choices are possible, and we plan to explore them in future work.

Viewpoint adaptation $q \rhd q'$ is defined as follows:

$$\begin{array}{lcccl}
\_ & \rhd & \mathsf{pos} & = & \mathsf{pos} \\
\_ & \rhd & \mathsf{neg} & = & \mathsf{neg} \\
q & \rhd & \mathsf{poly} & = & q
\end{array}$$

Therefore, pos and neg qualifiers remain the same, regardless of the context of adaptation, while poly qualifiers assume the type of q. Local poly variables are adapted in the context of invocation.

We should mention that our information flow systems only handle *explicit flows* (also known as data dependences). This is evident from the syntax and the rules in Figure 2. *Implicit flows* (known as control dependences) are important as well, but unfortunately, it has been difficult to incorporate them into practical information flow analysis tools. Taint analysis tools targeting large applications do not detect implicit flows [13, 21, 19, 22, 7]. Similarly, commercial tools such as IBM's AppScan and HP's Fortify, do not detect implicit flows (see [7] for a detailed evaluation). Our analysis targets large applications as well, and for this reason, we choose to exclude implicit flows (we believe that they can be handled by extending the type system, in a manner similar to the way we extended the reference immutability system ReIm to handle method purity [12]).

To make discussion concrete, we examine two polymorphic information flow systems, EnerJ and SFlow. EnerJ is a type system for energy efficiency [16]. SFlow is a confidentiality (taint) system that prevents flow from secret variables to public variables.

3.1 EnerJ

EnerJ allows programmers to designate certain variables as *precise* and other variables as *approximate*. Operations on approximate variables are more energy efficient than operations on precise variables. EnerJ allows for polymorphic variables. Essentially, certain methods have an approximate (and more energy efficient) version, and a precise (and less energy-efficient) version. Depending on invocation context, which in EnerJ is the type of the receiver, the call invokes the approximate or the precise version of the method.

EnerJ's qualifiers are:

$$\mathsf{precise} <: \mathsf{poly} <: \mathsf{approx}$$

(although in [16], poly is called context). Thus, EnerJ allows assignment from a precise variable to an approximate one, but disallows assignment from an approximate variable to a precise one. As already mentioned, EnerJ selects the *receiver* as the context of adaptation at (TCALL). That is, a is q_{y}. EnerJ guarantees that no approx variable "influences" the value of a precise variable.

3.2 SFlow

SFlow[1] has three qualifiers:

- A public reference x, and its transitively reachable state, may flow to an untrusted party (i.e., to a *sink*).
- A secret reference x, and its transitively reachable state, cannot flow to a sink.
- A poly reference x is polymorphic, i.e., it can be instantiated to public in some invocation contexts of x's enclosing method, and to secret in other invocation contexts.

The subtyping hierarchy is:

$$\mathsf{public} <: \mathsf{poly} <: \mathsf{secret}$$

Thus, we choose secret as the positive qualifier and public as the negative qualifier. Just as with EnerJ, SFlow selects the receiver as the context of adaptation at (TCALL): a is q_{y}. SFlow guarantees that there is no interference from secret variables to public variables.

Note that our choice of secret as the positive qualifier and public as the negative qualifier is arbitrary. We could have chosen public as the positive qualifier and secret as the negative one. Our choice corresponds to a confidentiality system, which prevents flow from secret variables to public variables. The opposite choice reflects the dual integrity (also known as "taint") system, which prevents flow from low-integrity (i.e., public) variables to high-integrity ones.

[1]Systems similar to SFlow were described in [17, 1]

To illustrate the importance of poly, consider the following excerpt from Stanford's securibench-micro from http://suif.stanford.edu/~livshits/work/securibench-micro/

```
protected void doGet(secret HttpServletRequest req,
                     public HttpServletResponse resp) {
  secret String s1 = req.getParameter("name");
  public String s2 = "abc";
  secret String s3 = s1.toUpperCase();
  public String s4 = s2.toUpperCase();

  public PrintWriter writer = resp.getWriter();
  writer.println(s3); /* BAD */
  writer.println(s4); /* OK */
}
```

The return value of HttpServletRequest.getParameter is a *source* and getParameter is typed as follows (we make parameter this explicit):

```
secret String getParameter(poly HttpServletRequest this,
                           poly String name)
```

The parameter of PrintWriter.println is a *sink* and thus, println is typed as follows:

```
void println(poly PrintWriter this, public String name)
```

