Type-based Taint Analysis for Java Web Applications

Technical Report

Wei Huang, Yao Dong, and Ana Milanova

Rensselaer Polytechnic Institute

January 17, 2014

Abstract. Static taint analysis detects information flow vulnerabilities. It has gained considerable importance in the last decade, with the majority of work focusing on dataflow and points-to-based approaches.
In this paper, we advocate *type-based taint analysis*. We present SFlow, a context-sensitive type system for secure information flow, and SFlowInfer, a corresponding worst-case cubic inference analysis. Our approach effectively handles reflection, libraries and frameworks, features notoriously difficult for dataflow and points-to-based taint analysis.
We implemented SFlow and SFlowInfer. Empirical results on 13 real-world Java web applications show that our approach is scalable and also precise, achieving false positive rate of 15%.

1 Introduction

Information flow vulnerabilities are one of the most common security problems according to OWASP [22]. A common information flow vulnerabiltiy is SQL injection, shown in the example in Fig. 1 (adapted from [15]).

```
HttpServletRequest request = ...;
Statement stat = ...;
String user = request.getParameter(''user'');
StringBuffer sb = ...;
sb.append("SELECT * FROM Users WHERE name = ");
sb.append(user);
String query = sb.toString();
stat.executeQuery(query);
```

Fig. 1. SQL Injection Example.

In this example, the user parameter of the HTTP request is obtained through request.getParameter("user") and stored in variable user, which is later appended to an SQL query string and sent to a database for execution: stat.executeQuery(query). At a first glance, this code snippet is unremarkable. However, if a malicious end-user supplies the user parameter with the value of "John OR 1 = 1", the unauthorized end-user can gain access to the information of all other users, because the WHERE clause always evaluates to true. Other information flow vulnerabilities include cross-site scripting, HTTP response splitting, path traversal and command injection [15].

Static taint analysis detects information flow vulnerabilities. It automatically detects flow from untrusted *sources* to security-sensitive *sinks*. In the example in Fig. 1, the return value of HttpServletRequest.getParameter() is a source, and the parameter p of Statement.executeQuery(String p) is a sink.

Research on static taint analysis for Java web applications has largely focused on dataflow and points-to-based approaches [8,15,28,30,31]. One issue with these approaches is that they usually rely on context-sensitive points-to analysis, which is expensive and non-modular (i.e., it requires a whole program). Arguably the toughest issue is dealing with reflection, libraries (JDK and third-party), and frameworks (Struts, Spring, Hibernate, etc.), features notoriously difficult for dataflow and points-to analysis, and yet ubiquitous in Java web applications.

In this paper, we advocate *type-based taint analysis*. Specifically, we present SFlow, a context-sensitive type system for secure information flow, and SFlowInfer, a corresponding worst-case cubic inference analysis. We leverage the inference and checking framework we built in previous work [13], which we have used to infer and check object ownership [13] and reference immutability [14]. Programmers only add a few annotations to specify sources and sinks, and the inference analysis infers a concrete typing or reports type errors indicating information flow violations. Evaluations on 13 real-world Java web applications have shown that our type-based taint analysis achieves both precision and scalability. It has zero false positive for most benchmarks and about 15% false positives on average.

Our inference is modular and compositional. It is modular in the sense that it can analyze any given set of classes L. Unknown callees in L are handled using appropriate defaults. Callers of L can be analyzed separately and composed with L without reanalysis of L. The inference requires annotations *only on sources and sinks*. Once the sources and sinks are built into annotated libraries, web applications are analyzed *without any input from the user*. Our approach effectively handles reflection, libraries, and frameworks. This handling is possible because SFlow does not require abstraction of heap objects, as it models flow from one variable to another through subtyping. To the best of our knowledge, this is the first type-based taint analysis for Java web applications, as well as the first analysis that is provably low polynomial and yet precise.

The paper makes the following contributions:

– SFlow, a context-sensitive type system for secure information flow.
– SFlowInfer, a novel, cubic inference analysis for SFlow.
– Effective handling of reflection, libraries and frameworks.
– An empirical evaluation on Java web applications of up to 126kLOC, comprising 473kLOC in total.

The rest of the paper is organized as follows. Sect. 2 describes the SFlow type system. Sect. 3 presents the dynamic semantics and soundness argument. Sect. 4 describes the inference analysis. Sect. 5 describes techniques for handling of reflection, libraries and frameworks. Sect. 6 presents the empirical evaluations. Sect. 7 discusses the related work, and Sect. 8 concludes the paper.

2 SFlow Type System

This section first describes the basic type qualifiers in SFlow (Sect. 2.1) followed by the extension for context sensitivity (Sect. 2.2). It proceeds to formalize SFlow (Sect. 2.3), and combine SFlow with reference immutability (Sect. 2.4).

2.1 SFlow Qualifiers

There are two basic type qualifiers in SFlow: tainted and safe.

- tainted: A variable x is tainted, if there is flow from a source to x. Sources, e.g., the return value of ServletRequest.getParameter(), are annotated as tainted.
- safe: A variable x is safe if there is flow from x to a sensitive sink. Sinks, e.g., the parameter p of Statement.executeQuery(String p), are annotated as safe.

SFlow disallows flow from tainted sources to safe sinks. Therefore, we define the following subtyping hierarchy[1]:

$$\mathsf{safe} <: \mathsf{tainted}$$

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2 (q is also a subtype of itself: $q <: q$). Thus, assigning a safe variable to a tainted one is allowed:

```
safe int s = ...;
tainted int t = s;
```

but assigning a tainted variable to a safe one is disallowed:

```
tainted int t = ...;
safe int s = t; // type error!
```

In the SQL injection example in Fig. 1, the return value of getParameter() is annotated as tainted, and the parameter of executeQuery(String p) is annotated as safe, as they are a source and a sink, respectively. The other variables are tainted:

```
...
tainted String user = request.getParameter("user");
tainted StringBuffer sb = ...;     // it includes the tainted user
sb.append("SELECT * FROM Users WHERE name = ");
sb.append(user);
tainted String query = sb.toString();
stat.executeQuery(query);          // type error!
```

[1] Note that this is the desired subtyping. Unfortunately, this subtyping is not always safe, as we discuss in detail in Sect. 2.4.

```
1 String user = request.getParameter("user");
2 StringBuffer sb1 = ...; StringBuffer sb2 = ...;
3 sb1.append("SELECT * FROM Users WHERE name = ");
4 sb2.append("SELECT * FROM Users WHERE name = ");
5 sb1.append(user);
6 sb2.append("John");
7 String query = sb2.toString();
8 stat.executeQuery(query);
```

Fig. 2. Context sensitivity example.

Since it is not allowed to assign the tainted query to the safe parameter of executeQuery(String p), statement 8 does not type-check, resulting in a type error. The type error signals an information flow violation.

2.2 Context Sensitivity

Context sensitivity is crucial to the typing precision of SFlow. Note that in the context-insensitive typing above, methods append and toString must be typed as follows (code throughout the paper makes parameter this explicit):

```
tainted StringBuffer append(tainted StringBuffer this, tainted String s) {...}
tainted String toString(tainted StringBuffer this) {...}
```

Such context-insensitive typing is imprecise, because it types the return value of toString as tainted. Consider the example in Fig. 2. query at line 7 is not tainted, but it is typed tainted because of the tainted return value of toString. Therefore, the program is rejected, even though it is safe.

SFlow achieves context sensitivity by making use of a polymorphic type qualifier, poly, and *viewpoint adaptation*.

- poly: The poly qualifier expresses context sensitivity. poly is interpreted as tainted in some invocation contexts and as safe in other contexts.

The subtyping hierarchy becomes

$$\mathsf{safe} <: \mathsf{poly} <: \mathsf{tainted}$$

and append and toString are typed as follows:

```
poly StringBuffer append(poly StringBuffer this, poly String s) {...}
poly String toString(poly StringBuffer this) {...}
```

The poly qualifiers must be interpreted according to invocation context. Intuitively, the role of *viewpoint adaptation* (which we elaborate upon shortly), is to interpret the poly qualifiers according to the invocation context. In Fig. 2, poly is interpreted as tainted at call sb1.append(user), and as safe at call sb2.append("John"). As a result, the tainted argument in the call through sb1 does not propagate to sb2; thus, query at line 7 is typed safe, and the type error at line 8 is avoided.

The type of a poly field f is interpreted in the context of the receiver at the field access. If the receiver x is tainted, then x.f is tainted. If the receiver x is safe, then x.f is safe. An instance field can be tainted or poly, but it cannot be safe; this is necessary to ensure soundness.

Viewpoint adaptation is a concept from Universe Types [5]. , which can be adapted to Ownership Types [4] and ownership-like type systems such as AJ [6, 32]. Viewpoint adaptation of a type q' from the viewpoint of another type q, results in the adapted type q''. This is written as $q \rhd q' = q''$. Viewpoint adaptation adapts fields, formal parameters, and method return values from the viewpoint of the receiver at the field access or method call.

The viewpoint adaptation operation is as follows:

$$\begin{aligned} \_ \rhd \mathsf{tainted} &= \mathsf{tainted} \\ \_ \rhd \mathsf{safe} &= \mathsf{safe} \\ q \rhd \mathsf{poly} &= q \end{aligned}$$

The underscore denotes a "don't care" value. Qualifiers tainted and safe do not depend on the viewpoint (context). Qualifier poly depends on the viewpoint; in fact, it adapts to that viewpoint (context).

2.3 Typing Rules

Now we are ready to define the typing rules for SFlow. For brevity, we restrict our formal attention to a core calculus in the style of Vaziri et al. [32] whose syntax appears in Fig. 3. 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 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. The SFlow 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.