SFlow must prevent flow from the source to the sink.
String.toUpperCase is polymorphic:

```
poly String toUpperCase(poly String this)
```

Recall that the context of adaptation is the receiver. Thus, at call s3 = s1.toUpperCase(), the rules in Figure 2 entail constraints:

$$q_{\mathsf{s1}} <: q_{\mathsf{s1}} \triangleright \mathsf{poly} \qquad q_{\mathsf{s1}} \triangleright \mathsf{poly} <: q_{\mathsf{s3}}$$

q_{s1} is secret, thus poly instantiates to secret in the context of s1. Since s3 is secret, the constraints hold. On the other hand, at call s4 = s2.toUpperCase() we have constraints:

$$q_{\mathsf{s2}} <: q_{\mathsf{s2}} \triangleright \mathsf{poly} \qquad q_{\mathsf{s2}} \triangleright \mathsf{poly} <: q_{\mathsf{s4}}$$

q_{s2} is public and therefore poly instantiates to public. Since s4 is public, the constraints hold.

Call writer.println(s3) does not type-check because

$$q_{\mathsf{s3}} <: q_{\mathsf{writer}} \triangleright \mathsf{public} \quad \equiv \quad \mathsf{secret} <: \mathsf{public}$$

does not hold. Call writer.println(s4) type-checks.

3.3 Issues with System N

So far, we overlooked the thorny issue of subtyping in the presence of mutable references. If we allowed subtyping for references, then the type system would permit flow from a pos variable to a neg variable as in the following example with SFlow:

```
secret X sx;        px = new public X;
secret A sa;        sx = px; // allowed by subtying
public X px;        sx.f = sa;
public A pa;        pa = px.f;
```

With subtpying for references, this program type-checks, but it allows flow from secret sa to public pa. This in fact, is the well-known issue with Java's covariant arrays [15].

The standard solution is to disallow subtyping for references [17, 16]. For example, EnerJ [16] defines two sets of qualifiers: precise <: poly <: approx for simple types, and Precise, Poly, Approx for references. While subtyping is allowed for simple types, it is disallowed for references.

Unfortunately, disallowing subtyping for references leads to imprecision, i.e., the type system rejects perfectly valid programs. It amounts to using equality constraints as opposed to subtyping constraints, and thus, propagating neg qualifiers bi-directionally, resulting in often unnecessary propagation. Disallowing subtyping is in some sense analogous to using unification constraints as opposed to subset constraints in points-to analysis. It is well-known that Steensgaard's points-to analysis [20], which uses unification (i.e., equality) constraints, is substantially less precise than Andersen's points-to analysis [2], which uses subset constraints.

To illustrate the problem, consider Long.valueOf from the standard JDK (slightly modified):

```
static Long valueOf(long l) {
   final int offset = 128;
   Long result;
   if (l >= −128 && l <= 127) {
      int t = (int) l+offset;
      result = LongCache.cache[t];
      return result;
   }
   result = new Long(l);
   return result;
}
```

The desired typing of Long.valueOf is poly → poly:

```
poly Long valueOf(poly long l)
```

or in other words, if the argument of a call to Long.valueOf is secret, then the left-hand-side of the call assignment should be secret, and vice versa, if the left-hand-side is public, then the argument should be public.[2] However, LongCache.cache is a global array, and as such, the array and its elements must be typed public (clearly, in some invocation contexts of Long.valueOf, the elements of LongCache will flow to public outputs). If subtyping for references were disallowed, the public LongCache.cache[t] would force result, as well as formal parameter l to be public. The only possible typing would be:

```
public Long valueOf(public long l)
```

Therefore, code such as

```
secret long li = ...
... lhs = Long.valueOf(li);
```

would be untypable, even if lhs does not flow to a sink.

The key observation behind our proposed approach is that when a reference x is *immutable*, then subtyping in the N qualifiers in assignments x = ... , is safe. Section 4 describes reference immutability, which ensures that references are immutable. Interestingly, reference immutability presents a special case of information flow. Section 5 presents a system which composes N with reference immutability, which allows for limited subtyping in N's qualifiers.