Fig. 4 shows the typing rules. The rules create subtyping constraints at explicit assignments (e.g., x = y, x = y.f) and at implicit assignments (e.g., assignments from actual arguments to formal parameters). The rules for field access, (TREAD) and (TWRITE), adapt the field f from the viewpoint of the *receiver* y, and create the expected subtyping constraints. The rule for method call, (TCALL), adapts formal parameters this and p and return value ret from the viewpoint of the *receiver* y, and creates the subtyping constraints that capture flows from actual arguments to formal parameters, and from return value to the left-hand-side of the call assignment.

Let us return to the example in Fig. 2. Method append is polymorphic, i.e., it is typed as follows:

poly StringBuffer append(**poly** StringBuffer this, **poly** String s) {...}

cd	$::=$	$\mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\}$	*class*
fd	$::=$	$t'\ \mathsf{f}$	*field*
md	$::=$	$t\ \mathsf{m}(t\ \mathsf{this},\ t\ \mathsf{x})\ \{\ \overline{t\ \mathsf{y}}\ \mathsf{s};\ \mathsf{return\ y}\ \}$	*method*
s	$::=$	$\mathsf{s};\mathsf{s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.f} \mid \mathsf{y.f} = \mathsf{x} \mid \mathsf{x} = \mathsf{y.m(z)}$	*statement*
t	$::=$	$q\ \mathsf{C}$	*qualified type*
q	$::=$	$\mathsf{tainted} \mid \mathsf{poly} \mid \mathsf{safe}$	*qualifier*
t'	$::=$	$q'\ \mathsf{C}$	*field qualified type*
q'	$::=$	$\mathsf{tainted} \mid \mathsf{poly}$	*field qualifier*

Fig. 3. Syntax. C and D are class names, f is a field name, m is a method name, and x, y, and z are names of local variables, formal parameters, or parameter this. As in the code examples, this is explicit. For simplicity, we assume all names are unique.

Let sb1 be typed tainted. The call at line 5, namely sb1.append(user), accounts for the following constraint (for brevity, for the rest of the paper, we typically use only the variable, e.g., user, instead of the more verbose q_{user}):

$$\mathsf{user} <: \mathsf{s1} \rhd \mathsf{s} \;\equiv\; \mathsf{user} <: \mathsf{s1} \rhd \mathsf{poly} \;\equiv\; \mathsf{user} <: \mathsf{s1}$$

Since user and s1 are tainted, the call at line 5 type-checks. Now let sb2 be typed safe. The call at line 6, sb2.append("John"), accounts for constraint:

$$\text{“John”} <: \mathsf{s2} \rhd \mathsf{s} \;\equiv\; \text{“John”} <: \mathsf{s2} \rhd \mathsf{poly} \;\equiv\; \text{“John”} <: \mathsf{s2}$$

Since string constant "John" and s2 are both safe, this type-checks as well. In the first context of invocation of append we interpreted poly s as tainted, while in the second context, we interpreted it as safe.

Method overriding is handled by the standard constraints for function subtyping. If m′ overrides m we require

$$\mathit{typeof}(\mathsf{m}') <: \mathit{typeof}(\mathsf{m})$$

and thus,

$$(q_{\mathsf{this}_{\mathsf{m}'}}, q_{\mathsf{p}_{\mathsf{m}'}} \to q_{\mathsf{ret}_{\mathsf{m}'}}) \;<:\; (q_{\mathsf{this}_{\mathsf{m}}}, q_{\mathsf{p}_{\mathsf{m}}} \to q_{\mathsf{ret}_{\mathsf{m}}})$$

This entails $q_{\mathsf{this}_{\mathsf{m}}} <: q_{\mathsf{this}_{\mathsf{m}'}}$, $q_{\mathsf{p}_{\mathsf{m}}} <: q_{\mathsf{p}_{\mathsf{m}'}}$, and $q_{\mathsf{ret}_{\mathsf{m}'}} <: q_{\mathsf{ret}_{\mathsf{m}}}$.

As it is evident from these typing rules, we consider only explicit flows (i.e., data dependences). To the best of our knowledge, all effective static taint analyses [1, 2, 8, 15, 28, 30, 31] forgo implicit flows.

2.4 Composition with Reference Immutability

The reader has likely noticed that subtyping safe <: poly <: tainted is not always sound. Suppose the field f of class A is poly in the following example:

```
tainted B tf = ...;
safe A s = ...;
```

$$\text{(TNEW)}\;\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}} \qquad \text{(TWRITE)}\;\frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}$$

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

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

Fig. 4. Typing rules. Function *typeof* retrieves the SFlow types of fields and methods, Γ is a type environment that maps variables to SFlow qualifiers.

```
tainted A t = s;    // because of safe <: tainted
t.f = tf;           // t.f is tainted
safe B sf = s.f;    // s.f is safe, unsafe flow!
```

The program type-checks, but the tainted variable tf flows to safe variable sf. This is the known problem of subtyping in the presence of mutable references, also known as the issue with Java's covariant arrays [21].

The standard solution is to disallow subtyping for references [24]. This solution demands two sets of qualifiers, safe <: poly <: tainted for simple types (e.g., `int,char`), and Safe, Poly, Tainted for reference types. While subtyping is allowed for simple types, it is disallowed for reference types. For example, EnerJ [24] 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 reference types leads to imprecision, i.e., the type system rejects valid programs. It amounts to using *equality constraints* as opposed to subtyping constraints, and thus, propagating safe and tainted qualifiers bi-directionally, resulting in often unnecessary propagation [18]. 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 [29], which uses unification (i.e., equality) constraints, is substantially less precise than Andersen's points-to analysis [3], which uses subset constraints.

The following example illustrates the problem:

```
ServletRequest request = ...;
String user = request.getParameter(''user'');
String str = ''abc'';
```

```
4 user = str;            // Equality constraint: user and str are of same type!
5 PrintWriter writer = resp.getWriter();
6 writer.print(str);    // type error!
```

Recall that the return value of ServletRequest.getParameter() is tainted, and the parameter of PrintWriter.print() is safe. If we disallowed subtying for references, the program would be rejected, even though there is no unsafe flow. This is because statement user = str would trigger an equality constraint instead of a subtyping constraint. The equality constraint would force user and str to be of the same type. However, this is impossible for a well-typed program, because statement 2 requires that user be tainted and 6 requires that str be safe.

We propose a solution using reference immutability, which allows for limited subtyping and improves precision. It is a theorem that subtyping is safe when the reference on the left-hand-side of the assignment (explicit or implicit) is an *immutable reference*, that is, the state of the referenced object, including its transitively reachable state, *cannot be mutated through this reference.*

We compose SFlow with ReIm, a reference immutability type system we developed in previous work [14]. We run ReImInfer [14], ReIm's inference tool, and obtain ReIm types for all variables. If the ReIm type of the left-hand-side of an assignment is readonly, i.e., it is guaranteed that this left-hand-side is immutable, we use a subtyping constraint in SFlow. Otherwise, i.e., if the ReIm type is not readonly, we use an equality constraint. For example, at (TREAD) $\mathsf{x} = \mathsf{y.f}$, if x is readonly in ReIm, we use constraint $q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}$; otherwise, we use constraint $q_{\mathsf{y}} \rhd q_{\mathsf{f}} = q_{\mathsf{x}}$. Sect. 3 outlines the dynamic semantics and soundness argument.

Returning to the above example, user is readonly and therefore statement 4 induces subtyping constraint str <: user. Therefore, str can be safe and user can be tainted, and the program type-checks.

3 Dynamic Semantics and Soundness

This section presents a dynamic semantics of information flow (Sect. 3.1). It proceeds to outline the argument for soundness of SFlow (Sect. 3.2).

3.1 Dynamic Semantics

First, we define the notion of the *chain*, which captures flow of values from one variable to another. Intuitively, there is a chain from local variable x to local variable y, denoted (x, y), if the value of x flows from x to y. Chains provide a mechanism for reasoning about aliasing.

Chains Fig. 5 shows the rules of the semantics. For brevity, we omit the parts of the semantics that are not strictly necessary. The rules record chains (x, y) in set C. Rule (DASSIGN) $\mathsf{x} = \mathsf{y}$ adds a new chain (w, x) for every chain $(\mathsf{w}, \mathsf{y}) \in C$. There is a chain $(\mathsf{y}, \mathsf{y}) \in C$ (we explain why shortly), and therefore, $\mathsf{x} = \mathsf{y}$ adds

$$\frac{C' = C \cup \{\ (\mathsf{w},\mathsf{x}) \mid (\mathsf{w},\mathsf{y}) \in C\ \}}{\{C\} \quad \mathsf{x} = \mathsf{y} \quad \{C'\}}\ \text{(DASSIGN)}$$

$$\frac{}{\{C\} \quad \mathsf{y.f} = \mathsf{x} \quad \{C\}}\ \text{(DWRITE)}$$

$$\frac{\begin{array}{c} \mathit{pointsto}(\mathsf{x}) = o \quad \mathit{lastwrite}(o.\mathsf{f})\ =\ \mathsf{y'.f} = \mathsf{x'} \\ C' = C \cup \{\ (\mathsf{w},\mathsf{x}) \mid (\mathsf{w},\mathsf{x'}) \in C\ \} \end{array}}{\{C\} \quad \mathsf{x} = \mathsf{y.f} \quad \{C'\}}\ \text{(DREAD)}$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathsf{m}) = \mathsf{this}, \mathsf{p}, \bar{\mathsf{l}}, \mathsf{ret} \\ C' = C \cup \{\ (\mathsf{w},\mathsf{this}) \mid (\mathsf{w},\mathsf{y}) \in C\ \} \cup \{\ (\mathsf{w'},\mathsf{p}) \mid (\mathsf{w'},\mathsf{z}) \in C\ \} \cup \{\ \overline{(\mathsf{l},\mathsf{l})}\ \} \end{array}}{\{C\} \quad \mathsf{x} = \mathsf{y.m(z)} \quad \{C'\}}\ \text{(DCALL)}$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathsf{m}) = \mathsf{this}, \mathsf{p}, \bar{\mathsf{l}}, \mathsf{ret} \\ C' = C \cup \{\ (\mathsf{w},\mathsf{x}) \mid (\mathsf{w},\mathsf{ret}) \in C\ \} \end{array}}{\{C\} \quad \mathsf{x} = \mathsf{y.m(z)} \quad \{C'\}}\ \text{(DRETURN)}$$

Fig. 5. Dynamic semantics that records chains. Each statement s takes as input a set of chains C, and produces a new set of chains C'; this is denoted as $\{C\}\ \ \mathsf{s}\ \ \{C'\}$. Function *pointsto*(x) returns the object o x refers to, and *lastwrite*(o.f) returns the last statement y'.f = x' that wrote location o.f. Function *mbody* takes as argument the called method m, and returns the body of m. The body consists of implicit parameter this, formal parameter p, set of local variables $\bar{\mathsf{l}}$ and return variable ret.

the chain (y,x) as expected. Rule (DWRITE) has no effect on C. Chains are defined over local variables: one end of the chain, x, is a local variable in one frame, and the other end, y, is another local variable that may be in a different frame. In our semantics, (DWRITE) y'.f = x' plays a role only in combination with (DREAD) x = y.f, where y and y' refer to the same heap object o and y'.f = x' was the last write to o.f before the read x = y.f. (DWRITE) y'.f = x' and (DREAD) x = y.f contribute a chain $(\mathsf{x'},\mathsf{x})$ (as well as other chains).

As it is customary in dynamic semantics [6, 17, 32], we break the static rule (TCALL) into two parts: (DCALL) and (DRETURN). Rule (DCALL) has two roles. First, it adds new chains due to the implicit assignments to this and formal parameter p. Second, it adds a self-chain (l,l) for every local variable l. Rule (DRETURN) adds new chains that account for the flows due to the implicit assignment of ret to the left-hand-side x of the call assignment. Consider:

```
1  class X {
2    Y f;
3    void set(Y param) {
4      this.f = param;
5    }
6    Y get() {
7      ret = this.f;
8      return ret;
9    }
10 }
```

```
1  main() {
2    x = new X()   o
3    x.set(a);
4    y = x
5    b = y.get();
6  }
```

Line 3 in main and rule (DCALL) contribute chains $(\mathsf{x}, \mathsf{this}_{\mathsf{set}})$ and $(\mathsf{a}, \mathsf{param})$. Line 4 in X.set does not contribute any new chains. Line 4 in main contributes chain (x,y), and line 5 contributes $(\mathsf{y}, \mathsf{this}_{\mathsf{get}})$ and $(\mathsf{x}, \mathsf{this}_{\mathsf{get}})$. Line 7 in X reads the f field of object o. Since the last write to o.f is at line 4, line 7 and (DREAD) contribute chains (param,ret) and (a,ret). Finally, line 5 in main and (DRET) contribute (ret,b), (param,b) and (a,b).

An important aspect of this semantics is that it forgoes the heap. It turns out, chains provide a sufficient mechanism for reasoning about aliasing. Specifically, the following proposition holds. If y' and y refer to the same object o, making $\mathsf{y}'.\mathsf{f}$ and y.f aliases, then there are chains $(\mathsf{w}, \mathsf{y}')$ and (w, y) in C, where w is the left-hand-side of the object creation assignment that created o. In our running example, $\mathsf{this}_{\mathsf{set}}$ and $\mathsf{this}_{\mathsf{get}}$ refer to the same object, the one created at line 1 in main and denoted by o. As we showed earlier, there are chains $(\mathsf{x}, \mathsf{this}_{\mathsf{set}})$ and $(\mathsf{x}, \mathsf{this}_{\mathsf{get}})$ in C before the execution of ret = this.f. In our static semantics, i.e., the SFlow type system, chains are tracked through subtyping, which obviates the need for pointer analysis.

Extended Chain In addition to the chain, we define the *extended chain*, which captures flows from the transitively reachable state of x to y. Informally, there is an extended chain from local variable x to local variable y, denoted $(\mathsf{x}, \mathsf{y})^+$, if the value of x, or a value that is part of x's transitively reachable state, flows to y. The dynamic semantics that records extended chains in E is the same as the semantics that records chains (Fig. 5), except for rule (DREAD):

$$\frac{\substack{\textsc{(dread)}\\ pointsto(\mathsf{x}) = o \quad lastwrite(o.\mathsf{f}) \;=\; \mathsf{y}'.\mathsf{f} = \mathsf{x}' \\ E' = E \cup \{\, (\mathsf{w}, \mathsf{x})^+ \mid (\mathsf{w}, \mathsf{x}')^+ \in E \,\} \cup \{\, (\mathsf{w}', \mathsf{x})^+ \mid (\mathsf{w}', \mathsf{y})^+ \in E \,\}}}{\{E\} \quad \mathsf{x} = \mathsf{y}.\mathsf{f} \quad \{E'\}}$$

The rule "connects" y and x, which is needed to account for the transitive state reachable through a variable. In the running example, the extended chains that originate at x are the following: $(\mathsf{x}, \mathsf{this}_{\mathsf{set}})^+$, $(\mathsf{x}, \mathsf{y})^+$, $(\mathsf{x}, \mathsf{this}_{\mathsf{get}})^+$, $(\mathsf{x}, \mathsf{ret})^+$, and $(\mathsf{x}, \mathsf{b})^+$. The targets of these extended chains account for the state, including transitive state, reachable from x. Intuitively, an extended chain $(\mathsf{x},\mathsf{y})^+$ captures "interference" or "information flow" from the source x to the target y.

3.2 Soundness

Runtime Interpretation of SFlow Types Recall the SFlow type system and its 3 qualifiers:

$$\mathsf{safe} <: \mathsf{poly} <: \mathsf{tainted}$$

At runtime, qualifier poly is interpreted as safe or tainted, depending on the invocation context [25]. Qualifier safe is always interpreted as safe, regardless of invocation context, and tainted is always interpreted as tainted.

The *frame abstraction* of a stack frame F, denoted by $\tau(F)$, is the *viewpoint adapter* at the call $\mathsf{x} = \mathsf{y.m(z)}$ that pushed F on the stack. For SFlow, the frame abstraction is the static type of the receiver y, q_{y}. Let x be a reference variable with static type $q_{\mathsf{x}} = \Gamma(\mathsf{x})$, in frame F_k, and let there be the following stack configuration:

$$S = \langle F_{\mathsf{main}} \rangle \langle F_1 \rangle \ldots \langle F_{k-1} \rangle \langle F_k \rangle$$

The *runtime interpretation of the SFlow type of* x, denoted $\mathit{RiSFlow}(\mathsf{x})$, is defined as follows:

$$\mathit{RiSFlow}(\mathsf{x}) = \tau(F_1) \rhd \ldots \tau(F_{k-1}) \rhd \tau(F_k) \rhd q_{\mathsf{x}}$$

We note that viewpoint adaptation $\rhd$ is associative, and therefore parentheses are unnecessary. Clearly, if q_{x} is safe or tainted, then $\mathit{RiSFlow}(\mathsf{x})$ is safe or tainted, respectively. In other words, when q_{x} is safe or tainted, the invocation context of x's method is irrelevant. The interesting case arises when q_{x} is poly. x assumes the first type of $\tau(F_k)$, $\tau(F_{k-1})$, etc. that is not poly, i.e., that is safe or tainted. To ensure that poly always has well-defined runtime interpretation as safe or tainted, we require that no variable in main is poly. Therefore, if all of $\tau(F_k), \tau(F_{k-1}) \ldots \tau(F_2)$ are poly, $\mathit{RiSFlow}(\mathsf{x}) = \tau(F_1) \neq \mathsf{poly}$. We write $\mathit{Stack} \rhd q_{\mathsf{x}}$ instead of $\tau(F_1) \rhd \ldots \tau(F_{k-1}) \rhd \tau(F_k) \rhd q_{\mathsf{x}}$ whenever the sequence of frames on the stack is not important.

Recall the example from the previous section, now with SFlow types:

```
class X {
  poly Y f;
  void set(poly X this, poly Y param) {
    this.f = param;
  }
  poly Y get(poly X this) {
    ret = this.f;
    return ret;
  }
}
```

```
1 main() {
2   safe X x = new safe X()   o
3   x.set(a);
4   safe Y y = x
5   safe Y b = y.get();
6 }
```

X is a polymorphic class that is instantiated to safe in main. Let F_1 be the frame for method set, pushed on the stack as a result of the call in line 3 of main.

$\mathit{RiSFlow}(\mathsf{this}_{\mathsf{set}})$ and $\mathit{RiSFlow}(\mathsf{param})$ in set are interpreted as follows:

$$\tau(F_1) \rhd \mathsf{poly} = q_{\mathsf{x}} \rhd \mathsf{poly} = \mathsf{safe}$$

Runtime interpretation can be applied to ReIm [14] as well. The runtime interpretation of the ReIm type of x, denoted $\mathit{RiReIm}(\mathsf{x})$, is computed as in SFlow, with the difference that in ReIm the viewpoint adapter is not the receiver y, but the left-hand-side x of the call assignment $\mathsf{x} = \mathsf{y.m(z)}$.

Well-formedness The well-formedness rules are shown in Fig. 6. Essentially, the rules require that for every chain (w, x) and extended chain $(\mathsf{w}, \mathsf{x})^+$, the runtime type of the source w is a subtype of the runtime type of the target x. Thus, well-formed runtimes cannot form chains or extended chains, where w is $\mathsf{tainted}$ and x is safe. Well-typedness guarantees well-formedness, which we argue shortly, and this entails that well-typed programs guarantee the absence of flow from $\mathsf{tainted}$ sources to safe sinks (of course, given that the extended chain is a suitable representation of information flow).