4. REFERENCE IMMUTABILITY

Reference immutability enforces the property that the state of an object, including its transitively reachable state, *cannot be mutated through an immutable reference*. Reference immutability is different from object immutability in that the former enforces constraints on references while the latter focuses on the object instance. For instance, in the following code, we cannot mutate the Date object by using the immutable reference rd, but we can mutate the same Date object through the mutable reference md:

[2]The context at static calls is the left-hand-side of the call.

```
Date md = new Date(); // mutable by default
readonly Date rd = md; // an immutable reference
md.setHours(1);         // OK, md is mutable
rd.setHours(1);         // error, rd is immutable
```

Reference immutability has been studied extensively in the literature [23, 12]. In previous work we presented ReIm, a type system for reference immutability [12]. ReIm works with qualifiers mutable, polyread, and readonly:

- A mutable reference can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages.
- A readonly reference x cannot be used to mutate the referenced object nor anything it references. For example, all of the following are forbidden:
  - x.f = z
  - x.setField(z) where setField sets a field of its receiver
  - y = id(x); y.f = z where id is a function that returns its argument
  - y = x.f; y.g = z
- A polyread reference x cannot be used to mutate the object it references, nor anything it references. Just as the poly qualifier in N, the polyread qualifier enables polymorphism. polyread can be instantiated to mutable in some invocation contexts and to readonly in other invocation contexts. For example,
  - x.f = y is not allowed, but
  - z = id(y); z.f = 0, where id is polyread X id(polyread X x) { return x; }, and z and y are mutable, is allowed. In this case, polyread is instantiated to mutable.

ReIm's subtyping hierarchy is as follows:

$$\text{mutable} <: \text{polyread} <: \text{readonly}$$

Viewpoint adaptation $q \rhd q'$ in ReIm is analogous to N:

$$\begin{array}{rcl} \_ \rhd \text{readonly} & = & \text{readonly} \\ \_ \rhd \text{mutable} & = & \text{mutable} \\ q \rhd \text{poyread} & = & q \end{array}$$

Just as in N, fields are typed polyread or readonly.

The rules in Figure 2 apply to ReIm with the following two extensions. First, at rule (TWRITE), ReIm forces q_{x} to be mutable. Second, the context of adaptation at (TCALL) is the *left-hand-side of the call assignment*, i.e., a is q_{x}. At calls without a left-hand-side, the context is readonly.

As an example, let us consider class DateCell.

```
class DateCell {
  polyread Date date;
  polyread Date getDate(polyread DateCell this) {
     return this.date;
  }
  void m1(mutable DateCell this) {
     mutable Date md = this.getDate();
     md.setHours(1);           // md is mutatated
  }
  void m2(readonly DateCell this) {
     readonly Date rd = this.getDate();
     int hour = rd.getHours();
  }
}
```

Field date is polyread, which means that in some contexts, date is interpreted as mutable and in other contexts, date is interpeted as readonly. Parameter this and return value ret of getDate are polyread, meaning that they are interpreted differently in different invocation contexts.

Consider the call md = this.getDate() in m1. The context of adaptation is the left-hand-side of the call assignment, that is, q_{md}. The typing rules in Figure 2 entail constraints:

$$q_{\mathsf{this}_{\mathsf{m1}}} <: q_{\mathsf{md}} \rhd q_{\mathsf{this}} \quad q_{\mathsf{md}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{md}}$$

Since q_{this} instantiates to mutable in this context, $q_{\mathsf{this}_{\mathsf{m1}}}$ must be mutable as well (and it is).

On the other hand, at the call to rd = this.getDate() in m2, the context of adaptation is q_{rd}. There are the following constraints:

$$q_{\mathsf{this}_{\mathsf{m2}}} <: q_{\mathsf{rd}} \rhd q_{\mathsf{this}} \quad q_{\mathsf{rd}} \rhd q_{\mathsf{ret}} <: q_{\mathsf{rd}}$$

q_{this} now instantiates to readonly, which allows $q_{\mathsf{this}_{\mathsf{m2}}}$ to be readonly.

The goal of reference immutability is to ensure that there is no flow from positive readonly references to negative mutable references. Thus, in its essence, reference immutability is an information flow system in our sense, albeit a special case of an information flow system, as we argue below.

Firstly, in information flow systems such as EnerJ and SFlow, the type qualifier of a reference x reflects on the *content* of the object x refers to, not on the reference or object itself. For example, in EnerJ, an approx reference x indicates that the simple-type fields, e.g., `int` and `char` fields, of the object referenced by x are approximate, not that the address stored in x or the object are somehow approximate. For reference immutability, qualifiers mutable, polyread and readonly reflect the mutability of the reference itself, *in addition to* the mutability of its content (i.e., fields) of reference type. For example, mutable x may mean that x is updated directly at some x.f = y, or that a field obtained through x is mutated, even though x itself is not updated directly (e.g., y = x.f; y.g = 0).