Note the stronger requirement for chains — when the ReIm runtime type of the target x is $\mathsf{mutable}$, the SFlow types of the source and the target must be equal. This is necessary for the safe handling of aliasing.

$$\textsc{(wf-chain)}\quad \frac{\forall (\mathsf{w}, \mathsf{x}) \in C \quad \begin{cases} \mathit{RiSFlow}(\mathsf{w}) = \mathit{RiSFlow}(\mathsf{x}) & \text{if } \mathit{RiReIm}(\mathsf{x}) = \mathsf{mutable} \\ \mathit{RiSFlow}(\mathsf{w}) <: \mathit{RiSFlow}(\mathsf{x}) & \text{if } \mathit{RiReIm}(\mathsf{x}) = \mathsf{readonly} \end{cases}}{C \text{ is WF}}$$

$$\textsc{(wf-extended-chain)}\quad \frac{\forall (\mathsf{w}, \mathsf{x})^+ \in E \quad \mathit{RiSFlow}(\mathsf{w}) <: \mathit{RiSFlow}(\mathsf{x})}{E \text{ is WF}}$$

$$\textsc{(wf-configuration)}\quad \frac{C \text{ is WF} \quad E \text{ is WF}}{CE \text{ is WF}}$$

Fig. 6. Well-formedness rules.

Soundness Theorems We are ready to state the two soundness theorems.

Theorem 1. (PRESERVATION) *If CE is WF and $CE \xrightarrow{\mathsf{s}} C'E'$, then $C'E'$ is WF.*

Proof. We sketch the proof of the theorem. As it is customary for such proofs, we must consider cases for all kinds of statements: (DASSIGN), (DWRITE), (DREAD), (DCALL) and (DRETURN). The proof is by induction on the steps of the dynamic semantics.

The most interesting case is (DREAD) $\mathsf{x} = \mathsf{y.f}$. Let o be the object y refers to, let y'.f = x' be the last write to o.f, and let C be the set of chains just before the execution of x = y.f. As we claimed earlier, since y and y' refer to the same object, there must exist chains $(\mathsf{w}, \mathsf{x}) \in C$ and $(\mathsf{w}, \mathsf{y}) \in C$, where w is the left-hand-side at the object creation assignment that created o. By the inductive hypothesis, we have $\mathit{RiSFlow}(\mathsf{w}) = \mathit{RiSFlow}(\mathsf{y}')$ (since y' is mutated at y'.f = x', $\mathit{RiReIm}(\mathsf{y}')$ is clearly mutable), and $\mathit{RiSFlow}(\mathsf{w}) <: \mathit{RiSFlow}(\mathsf{y})$. Thus, we have

$$\mathit{RiSFlow}(\mathsf{y}') <: \mathit{RiSFlow}(\mathsf{y})$$

We write $\mathit{RiSFlow}(\mathsf{y}') <: \mathit{RiSFlow}(\mathsf{y})$ as

$$\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{y}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{y}}$$

We must show that all new chains are well-formed. Thus, we must show $\mathit{RiSFlow}(\mathsf{x}') <: \mathit{RiSFlow}(\mathsf{x})$, or equivalently,

$$\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$$

(Strictly, we must show that $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} = \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$ if $\mathit{RiReIm}(\mathsf{x})$ is mutable, and that at least $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$ if $\mathit{RiReIm}(\mathsf{x})$ is readonly. This can be proven with appropriate reasoning about ReIm. For brevity, we only show the above special case.)

Clearly, x' and x are in the frames of y' and y respectively; the runtime interpretation of x' uses $\mathit{Stack}_{\mathsf{y}'}$ just as y' does, and the runtime interpretation of x uses $\mathit{Stack}_{\mathsf{y}}$ just as y.

The well-typedness of the program entails

$$q_{\mathsf{x}'} <: q_{\mathsf{y}'} \rhd q_{\mathsf{f}} \text{ and } q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}$$

If q_{f} is tainted, then q_{x} is tainted and $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$ holds for any value of $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'}$ because $\mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$ is tainted.

If q_{f} is poly then we have

$$q_{\mathsf{x}'} <: q_{\mathsf{y}'} \text{ and } q_{\mathsf{y}} <: q_{\mathsf{x}}$$

We say that viewpoint adaptation is *order preserving* if for every triple of qualifiers q, q', q'', $q' <: q'' \Rightarrow q \rhd q' <: q \rhd q''$. One can easily show that viewpoint adaptation in SFlow is order preserving.

Since viewpoint adaptation is order preserving, we have

$$\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} <: \mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{y}'} \text{ and } \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{y}} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$$

and since $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{y}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{y}}$ holds (see above), $\mathit{Stack}_{\mathsf{y}'} \rhd q_{\mathsf{x}'} <: \mathit{Stack}_{\mathsf{y}} \rhd q_{\mathsf{x}}$ as well.

We must show that all new extended chains are well-formed as well. This can shown analogously.

Theorem 2. (PROGRESS) *If* CE *is WF then* $CE \xrightarrow{\mathsf{s}} C'E'$.

4 Type Inference

Type inference derives a *valid typing*, i.e., an assignment of qualifiers to program variables that type-checks with the rules in Fig. 4. If inference succeeds, then the program is safe, i.e., there are no flows from sources to sinks. If it fails, then a valid typing does not exist, meaning that there could be unsafe flow from sources to sinks.

Type inference leverages the framework we developed in [13]. It first computes a *set-based solution* S, which maps variables to *sets* of potential type qualifiers. The key novelty over [13] is the use of *method summary constraints*, which refine the set-based solution, and help derive a valid typing.

4.1 Set-based Solution

The set-based solution is a mapping S from variables to sets of qualifiers. The variables in the mapping can be (1) local variables, (2) parameters (including this), (3) fields, and (4) method returns. For example, $S(\mathsf{x}) = \{\mathsf{poly}, \mathsf{safe}\}$ denotes the type of variable x can be poly, or safe, but not tainted. Programmer-annotated variables, including annotated library variables, are initialized to the singleton set that contains the programmer-provided qualifier. In SFlow, all sources and sinks are programmer-provided, i.e., sources and sinks are annotated as tainted and safe, respectively. Fields are initialized to $S(\mathsf{f}) = \{\mathsf{tainted}, \mathsf{poly}\}$. All other variables are initialized to the maximal set of qualifiers, i.e., $S(\mathsf{x}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$.

The inference creates constraints for all program statements according to the typing rules in Fig. 4. It takes into account ReIm: if the left-hand-side of the assignment is readonly, the inference creates a subtyping constraint; otherwise, it creates an equality constraint. Consider (TREAD) $\mathsf{x} = \mathsf{y.f}$. If x is readonly, the inference creates constraint $q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}$; otherwise, it creates an equality constraint $q_\mathsf{y} \rhd q_\mathsf{f} = q_\mathsf{x}$. In the latter case, the inference actually creates two subtyping constraints that are equivalent to the equality constraint. In the above example, it creates $q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}$ and $q_\mathsf{x} <: q_\mathsf{y} \rhd q_\mathsf{f}$.

Subsequently, the set-based solver iterates over these constraints, and runs $\textsc{SolveConstraint}(c)$ for each constraint c. $\textsc{SolveConstraint}(c)$ removes infeasible qualifiers from the set of variables that participate in c. It works as follows (for a more formal description, see [13]). Consider $\mathsf{x} = \mathsf{y.f}$ again, and suppose x is readonly, thus creating the sole subtyping constraint $q_\mathsf{y} \rhd q_\mathsf{f} <: q_\mathsf{x}$. Suppose that before processing this constraint, we have $S(\mathsf{x}) = \{\mathsf{poly}\}$, $S(\mathsf{y}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$, and $S(\mathsf{f}) = \{\mathsf{tainted}, \mathsf{poly}\}$. The solver removes tainted from $S(\mathsf{y})$ because there do not exist $q_\mathsf{f} \in S(\mathsf{f})$ and $q_\mathsf{x} \in S(\mathsf{x})$ that satisfy $\mathsf{tainted} \rhd q_\mathsf{f} <: q_\mathsf{x}$. Similarly, tainted is removed from $S(\mathsf{f})$. After processing the constraint, S is updated to $S(\mathsf{x}) = \{\mathsf{poly}\}$, $S(\mathsf{y}) = \{\mathsf{poly}, \mathsf{safe}\}$, and $S(\mathsf{f}) = \{\mathsf{poly}\}$. If the infeasible qualifier is the last element in $S(\mathsf{x})$, $\textsc{SolveConstraint}(c)$ keeps this qualifier in $S(\mathsf{x})$, and reports a *type error* at c (we keep the qualifier in order to produce better error reports: a type error $\mathsf{x}\{\mathsf{tainted}\} <: \mathsf{y}\{\mathsf{safe}\}$ is more informative than $\mathsf{x}\{\} <: \mathsf{y}\{\mathsf{safe}\}$).

```
1  void doGet(A this, ServletRequest request, ServletResponse response) {
2    StringBuffer buf = ...;
3    this.foo(buf,buf,request,response);   [buf = this_doGet ▷ b1]      (S(buf) = {tainted})
4  }                                       [buf <: this_doGet ▷ b2]     (S(b2) = {tainted, poly})
5  void foo(A this, StringBuffer b1, StringBuffer b2,
6        ServletRequest req, ServletResponse resp) {
7    String url = req.getParameter("url"); [req ▷ tainted <: url]       (S(url) = {tainted})
8    b1.append(ural);                      [url <: b1 ▷ poly]           (S(b1) = {tainted})
9    String str = b2.toString();           [b2 ▷ poly <: str]           (S(str) = {tainted, poly})
10   PrintWriter writer = resp.getWriter();
11   writer.print(str);                    [str <: writer ▷ safe]       (TYPE ERROR!)
12 }
```

Fig. 7. Aliasing5 example from Stanford SecuriBench Micro. The frame box beside each statement shows the corresponding constraints the statement generates. The oval boxes show propagation during the set-based solution. The constraint at 7 forces url to be tainted, and the constraint at 8 forces b1 to be tainted. The constraint at 3 forces buf to be tainted and the one at 4 forces b2 to be tainted or poly (i.e., the set-based solver removes safe from b2's set). The constraint at 9 then forces str to be tainted or poly. There is a TYPE ERROR at writer.print(str).

The set-based solver iterates over the constraints and refines the sets until it reaches a fixpoint. There are two outcomes: (1) there are no type errors, and (2) there are one or more type errors. If the set-based solver arrives at type errors, this means that the programmer-provided sources and sinks are inconsistent, and the program cannot be typed. In other words, a type error indicates that there could be unsafe flow from a source to a sink.

Consider the Aliasing5 example from Ben Livshits' Stanford SecuriBench Micro benchmarks[2] in Fig. 7. foo is safe when b1 and b2 refer to distinct StringBuffer objects. However, when b1 and b2 are aliased, foo creates dangerous flow from source req.getParameter to a sink, the parameter of PrintWriter.print. Note that the constraint at line 3 is an equality constraint: b1 is mutated at b1.append(url), ReIm infers b1 as mutable, and hence the equality constraint. The set-based solver reports a type error at statement 11; the constraint at 11 is unsatisfiable as it requires that str is safe, which contradicts the finding that str is {tainted, poly}.

4.2 Valid Typing

The set-based solver removes many infeasible qualifiers and in many cases, it discovers type errors. In our experience, the set-based solver, which is worst-case quadratic and linear in practice, discovers the vast majority of type errors, and therefore it is useful on its own. Unfortunately, when the set-based solver terminates without type errors, it is unclear if a valid typing exists or not, and

[2] http://suif.stanford.edu/~livshits/work/securibench-micro/

1: **procedure** RunSolver
2: **repeat**
3: **for each** c in C **do**
4: SolveConstraint(c)
5: **if** c is $q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 1
6: Add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C
7: **else if** c is $q_{\mathsf{x}} \rhd q_{\mathsf{f}} <: q_{\mathsf{y}}$ **and** $S(\mathsf{f})$ is $\{\mathsf{poly}\}$ **then** ▷ Case 2
8: Add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C
9: **else if** c is $q_{\mathsf{x}} <: q_{\mathsf{y}}$ **then** ▷ Case 3
10: **for each** $q_{\mathsf{y}} <: q_{\mathsf{z}}$ in C **do** add $q_{\mathsf{x}} <: q_{\mathsf{z}}$ to C **end for**
11: **for each** $q_{\mathsf{w}} <: q_{\mathsf{x}}$ in C **do** add $q_{\mathsf{w}} <: q_{\mathsf{y}}$ to C **end for**
12: **else if** c is $q_{\mathsf{z}} <: q_{\mathsf{y}} \rhd q_p$ **then** ▷ Case 4
13: **if** $q_p <: q_{p'}$ **and** $q_{\mathsf{y}} \rhd q_{p'} <: q_{\mathsf{x}}$ in C **then** Add $q_{\mathsf{z}} <: q_{\mathsf{x}}$ to C **end if**
14: **end if**
15: **end for**
16: **until** S and C remain unchanged
17: **end procedure**

Fig. 8. Computation of method summary constraints. C is the set of constraints, which initially contains the constraints for program statements, derived as described in Sect. 4.1 (recall that each equality constraint is written as two subtyping constraints). Cases 1 and 2 add $q_{\mathsf{x}} <: q_{\mathsf{y}}$ into C because $q_{\mathsf{y}} \rhd \mathsf{poly}$ always yields q_{y}. Case 3 adds constraints due to transitivity; this case discovers constraints from formals to return values. Case 4 adds constraints between actual(s) and left-hand-side(s) at calls: if there are constraints $q_{\mathsf{z}} <: q_{\mathsf{y}} \rhd q_p$ (flow from actual to formal) and $q_{\mathsf{y}} \rhd q_{p'} <: q_{\mathsf{x}}$ (flow from return value to left-hand-side), and also $q_p <: q_{p'}$ (flow from formal to return value, usually discovered by Case 3), Case 4 adds $q_{\mathsf{z}} <: q_{\mathsf{x}}$. Note that line 4 calls SolveConstraint(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until C and S stay unchanged.

therefore, there is no guarantee of safety. The question is, how do we extract a valid typing, or conversely, show that a valid typing does not exist?

The key idea is to compute what we call *method summary constraints*, which remove additional qualifiers from the set-based solution. These constraints reflect the relations (subtyping or equality) between formal parameters (including this) and return values (ret). Such references are usually "connected" indirectly, e.g. this and ret can be connected through two constraints this <: x and x <: ret. Note that intuitively, the subtyping relation reflects flow: there is flow from this to x, there is flow from x to ret, and due to transitivity, there is flow from this to ret. Once we have computed the relations between formal parameters and return values of a method m, we connect the actual arguments to the left hand sides of the call assignment at calls to m. The computation of method summary constraints is presented in Fig. 8. As an example, consider the following code snippet:

```
class A {
  String f;
  String get()
  {return this.f;}
}
```
$\boxed{\mathsf{this} \rhd \mathsf{f} <: \mathsf{ret}}$

```
A y = ...;
PrintWriter writer = ...;
String x = y.get();
writer.print(x);
```
$\boxed{\mathsf{y} <: \mathsf{y} \rhd \mathsf{this}}$ $\boxed{\mathsf{y} \rhd \mathsf{ret} <: \mathsf{x}}$

$\boxed{\mathsf{x} <: \mathsf{writer} \rhd \mathsf{safe}}$

where generated constraints are shown in the frame boxes beside statements. The set-based solver yields $S(\mathsf{x}) = \{\mathsf{safe}\}$, $S(\mathsf{y}) = \{\mathsf{tainted}, \mathsf{poly}, \mathsf{safe}\}$, $S(\mathsf{this}) = \{\mathsf{poly}, \mathsf{safe}\}$, $S(\mathsf{ret}) = \{\mathsf{poly}, \mathsf{safe}\}$, and $S(\mathsf{f}) = \{\mathsf{poly}\}$. Case 2 in Fig. 8 creates $\mathsf{this} <: \mathsf{ret}$. This entails $\mathsf{y} \rhd \mathsf{this} <: \mathsf{y} \rhd \mathsf{ret}$ since viewpoint adaptation preserves subtyping [18]. Case 4 combines this with constraints $\mathsf{y} <: \mathsf{y} \rhd \mathsf{this}$ and $\mathsf{y} \rhd \mathsf{ret} <: \mathsf{x}$, yielding a new constraint $\mathsf{y} <: \mathsf{x}$. Because tainted and poly are not subtypes of safe, SolveConstraint removes them from $S(\mathsf{y})$, and $S(\mathsf{y})$ becomes $\{\mathsf{safe}\}$.

RunSolver terminates either (1) without type errors, or (2) with type errors, just as the set-based solver. When it terminates without errors, SFlowInfer types each variable x by picking the *maximal* element of $S(\mathsf{x})$, according to the following preference ranking: $\mathsf{tainted} > \mathsf{poly} > \mathsf{safe}$. This *maximal typing* practically always type-checks. In the above example, typing $\Gamma(\mathsf{x}) = \Gamma(\mathsf{y}) = \mathsf{safe}$, $\Gamma(\mathsf{this}) = \Gamma(\mathsf{ret}) = \Gamma(\mathsf{f}) = \mathsf{poly}$ type-checks (in contrast, the maximal typing extracted from the set-based solution, does not type-check). In our experiments, the maximal typing always type-checks, except for 2 constraints in one of our benchmarks, jugjobs. Fortunately, even if the maximal typing does not type-check, it is a theorem that the program is still safe, i.e., there is no flow from sources to sinks. We confirmed this for the 2 constraints in jugjobs.

4.3 Complexity

The inference is dominated by RunSolver. To better reason about complexity, we present an equivalent algorithm in Fig. 9 (i.e., it computes the same fixpoint S). This algorithm merges Case 3 and Case 4 and removes C from the repeat condition. Below, we sketch the proof of why it is safe to remove C. Consider the iteration i, when S stayed unchanged. It is easy to see that C stayed unchanged as well. Suppose that iteration i discovered new constraints, and let $q_{\mathsf{x}} <: q_{\mathsf{y}}$ be the first such constraint. The new constraint cannot be due to Case 1 or Case 2 because $S(\mathsf{f})$ did not change from iteration $i-1$ to iteration i (as S already reached the fixpoint). It cannot be due to Case 3 either: if it were, then there would be two constraints $q_{\mathsf{x}} <: q_{\mathsf{z}}$ and $q_{\mathsf{z}} <: q_{\mathsf{y}}$ already in C due to previous iterations; but then $q_{\mathsf{x}} <: q_{\mathsf{y}}$ would have been discovered in a previous iteration as well (through line 10 if $q_{\mathsf{z}} <: q_{\mathsf{y}}$ were discovered before $q_{\mathsf{x}} <: q_{\mathsf{z}}$, or through line 11, if $q_{\mathsf{x}} <: q_{\mathsf{z}}$ were discovered before $q_{\mathsf{z}} <: q_{\mathsf{y}}$). It cannot be due to Case 4 either, due to similar reasons.