Furthermore, unlike in EnerJ and SFlow, subtyping in ReIm is always safe. Consider b = y.f, where b is mutable. The mutable b forces y to be mutable (as we discussed above, mutable reflects on the reference and its *content*). The mutabilty of b also forces f to be polyread. At x.f = a, the polyread field adapts to mutable, making it impossible to transmit a readonly a to the mutable b.

5. COMPOSITION SYSTEM $N \times R$

The composition system $N \times R$ consists of two orthogonal components. N is an information flow system (e.g., EnerJ, SFlow), and R is a reference immutability system. Our discussion instantiates R to ReIm, but in general, R can be instantiated to other reference immutability systems.

The universe of type qualifiers is the cartesian product of N and ReIm: $U_{N\times\mathsf{ReIm}} = \{\langle\mathsf{pos\ readonly}\rangle, \langle\mathsf{pos\ polyread}\rangle, \langle\mathsf{pos\ mutable}\rangle, \langle\mathsf{poly\ readonly}\rangle, \langle\mathsf{poly\ polyread}\rangle, \langle\mathsf{poly\ mutable}\rangle, \langle\mathsf{neg\ readonly}\rangle, \langle\mathsf{neg\ polyread}\rangle, \langle\mathsf{neg\ mutable}\rangle\}$.

The viewpoint adaptation operation in $N \times$ ReIm is as expected: it amounts to component-wise application of the respective viewpoint adaptation operations of N and ReIm. We use superscript n to denote the N component of a $N \times$ ReIm type qualifier, and r to denote the ReIm component. For example, in $q_{\mathsf{x}} = \langle q_{\mathsf{x}}^n\ q_{\mathsf{x}}^r\rangle$, q_{x}^n denotes the N component of the type of x, and q_{x}^r denotes the ReIm component. $q \rhd q_1$

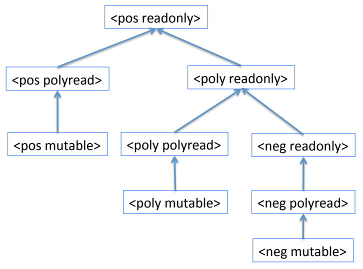

Figure 3: $N \times$ ReIm subtyping hierarchy. Arrows link subtypes to supertypes.

is as follows (we abuse notation a bit by overloading $\rhd$ to act on tuples as well as on individual components):

$$\langle q^n\ q^r\rangle \rhd \langle q_1^n\ q_1^r\rangle = \langle q^n \rhd q_1^n\ \ q^r \rhd q_1^r\rangle$$

The subtyping hierarchy is shown in Figure 3. <: is the relation that induces the maximal number of pairs $q <: q_1$ such that the following three conditions hold for each $q <: q_1$:

1. $q <: q_1 \Rightarrow q^n <: q_1^n$ and $q^r <: q_1^r$
2. $q <: \langle q_1^n\ \mathsf{mutable}\rangle \Rightarrow q = \langle q_1^n\ \mathsf{mutable}\rangle$. In other words, if the left-hand-side of an assignment is mutable, not only that the right-hand-side becomes mutable, but the N components of q and q_1 must be equal. Thus, subtyping of mutable references is disallowed, as expected. If it were allowed, we would have encountered the unsoundness issues described in Section 3.3.
3. Viewpoint adaptation is *order preserving*: that is, for every $q, q_1, q_2 \in U_{N\times\mathsf{ReIm}}$, $q_1 <: q_2 \Rightarrow q \rhd q_1 <: q \rhd q_2$. Intuitively, if q_1 and q_2, where $q_1 <: q_2$, qualify respectively local variables x and y in some method m, adapting m (and thus q_1 and q_2) from some context q, should preserve subtyping for x and y. Order preservation is necessary to ensure soundness. As an example, order preservation forbids that ⟨poly polyread⟩ is a subtype of ⟨pos polyread⟩ (note that there is no link in Figure 3), even though subtyping holds for the individual components. However, if we adapted from ⟨neg mutable⟩, ⟨poly polyread⟩ would be ⟨neg mutable⟩ and ⟨pos polyread⟩ would be ⟨pos mutable⟩. Clearly, ⟨neg mutable⟩ cannot be a subtype of ⟨pos mutable⟩ for the reasons discussed in 2.

The important benefit of the composition of N and R is that it allows subtyping in the N-component whenever the left-hand-side of the assignment is a readonly reference. As a result, it achieves better precision.

Returning to the Long.valueOf example from the previous section, it can now be typed poly → poly in SFlow:

```
static ⟨poly readonly⟩ Long valueOf(⟨poly readonly⟩ long l) {
   final ⟨poly readonly⟩ int offset = 128;
   Long ⟨poly readonly⟩ result;
   if (l >= −128 && l <= 127) {
      ⟨poly readonly⟩ int t = (int) l+offset;
      result = LongCache.cache[t];
      return result;
   }
   result = new Long(l);
   return result;
}
```

Since result is readonly in ReIm, we can subtype in the SFlow component at result = LongCache.cache[t].

6. TYPE INFERENCE

Type inference is structured in the framework we developed in [11]. The key idea is to compute a *set-based* solution S which maps variables to *sets* of type qualifiers. The set-based solver initializes every programmer-annotated variable to the singleton set that contains the programmer-provided qualifier. It initializes unannotated variables to the maximal set of qualifiers (e.g. the set of $\{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$ in the case of N).

There is a function f_s for each statement s. Each f_s takes as input the current mapping S and outputs an updated mapping S'. f_s removes infeasible qualifiers from the set of each reference that participates in s according to the typing rule for s in Figure 2.

For example, consider statement $\mathsf{x} = \mathsf{y.f}$, which corresponds to the typing rule (TREAD) defined in Figure 2. Suppose that before the application of the transfer function, we have $S(\mathsf{x}) = \{\mathsf{poly}\}$, $S(\mathsf{y}) = \{\mathsf{pos}, \mathsf{poly}, \mathsf{neg}\}$, and $S(\mathsf{f}) = \{\mathsf{pos}, \mathsf{poly}\}$. The function removes pos from $S(\mathsf{y})$ because there does not exist $q_\mathsf{f} \in S(\mathsf{f})$ and $q_\mathsf{x} \in S(\mathsf{x})$ that satisfy $\mathsf{pos} \rhd q_\mathsf{f} <: q_\mathsf{x}$. After the application of the transfer function, S' is updated to $S'(\mathsf{x}) = \{\mathsf{poly}\}$, $S'(\mathsf{y}) = \{\mathsf{poly}, \mathsf{neg}\}$, and $S'(\mathsf{f}) = \{\mathsf{poly}\}$.

Note the difference in inferring N and $N \times \mathsf{ReIm}$ types. When inferring N, subtyping constraints at explicit and implicit assignments degenerate into equality constraints whenever reference types are involved. For example, at field read, whenever x and f are of reference type, the constraint we must satisfy becomes $q_\mathsf{y} \rhd q_\mathsf{f} = q_\mathsf{x}$, not $q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}$. When inferring $N \times \mathsf{ReIm}$, we first infer ReIm types with our tool ReImInfer [12]. Subsequently, we infer the N component: if the right-hand side of a constraint has $\mathsf{readonly}$ type in its ReIm component, we apply the subtyping constraint in the N component; otherwise, we apply the equality constraint.

The set-based solver repeats the above process for each statement and refines the sets until either (1) the iteration reaches a fixpoint, or (2) a variable gets assigned the empty set, in which case the inference terminates with a *type error*. If the set-based solver arrives at a type error, this means that the initial set of programmer-provided annotations is inconsistent.

The resulting set-based solution S contains all valid typings in the program (just as in ownership types, there are many valid typings for a program). The question is, how do we extract a "desirable" valid typing from this set-based solution?

First, we provide a preference ranking over the qualifiers:

$$\mathsf{pos} > \mathsf{poly} > \mathsf{neg}$$

This ranking induces a ranking over the valid typings as we describe in detail in [11]. The "best" typing is the one that has the largest number of variables typed pos. If there are two or more typings that have the largest number of pos variables, the one (or ones) with the larger number of poly variables is "best". The worst typing is the one that types each variable neg. Informally, the "best" typing maximizes the number of positive qualifiers and minimizes the number of negative ones. Our goal is to infer a typing as close to the "best" typing as possible.