The algorithm in Fig. 9 reaches the *fixpoint* (when S stays unchanged) in $O(n^3)$ time, where n is the size of the program. There are at most $O(3n)$ iterations of the outer loop (line 2), because in each iteration at least one of $O(n)$ references is updated to refer to a smaller set of qualifiers, and each set has at most 3

```
1: procedure RunSolver
2:   repeat
3:     for each c in C do
4:       SolveConstraint(c)
5:       if c is q_x <: q_y ▷ q_f and S(f) is {poly} then        ▷ Case 1
6:         Add q_x <: q_y into C
7:       else if c is q_x ▷ q_f <: q_y and S(f) is {poly} then   ▷ Case 2
8:         Add q_x <: q_y into C
9:       else if c is q_x <: q_y then                              ▷ Case 3
10:        for each q_y <: q_z in C do add q_x <: q_z to C end for
11:        for each q_w <: q_x in C do add q_w <: q_y to C end for
12:        for each q_w <: q_a ▷ q_x and q_a ▷ q_y <: q_z in C do   ▷ Case 4
13:          Add q_w <: q_z to C
14:        end for
15:      end if
16:    end for
17:  until S remains unchanged
18: end procedure
```

Fig. 9. An improved version of the algorithm in Fig. 8 to better reason about complexity. Notice Case 4 is merged into Case 3 and C is removed from the repeat condition. When the inner loop (line 3) discovers a new constraint, it is appended at the end of C, and processed in the same iteration of the outer loop (line 2).

qualifiers. The inner loop (line 3) iterates over at most $O(n^2)$ constraints, because in the worst case every two references can form a constraint, resulting in $O(n^2)$ constraints. Altogether, we have worst-case complexity of $O(n^3)$. Although at first glance lines 11-13 (Cases 3-4) appear to contribute $O(n) * O(n^2) * O(3n)$, a closer look reveals they contribute only $O(n) * O(n^2)$, or $O(n^3)$ (this is because lines 10-13 run only when a new constraint $q_{\mathsf{x}} <: q_{\mathsf{y}}$ is discovered, and there are at most $O(n^2)$ such new constraints).

4.4 Examples

To demonstrate the precision of the type system and inference analysis, we illustrate the handling of one example which has posed challenges for previous taint analyses [15, 30].

The example, shown in Fig. 10 illustrates the handling of context sensitivity. There are two instances of DataSource, one that holds a tainted string in its f field, and another one that holds a safe string. The code is safe because s2, which flows to the sensitive sink, is read from the "safe" DataSource object. A context-insensitive taint analysis would merge the flows through setUrl and getUrl across the two different instances of DataSource, and report a spurious warning.

Fig. 10 illustrates our solution. The inferred typing types class DataSource as polymorphic. The poly types are instantiated to tainted for object ds1 and to safe for object ds2.

class DataSource {		
String f;		
void setUrl(String url) {		S(this$_{\mathsf{setUrl}}$) = {~~tainted~~, **poly**}
this.f = url;	url <: this$_{\mathsf{setUrl}}$ ▷ f	S(url) = {~~tainted~~, **poly**}
}	url <: this$_{\mathsf{setUrl}}$	
String getUrl() {		S(this$_{\mathsf{getUrl}}$) = {~~tainted~~, **poly**}
return this.f;	this$_{\mathsf{getUrl}}$ ▷ f <: ret$_{\mathsf{getUrl}}$	S(f) = {**poly**}
}	this$_{\mathsf{getUrl}}$ <: ret$_{\mathsf{getUrl}}$	S(ret$_{\mathsf{getUrl}}$) = {**poly**, safe}
}		
String tUrl = req.getParameter(..);	req ▷ **tainted** <: tUrl	S(tUrl) = {**tainted**}
DataSource ds1 = new DataSource();		
ds1.setUrl(tUrl);	tUrl <: ds1 ▷ url	
	ds1 = ds1 ▷ this$_{\mathsf{setUrl}}$	S(ds1) = {**tainted**, ~~poly~~, ~~safe~~}
String sUrl = "http://localhost/";		S(sUrl) = {~~tainted~~, ~~poly~~, **safe**}
DataSource ds2 = new DataSource();		
ds2.setUrl(sUrl);	sUrl <: ds2 ▷ url	
	ds2 = ds2 ▷ this$_{\mathsf{setUrl}}$	S(ds2) = {~~tainted~~, ~~poly~~, **safe**}
String s1 = ds1.getUrl();	ds1 <: ds1 ▷ this$_{\mathsf{getUrl}}$	
	ds1 ▷ ret$_{\mathsf{getUrl}}$ <: s1	S(s1) = {**tainted**, ~~poly~~, ~~safe~~}
	ds1 <: s1	
String s2 = ds2.getUrl();	ds2 <: ds2 ▷ this$_{\mathsf{getUrl}}$	
	ds2 ▷ ret$_{\mathsf{getUrl}}$ <: s2	
	ds2 <: s2	
writer.println(s2);	s2 <: writer ▷ **safe**	S(s2) = {**safe**}

Fig. 10. The DataSource example due to Ben Livshits [15]. The frame box beside each statement shows the generated constraints correspondingly. The bold frame boxes show the constraints generated by the algorithm in Fig. 8. The oval boxes show the set-based solution, where overstruck qualifiers are eliminated by the the algorithm in Fig. 8. The bold qualifiers are the final maximal typing. It type checks.

As illustrated, the analysis handles naturally these difficult idioms. The handling of DataSource can be interpreted as object sensitivity [19]: essentially, the analysis processes polymorphic setUrl and getUrl separately for object contexts ds1 and ds2, just as standard object-sensitive analysis does.

4.5 Precision Improvements

We employ two techniques to improve the precision of SFlow and SFlowInfer. One is the composition with ReIm we described earlier. The other one is special-casing of global mapping data structures Properties from the java.util package, and ServletRequest and HttpSession from the javax.servlet package. In order to illustrate the problem, consider the example in Fig. 11 refactored from benchmark blojsom. At line 6, the tainted inAuthor is put into the mapping of req. Then it is

```
1  class BlojsomServlet {
2    public static final String AUTHOR = "BLOJSOM_AUTHOR";
3    public void doGet(HttpServletRequest req, HttpServletResponse resp) {
4      String inAuthor = req.getParameter(''author''); // tainted source
5      req.setAttribute(BLOJSOM_AUTHOR, inAuthor);
6    }
7  }
8  class html_dcomments_jsp {
9    public void _jspService(HttpServletRequest req, HttpServletResponse resp) {
10     String outAuthor = (String) req.getAttribute(BlojsomServlet.AUTHOR);
11     PrintWriter out = ...;
12     out.print(outAuthor);       // safe sink
13   }
14 }
```

Fig. 11. Imprecision caused by mapping data structures.

retrieved at line 13 through req.getAttribute() and printed to the client page. The parameter of PrintWriter.print() is a safe sink according to [15]. Therefore, there is unsafe flow from req.getParameter() to out.print().

If outAuthor = req.getAttribute(...) were handled according to the typing rules in Fig. 4, the safe outAuthor would cause req to be safe, and safe would propagate to all calls on receiver req, not only to the call with argument req.setAttribute(...,inAuthor).

Therefore, we special-case set$*$ and get$*$ methods for such mapping data structures, similarly to Sridharan et al. [28]. If the key of the set$*$ method call set(key, value) is a constant, the inference simply creates the equality constraint key = value. Similarly, if the key of get$*$ method call x = get(key) is a constant, the set-based solver creates constraint x = key. For the example in Fig. 11, the set-based solver enforces BlojsomServlet.BLOJSOM_AUTHOR = inAuthor at line 5 and outAuthor = BlojsomServlet.BLOJSOM_AUTHOR at line 10. Thus, inAuthor and outAuthor are connected and outAuthor is typed as tainted. The unsafe information flow is detected because there is a type error when passing tainted outAuthor to the safe parameter of out.print().

5 Handling of Reflection, Libraries and Frameworks

Reflection, libraries (standard and third-party) and frameworks (e.g., Struts, Spring, Hibernate) are the bane of static taint analysis. Yet they are ubiquitous in Java web applications. The type-based approach we espouse, handles these features safely and effortlessly.

5.1 Reflection

Use of reflection in web application code is widespread. Therefore, ignoring reflection (as many static analyses do) renders a static analysis useless. Consider the example:

```
X x = Class.forName("someInput").newInstance();
x.f = a;      // a is tainted, comes from source
y = x;
b = y.f;      // b is safe, flows to sink
```

If a points-to-based static analysis fails to handle newInstance(), the points-to sets of x and y will be empty, and the flow from a to b will be missed. On the other hand, handling of reflection is notoriously difficult and generally unsound.

We handle newInstance() safely and effortlessly. The key is that SFlow *does not need to abstract heap objects*; instead, it tracks dependences between variables through subtyping. It can be shown that, roughly speaking, if x flows to y, then x <: y holds. In the above example, x <: y and subsequently a <: b holds. SFlowInfer reports a type error because of the flow from tainted a to safe b.

5.2 Libraries

Our inference analysis is modular. Thus, it can analyze any given set of classes L. If there is an unknown callee in L, e.g. a library method whose source code is unavailable, the analysis assumes typing $\mathsf{poly}, \mathsf{poly} \rightarrow \mathsf{poly}$ for the callee. This typing conservatively propagates tainted arguments to the receiver and left-hand-side of the call assignment. Similarly, it propagates a safe left-hand-side to the receiver and arguments at the call. E.g., String.toUpperCase() is typed as

poly String toUpperCase(**poly** String this)

At call s2 = s1.toUpperCase() we have constraint $\mathsf{s1} \triangleright \mathsf{poly} <: \mathsf{s2}$ or equivalently $\mathsf{s1} <: \mathsf{s2}$. Thus, a tainted s1 propagates to s2, and a safe s2 propagates to s1.

We apply the $\mathsf{poly}, \mathsf{poly} \rightarrow \mathsf{poly}$ typing to all methods in the standard library, third-party libraries (e.g., apache-tomcat, xalan) and frameworks, with several exceptions described in the next section.

5.3 Frameworks

Most Java web applications are built on top of one or more *web application frameworks* such as Struts, Spring, Hibernate, and etc. The problem with these frameworks is twofold. First, these frameworks contain "hidden" sources and sinks, i.e., sources and sinks deep in framework code that affect the public API. For example, Hibernate (version 2.1) contains a public method Session.find(String s), where s flows to query at sink prepareStatement(query). This happens deep in the code of Hibernate. We run a version of our inference analysis and "lift" such hidden sources and sinks to the return values and parameters of the public methods they affect. In the above example, Session.find() is typed as

poly List find(**poly** Session this, **safe** String s)

Callers to find() in application code must handle the argument of find() as safe. To the best of our knowledge, no other taint analysis attempts to "lift" these "hidden" sources and sinks in the frameworks.