One potential typing, which we call the *maximal typing*, is derived as follows: for each variable x, we pick the maximal element of $S(\mathsf{x})$ according to the above qualifier ranking. If the maximal typing type checks (it provably type checks for many interesting systems: Universe Types [5, 11], ReIm [12], AJ [24, 10]), then it is the "best" typing.

Unfortunately, the maximal typing does not always type check for N. Suppose the set-based solution for statement $\mathsf{x} = \mathsf{y.m()}$ is: $S(\mathsf{x}) = \{\mathsf{neg}\}$, $S(\mathsf{y}) = \{\mathsf{poly}, \mathsf{neg}\}$, and $S(\mathsf{ret}) = \{\mathsf{poly}, \mathsf{neg}\}$. The resulting maximal tying is $\Gamma(\mathsf{x}) = \mathsf{neg}$, $\Gamma(\mathsf{y}) = \mathsf{poly}$, and $\Gamma(\mathsf{ret}) = \mathsf{poly}$. Clearly, this does not type check, because $\mathsf{y} \rhd \mathsf{ret}$ is $\mathsf{poly} \rhd \mathsf{poly} = \mathsf{poly}$, which is not a subtype of neg x. We call a statement a *conflict* if it does not type check with the maximal typing derived from the set-based solution.

Benchmark	Valid			Set-based		
	SFlow	SFlow × ReIm	Change	SFlow	SFlow × ReIm	Change
blojsom-1.9.6	531	317	-40%	346	160	-54%
blueblog-1.0	262	153	-42%	210	129	-39%
friki-2.1.1	158	115	-27%	80	35	-56%
gestcv-1.0	65	52	-20%	60	49	-18%
jboard-0.3	127	47	-63%	46	28	-39%
jspwiki-2.4	7789	5093	-35%	6439	3657	-43%
jugjobs-alpha	75	16	-79%	55	16	-71%
pebble-1.6beta1	1811	998	-45%	959	317	-67%
personalblog-1.2.6	564	223	-60%	443	80	-82%
photov-2.1	1917	615	-68%	1571	377	-76%
roller-0.9.9	4489	2232	-50%	3393	1321	-61%
snipsnap-1.0beta	3638	2174	-40%	1887	1182	-37%
webgoat-0.9	546	211	-61%	242	102	-58%

Table 1: public variables for SFlow and SFlow×ReIm. The Valid column contains the numbers for the inferred valid typing. The Set-based column contains the numbers for the set-based solution.

Fortunately, conflicts occur in only two, well-defined cases:

- At method calls $\mathsf{y.m(z)}$, when $S(\mathsf{z}) = \{\mathsf{neg}\}$, $\mathsf{y} \rhd \mathsf{p}$ is not $\mathsf{readonly}$, $S(\mathsf{p})$ is $\{\mathsf{poly}, \mathsf{neg}\}$ and $S(\mathsf{y}) \supset \{\mathsf{neg}\}$. In this case, we have a choice between (1) being polymorphic in the parameter p, or (2) being neg in p.
- Method return $\mathsf{x} = \mathsf{y.m()}$ when $S(\mathsf{x}) = \{\mathsf{neg}\}$, $S(\mathsf{ret}) = \{\mathsf{poly}, \mathsf{neg}\}$ and $S(\mathsf{y}) \supset \{\mathsf{neg}\}$. Again, we have choice between (1) being polymorphic in the return type, or (2) being neg in the return type.

We resolve conflicts automatically by always opting for choice (1). I.e., we choose to be polymorphic in the parameter/return type. This choice is natural as it strives to infer polymorphic method signatures. Fortunately, conflicts are relatively rare and the inference arrives at a valid typing. We note however that we have no guarantee as for how close this typing is to the "best" typing.

7. EMPIRICAL RESULTS

In order to evaluate the precision improvement resulting from composing with ReIm, we implement SFlow with equality constraints and SFlow×ReIm within our type inference framework [11]. We run SFlow and SFlow×ReIm on 13 Java web applications of size ranging from 1843 LOC to 127 KLOC. All sinks from Livshits et. al. [13] were annotated as public, but no sources were annotated as secret (including the sources would have lead to type errors, as these web applications contain true unsafe information flow, and therefore no valid typing can be obtained, not even with SFlow×ReIm).