Second, these frameworks rely heavily on reflection and callbacks, which must be handled in the analysis. These are notoriously issues for dataflow and points-to based analysis, which usually relies on reachability analysis. Our type-based analysis handles these features safely and effortlessly through the method overriding constraints.

As an illustrating example, Struts defines framework classes ActionForm and Action and method Action.execute(ActionForm form). The application built on top of Struts defines numerous xxxForm classes extending ActionForm, and numerous xxxAction classes extending Action. Framework code performs the following (roughly):

1. Action a = Class.forName("inputClass").newInstance(); a instantiates one user-defined xxxAction class.
2. ActionForm f = Class.forName("inputForm").newInstance(); similarly, this instantiates one user-defined xxxForm class.
3. Framework populates the xxxForm object with *tainted* values from sources.
4. Framework calls a.execute(f), a callback to user-defined xxxAction.execute.

In our type-based analysis Action.execute is typed as

execute(**poly** Action this, **tainted** ActionForm form)

The method overriding constraints (recall Sect. 2.3) propagate tainted to the form parameter of each execute method in user-defined subclasses. As a result, all values retrieved through get methods from forms in user code are tainted, which accurately reflects that the xxxForm object is populated with tainted values.

6 Empirical Results

SFlow and SFlowInfer are implemented within our type inference framework [13, 14], which is built on top of the Checker Framework (CF) [23]. The type inference framework, including SFlow and SFlowInfer, is publicly available at `http://code.google.com/p/type-inference/`.

The implementation is evaluated on 13 relatively large Java web applications, used in previous work [15, 28, 31]. We run SFlowInfer on these benchmarks on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (the maximal heap size is set to 2 GB). The software environment consists of Oracle JDK 1.6 and the Checker Framework 1.1.5 on GNU/Linux 3.2.0.

6.1 Experiments

We use the sources and sinks described in detail in Livshits and Lam [15, 16]. In addition, we use 59 sources and sinks in API methods of Struts, Spring, and

Benchmark	Version	#File	#Line	Time (s)
blojsom	1.9.6	61	12830	15.1
blueblog	1.0	31	4139	7.5
friki	2.1.1	21	1843	4.5
gestcv	1.0	119	7422	10.1
jboard	0.3	89	17405	22.2
jspwiki	2.4	364	83329	126.9
jugjobs	alpha	25	4044	18.7
pebble	1.6beta1	234	42542	50.3
personalblog	1.2.6	68	9943	17.6
photov	2.1	129	126886	640.2
roller	0.9.9	276	81171	213.4
snipsnap	1.0beta	488	73295	87.3
webgoat	0.9	35	8474	9.6

Fig. 12. Information about benchmarks and running time of SFlowInfer. The file and line counts include Java files precompiled from JSP files. The time is for running configuration [*Parameter manipulation, SQL injection*]. The time for running other configurations is practically the same.

	[*Parameter*,*SQL*]			[*Parameter*,*XSS*]			[*Parameter*,*HTTP*]			[*Parameter*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	1	0	0 (0%)	10	1	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	3	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	1	0	9 (90%)	8	1	0 (0%)
gestcv	1	0	0 (0%)	0	8	2 (20%)	0	0	0 (0%)	1	0	0 (0%)
jboard	3	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	25 (100%)	73	12	20 (19%)	23	0	16 (34%)	72	0	23 (24%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	2	0	0 (0%)	4	0	3 (37%)	43	3	0 (0%)
personalblog	6	0	0 (0%)	3	21	2 (8%)	0	0	0 (0%)	0	0	0 (0%)
photov	46	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	21	2	0 (0%)	1	2	1 (25%)	0	5	19 (79%)
snipsnap	0	0	3 (100%)	1	0	0 (0%)	6	0	0 (0%)	8	26	13 (28%)
webgoat	10	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	1	0	4 (80%)
Average			(**15%**)			(**4%**)			(**14%**)			(**16%**)

Fig. 13. Inference results for [*Parameter*, *SQL*], [*Parameter*, *XSS*], [*Parameter*, *HTTP*] and [*Parameter*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations; note that a large number of benchmarks have 0 type errors, i.e., they are proven safe.

Hibernate, discovered as described in Sect. 5. There are 3 categories of sources [15]: *Parameter manipulation*, *Header manipulation*, and *Cookie poisoning*. There are 4 categories of sinks [15]: *SQL injection*, *HTTP splitting*, *Cross-site scripting (XSS)*, and *Path traversal*. These sources and sinks are added to the annotated JDK, Struts, Spring, and Hibernate, which is easily done with the CF. Once these annotated libraries are created, individual web applications are analyzed without any input from the user. We run the benchmarks with all 12 configurations.

Fig. 12 presents the sizes of the benchmarks as well as the running times of SFlowInfer in seconds. The running times attest to efficiency — for all but 1 benchmark, the analysis completes in less than 4 minutes; we strongly believe that these running times can be improved.

We examined the type errors reported by SFlowInfer, and classified them as Type-1 (**T1**), Type-2 (**T2**), or False-positive (**FP**). Type-1 errors reflect direct

	[*Header*,*SQL*]			[*Header*,*XSS*]			[*Header*,*HTTP*]			[*Header*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	3 (100%)
gestcv	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jboard	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	53? (100%)	0	0	113? (100%)	0	0	50? (100%)	0	0	154? (100%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
personalblog	1	0	0 (0%)	0	16	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
photov	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	1	0	0 (0%)	1	0	0 (0%)	0	0	0 (0%)
snipsnap	0	0	0 (0%)	7	0	0 (0%)	2	0	0 (0%)	0	25	54 (68%)
webgoat	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
Average			(**8%**)			(**8%**)			(**8%**)			(**21%**)

Fig. 14. Inference results for [*Header*, *SQL*], [*Header*, *XSS*], [*Header*, *HTTP*] and [*Header*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations. Again a large number of benchmarks have 0 type errors, i.e., they are proven safe. Due to time constraints, we did not examine the type errors for jspwiki; instead, we conservatively classified them as False-positive. Therefore, the actual False-positive rate is lower than the one reported.

	[*Cookie*,*SQL*]			[*Cookie*,*XSS*]			[*Cookie*,*HTTP*]			[*Cookie*,*Path*]		
Benchmark	T1	T2	FP	T1	T2	FP	T1	T2	FP	T1	T2	FP
blojsom	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
blueblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
friki	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
gestcv	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jboard	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
jspwiki	0	0	53? (100%)	0	0	172? (100%)	0	0	50? (100%)	0	0	155? (100%)
jugjobs	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
pebble	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
personalblog	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
photov	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
roller	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	1 (100%)
snipsnap	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	19	8 (30%)
webgoat	1	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)	0	0	0 (0%)
Average			(**8%**)			(**8%**)			(**8%**)			(**18%**)

Fig. 15. Inference results for [*Cookie*, *SQL*], [*Cookie*, *XSS*], [*Cookie*, *HTTP*] and [*Cookie*, *Path*]. The multicolumns show numbers of Type-1 (**T1**), Type-2 (**T2**), and False-positive (**FP**) type errors for the four configurations. Again, we conservatively classified all errors in jspwiki as False-positive and the actual False-positive rate is lower than the one reported.

flow from a source to a sink. The following code, adapted from webgoat, is a Type-1 error for configuration *[Parameter,SQL]*:

```
String u = request.getParameter("user"); //source
String s = "SELECT * FROM users WHERE name = " + u;
stat.executeQuery(s); //sink, type error!
```

Another example of a Type-1 error, adapted from benchmark blueblog, is shown below. This is a type error for configuration *[Parameter,Path]*. This example illustrates a complex flow that goes through heap objects and method calls. It attests to the power of our analysis.

```
1  class BBServlet {
2      ...
3      String title = request.getParameter("title"); //source
4      String content = ...
5      BlogData bd = new BlogData(title, content);
6      currentCategory.addNewBlog(bd);
7      ...
8  }
9  class FSCategory extends Category {
10    ...
11    Blog addNewBlog(BlogData bd) {
12      ...
13      return FSBlog.createNewBlog(...,bd,...);
14    }
15 }
16 class FSBlog extends Blog {
17    static FSBlog createNewBlog(...,BlogData blogData,...) {
18      String filename = blogData.getSuggestedId(); //type error!
19      File file = new File(filename+fileEndings); //sink
20      ...
21    }
22 }
23 class BlogData {
24    String title;
25    String suggestedId;
26    BlogData(String title, String content) {
27      this.title = title;
28      this.suggestedId = constructSuggestedId(title);
29      ...
30    }
31 }
```

Observe the complex flow from the source at line 3 to the sink at line 19. The servlet creates a new BlogData object, and passes the tainted title to it. Fields title and suggestedId of the BlogData object store tainted values. The BlogData object is then passed as argument to addNewBlog in FSCategory (line 6) and then to createNewBlog in FSBlog (line 13). createNewBlog reads the suggestedId field of the BlogData object and sends it to the sink. SFlowInfer reports a type error at line 18.

Type-2 errors reflect key-value dependences. The following code, adapted from personalblog, is a Type-2 error for configuration *[Parameter,XSS]*:

```
HashMap map = ...; PrintWriter out = ...;
String id = request.getParameter(''id''); //source
User user = (User) map.get(id);
out.print(user.getName()); //sink, type error!
```

The tainted id is used as a key to retrieve the user from the map, then user.getName() is sent to a safe sink (the parameter of PrintWriter.print). This is a dangerous flow according to the semantics of noninterference, because the tainted value of the key affects the value of the safe sink.

We classified as **FP** all errors that we could not easily identify as Type-1 or Type-2. The results over the 12 configurations are presented in Fig. 13, Fig. 14 and Fig. 15.