Table 1 presents results of running SFlow and SFlow×ReIm on all benchmarks. We show the number of public (i.e., neg) variables. Less public variables means better precision. This notion of precision is motivated not only by the notion of "best" typing we discussed in Section 6, but also by practical consideration — the further the sink annotations propagate, the more likely it is they will clash with source annotations.

The **Valid** column in Table 1 contains the number of public variables in the inferred valid typing, and the **Set-based** column contains the numbers of {public} sets in the set-based solution. The latter is the lower bound (i.e. any valid typing by SFlow or SFlow×ReIm will get at least as many public variables as the respective **Set-based** column shows). Evidently, there is a significant precision improvement due to the composition with ReIm. Even when comparing SFlow×ReIm **Valid** with SFlow **Set-based** (the lower bound for SFlow), SFlow×ReIm still gets 20% improvement on average.

The improvement is also reflected by one benchmark, jugjobs, which is accepted by SFlow×ReIm but rejected by SFlow, resulting in 10 (false positive) type errors when enabling all sources as in [13].

The improvement is due to the fact that ReIm enables subtyping, which limits the propagation of neg qualifiers.

8. RELATED WORK

The closest related work is the work by Shankar et al. [17], which presents a type system for detecting string format vulnerabilities for C programs, and more generally, the work on type qualifiers [6, 8]. In this work, the *polymorphic* function is provided as an extension, while in our case, polymorphism is built into the type system. While CQual and JQual rely on a pointer analysis to build the dependence graph and propagate type qualifiers, our system encodes polymorphism in the typing rules, which translates naturally to the type inference. To the best of our knowledge, none of the previous systems attempts to use reference immutability to mitigate the effect of equality constraints.

There is a large body of work on taint analysis for web applications [13, 21, 19, 26, 22]. More recently, there is work on taint analysis for Android apps [7]. These approaches are different from ours, in the sense that they use dataflow analysis, and typically require context-sensitive points-to analysis [13, 21]. Of these works, only FlowDroid [7] is publicly available for comparison (since late April'13). We are interested in comparison with these approaches, both theoretical and empirical.

Due to space constraints, we cannot enumerate all work on type systems for information flow control. Classical work in this space includes the type systems by Volpano et al. [25], Myers [14], and Banerjee and Naumann [3]. Our type system, SFlow, is substantially simpler, in the hope that it will permit inference on very large codes such as the Android SDK.

9. CONCLUSIONS AND FUTURE WORK

We presented a system that combines information flow with reference immutability and demonstrated precision improvement. There are two directions of future research. First, we will formalize the large family of systems fitting in the above framework. There are two novel aspects of the formalization we envision. One is a new, "heapless" operational semantics, which to the best of our knowledge, has not been studied in the literature. Another is the interpretation from a Program Dependence Graph (PDG) point of view and the connection with dataflow analysis. We plan to use ideas from Snelting et al. [18, 9] in this direction. Second, we plan to develop type-based information flow analysis for both Java web applications and Android.

Acknowledgments. We thank the anonymous reviewers for their extensive feedback. This work was supported by NSF Career Award CCF-0642811 and a Google Faculty Research Award (Feb. 2013).

10. REFERENCES

[1] The Checker Framework. `http://types.cs.washington.edu/checker-framework/`, 2013.

[2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, pages 253–, 2002.

[4] D. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[5] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[6] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, May 1999.

[7] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. EC SPRIDE Technical Report TUD-CS-2013-0113. `http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf`, 2013.

[8] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, 2007.

[9] C. Hammer, J. Krinke, and G. Snelting. Information flow control based on path conditions in dependence graphs. In *IEEE ISSSE*, pages 87–96, 2006.

[10] W. Huang and A. Milanova. Inferring AJ Types for Concurrent Libraries. In *FOOL*, 2012.

[11] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[12] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, pages 879–896, 2012.

[13] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security*, 2005.

[14] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[15] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, pages 132–145, 1997.

[16] A. Sampson, W. Dietl, and E. Fortuna. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[17] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.

[18] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[19] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F : Taint Analysis of Framework-based Web Applications. In *OOPSLA*, pages 1053–1068, 2011.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[21] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *PLDI*, pages 87–97, 2009.

[22] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

[23] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[24] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.

[25] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, pages 167–187, 1996.

[26] M. Zanioli, P. Ferrara, and A. Cortesi. Sails: static analysis of information leakage with Sample. In *SAC*, pages 1308–1313, 2012.