6.2 Comparison

Direct comparison with TAJ [31], F4F [28], and ANDROMEDA [30] is impossible because the analysis tools are proprietary, and therefore unavailable. Instead, we run SFlowInfer on DroidBench [8], which is a suit of Android apps, and compare with three other taint analysis tools – AppScan Source [2], Fortify SCA [1], and FlowDroid [8], using the results presented by Fritz et al. [8]. The comparison with AppScan Source is an indirect comparison with TAJ, F4F, and ANDROMEDA, because these analyses are built into AppScan Source.

Fig. 16 presents the result of the comparison. Although SFlowInfer performs slightly worse in terms of precision (due to the conservativeness of the type system), it outperforms all other tools in terms of recall, i.e. it detects more vulnerabilities than all other tools. Commercial tools AppScan Source and Fortify SCA detect less than 61% of all vulnerabilities, while SFlowInfer detects 100%. FlowDroid, which targets Android apps, not Java web applications, is more precise than SFlowInfer. This is because it uses a flow-sensitive analysis, which unfortunately can be costly.

7 Related Work

There is a large amount of work on information flow control. Unfortunately, we cannot include all related work on information flow control.

The most closely related to ours is the work by Shankar et al. [26]. They present a type system for detecting string format vulnerabilities in C programs. The type system has two type qualifiers, tainted and untainted; polymorphism is not part of the core system. They include a type inference engine built on top of CQual [7]. CQual, and its counterpart for Java JQual [10] rely on dependence graphs built using points-to analysis. Thus, they still face the burden of reflection and frameworks. In order to handle polymorphism, they provide the *polymorphic* function as an extension. In contrast, SFlow and SFlowInfer handle polymorphism naturally, as it is built into the type system using the poly qualifier and viewpoint adaptation. In addition, we compose with reference immutability, thus improving precision significantly. SFlow and SFlowInfer handle reflection and frameworks seamlessly.

Tripp et al. [31] present TAJ, a points-to-based taint analysis for industrial applications. TAJ is a dataflow and points-to-based analysis. In contrast, our type-based taint analysis is modular and compositional. In order to handle Struts, TAJ

Tool Name	**AppScan Source**	**Fortify SCA**	**FlowDroid**	**SFlowInfer**
Arrays and Lists				
ArrayAccess1			$\times$	$\times$
ArrayAccess2	$\times$	$\times$	$\times$	$\times$
ListAccess1	$\times$	$\times$	$\times$	$\times$
Callbacks				
AnonymousClass1	$\bigcirc$	$\surd$	$\surd$	$\surd$
Button1	$\bigcirc$	$\surd$	$\surd$	$\surd$
Button2	$\surd\bigcirc\bigcirc$	$\surd\bigcirc\bigcirc$	$\surd\surd\surd\times$	$\surd\surd\surd\times$
LocationLeak1	$\bigcirc\bigcirc$	$\bigcirc\bigcirc$	$\surd\surd$	$\surd\surd$
LocationLeak2	$\bigcirc\bigcirc$	$\bigcirc\bigcirc$	$\surd\surd$	$\surd\surd$
MethodOverride1	$\surd$	$\surd$	$\surd$	$\surd$
Field and Object Sensitivity				
FieldSensitivity1				
FieldSensitivity2				
FieldSensitivity3	$\surd$	$\surd$	$\surd$	$\surd$
FieldSensitivity4	$\times$			$\times$
InheritedObjects1	$\surd$	$\surd$	$\surd$	$\surd$
ObjectSensitivity1				
ObjectSensitivity2	$\times$			$\times\times$
Inter-App Communication				
IntentSink1	$\surd$	$\surd$	$\bigcirc$	$\surd$
IntentSink2	$\surd$	$\surd$	$\surd$	$\surd$
ActivityCommunication1	$\surd$	$\surd$	$\surd$	$\surd$
Lifecycle				
BroadcastReceiverLifecycle1	$\surd$	$\surd$	$\surd$	$\surd$
ActivityLifecycle1	$\surd$	$\surd$	$\surd$	$\surd$
ActivityLifecycle2	$\bigcirc$	$\surd$	$\surd$	$\surd$
ActivityLifecycle3	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$
ActivityLifecycle4	$\bigcirc$	$\surd$	$\surd$	$\surd$
ServiceLifecycle1	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$
General Java				
Loop1	$\surd$	$\bigcirc$	$\surd$	$\surd$
Loop2	$\surd$	$\bigcirc$	$\surd$	$\surd$
SourceCodeSpecific1	$\surd$	$\surd$	$\surd$	$\surd$
StaticInitialization1	$\bigcirc$	$\surd$	$\bigcirc$	$\surd$
UnreachableCode		$\times$		$\times$
Miscellaneous Android-Specific				
PrivateDataLeak1	$\bigcirc$	$\bigcirc$	$\surd$	$\surd$
PrivateDataLeak2	$\surd$	$\surd$	$\surd$	$\surd$
DirectLeak1	$\surd$	$\surd$	$\surd$	$\surd$
InactiveActivity	$\times$	$\times$		$\times$
LogNoLeak				
Sum, Precision and Recall–excluding implicit flows				
$\surd$, higher is better	14	17	26	28
$\times$, lower is better	5	4	4	9
$\bigcirc$, lower is better	14	11	2	0
Precision $p = \surd/(\surd + \times)$	74%	81%	86%	76%
Recall $r = \surd/(\surd + \bigcirc)$	50%	61%	93%	100%
F-measure $2pr/(p + r)$	0.60	0.70	0.89	0.86

Fig. 16. Summary of comparison with other taint analysis tools ($\surd$ = correct warning, $\times$ = false warning, $\bigcirc$ = missed flow). multiple circles in one row: multiple leaks expected, all-empty row: no leaks expected, none reported.

treats all Action classes as entry points. In addition, it simulates the passing of all subclasses of ActionForm to Action.execute, by generating a constructor, which assigns tainted values to all fields of the subclasses. In contrast, our inference analysis handles Struts by annotating the ActionForm parameter of Action.execute as tainted. Our handling is simpler and equally precise. Finally, TAJ approximates the behavior of Java reflection APIs by synthesizing an abstract object whenever the instantiated class can be inferred. It is unclear how TAJ handles reflection when the instantiated class cannot be inferred (e.g. the argument is not a string constant). according to Sridharan et al. [28], TAJ's reflection modeling is not scalable. In contrast, our type-based analysis does not need abstract objects, and handles reflection seamlessly and safely.

Livshits and Lam [15] present a static analysis based on a scalable and precise points-to analysis. The analysis is built on top of a context-sensitive Java points-to analysis [34] based on Binary Decision Diagrams (BDDs). In contrast, our inference analysis is type-based and modular. In order to handle reflection, they look for all calls to Class.forName(s) that may return className, then find all constant strings that s may refer to, and finally augment the call graph by adding an edge from the call site of newInstance to new S(), which is represented by s. Similarly to TAJ, they handle reflection by trying to infer the value of string s at forName(s).newInstance() calls. In addition, Livshits and Lam's analysis does not handle frameworks, which are essential for web applications.

Sridharan et al. [28] present F4F, a system for taint analysis of framework-based web applications. In order to handle frameworks, F4F analyzes the application code and XML configuration files to construct a specification, which summarizes reflection and callback-driven behavior. In contrast, our analysis handles frameworks by inferring or adding annotations to sources and sinks in the frameworks, which propagate to user code through subtyping. Tripp et al. [30] present ANDROMEDA, a demand-driven analysis that improves on F4F.

Very recent work by Fritz et al. present FlowDroid, a taint analysis for Android [8]. The analysis is dataflow and points-to-based; also, it focuses on Android apps. Our analysis is type-based and focuses on Java web applications.

Volpano et al. [33] and Myers [20] present type systems for secure information flow. These systems are substantially more complex than SFlow. They focus on type checking and do not include type inference or include only local (intra-procedural) type inference. In contrast, SFlowInfer handles large web applications.

Snelting et al. [9,11,12,27] present information flow analysis based on Program Dependence Graphs (PDGs). Their analysis relies on highly precise context-sensitive dataflow and points-to analysis.

8 Conclusions

We have presented SFlow, a context-sensitive type system for secure information flow, and SFlowInfer, the corresponding cubic inference analysis. Our approach handled reflection, libraries and frameworks safely and effectively. Experiments on 13 Java web applications showed that SFlowInfer is scalable and precise.

References

1. HP fortify static code analyzer. `http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#.Uk4YZWRhsyk`, 2013.

2. IBM security AppScan. `http://www-03.ibm.com/software/products/us/en/appscan/`, 2013.
3. L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, DIKU, University of Copenhagen, 1994.
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. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):1–48, Apr. 2012.
7. J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, May 1999.
8. 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.
9. D. Giffhorn and C. Hammer. Precise analysis of java programs using joana. In *SCAM*, pages 267–268, 2008.
10. D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, 2007.
11. C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *ISoLA*, pages 119–128, 2006.
12. C. Hammer, R. Schaade, and G. Snelting. Static path conditions for java. In *PLAS*, pages 57–66, 2008.
13. W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.
14. 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.
15. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security*, 2005.
16. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. Technical Report. Stanford University. `http://suif.stanford.edu/~livshits/papers/tr/webappsec_tr.pdf`, 2005.
17. A. Milanova and W. Huang. Static object race detection. In *APLAS*, pages 255–271, 2011.
18. A. Milanova and W. Huang. Dataflow and type-based formulations for reference immutability. In *FTfJP*, 2013.
19. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.
20. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
21. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.
22. OWASP. Top ten project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2013.
23. M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.

24. A. Sampson, W. Dietl, and E. Fortuna. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.
25. A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation Full Proofs. `http://homes.cs.washington.edu/~asampson/files/enerjproofs.pdf`, 2011.
26. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.
27. 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.
28. 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.
29. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.
30. 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.
31. 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.
32. M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.
33. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, pages 167–187, 1996.
34. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.