Type-based Taint Analysis for Java Web Applications

Wei Huang, Yao Dong, and Ana Milanova

Rensselaer Polytechnic Institute

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 [14]. A common information flow vulnerability is SQL injection, shown in the example in Fig. 1 (adapted from [9]).

```
1 HttpServletRequest request = ...;
2 Statement stat = ...;
3 String user = request.getParameter(''user'');
4 StringBuffer sb = ...;
5 sb.append("SELECT * FROM Users WHERE name = ");
6 sb.append(user);
7 String query = sb.toString();
8 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 (XXS), HTTP response splitting, path traversal and command injection [9].

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 [5, 9, 18–20]. 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 [6], which we have used to infer and check object ownership [6] and reference immutability [8].

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*. The modularity of the inference allows for the effective handling of libraries and frameworks. Our approach handles reflective object creation as well. This is possible because SFlow does not require abstraction of heap objects; instead, 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 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 reflective object creation, 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 and Sect. 3 describes the inference analysis. Sect. 4 describes techniques for handling of reflection, libraries and frameworks. Sect. 5 presents the empirical evaluations. Sect. 6 discusses the related work, and Sect. 7 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]:

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

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:

```
2  ...
3  tainted String user = request.getParameter(“user”);
4  tainted StringBuffer sb = ...;    // it includes the tainted user
5  sb.append("SELECT * FROM Users WHERE name = ");
6  sb.append(user);
7  tainted String query = sb.toString();
8  stat.executeQuery(query);         // type error!
```

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 by any input, but it is *typed* tainted because the return value of toString is of type tainted. 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*.

[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.

– 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 [3]. Viewpoint adaptation of a type q' from the viewpoint of another type q, results in the adapted type q''. This is written as $q \triangleright q' = q''$. 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:

$$\_ \triangleright \mathsf{tainted} = \mathsf{tainted} \qquad \_ \triangleright \mathsf{safe} = \mathsf{safe} \qquad \mathsf{q} \triangleright \mathsf{poly} = \mathsf{q}$$

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

Fig. 3 shows the typing rules over a syntax in "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. The SFlow type system is *orthogonal*

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\textsc{tnew}) \qquad \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{x}} <: q_{\mathsf{y}} \rhd q_{\mathsf{f}}}{\Gamma \vdash \mathsf{y.f} = \mathsf{x}}\ (\textsc{twrite})$$

$$\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\textsc{tassign}) \qquad \frac{\Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\textsc{tread})$$

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

Fig. 3. Typing rules. Function *typeof* retrieves the SFlow types of fields and methods, Γ is a type environment that maps variables to SFlow qualifiers.

to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

The rules create subtyping constraints at explicit assignments (e.g., $\mathsf{x} = \mathsf{y}$, $\mathsf{x} = \mathsf{y.f}$) 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) {...}

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}):

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

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:

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

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,

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

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, 5, 9, 18–20] 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 = ...;
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 [13].

The standard solution is to disallow subtyping for references [16]. 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. 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 [11].

We propose a solution using reference immutability, which allows limited subtyping and improves precision. It is a theorem that subtyping is safe when the reference on the left-hand-side of the assignment 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 [8]. We run ReImInfer [8], 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) x = y.f, if x is readonly, we use constraint $q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}}$; otherwise, we use constraint $q_{\mathsf{y}} \rhd q_{\mathsf{f}} = q_{\mathsf{x}}$. Due to space constraints, we do not describe the details of the type system. The dynamic semantics and soundness proof can be found in the accompanying technical report [7]. This composition approach achieves at least 20% precision improvement over the standard approach as shown in our previous work [11].

3 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. 3. If inference succeeds, then the program is safe, i.e., it is guaranteed that there is no flow from a source to a sink. If it fails, then a valid typing does not exist, meaning that there could be unsafe flow from a source to a sink.

Type inference leverages the framework we developed in [6]. It first computes a *set-based solution* S, which maps variables to *sets* of potential type qualifiers.

The key novelty over [6] is the use of *method summary constraints*, which refine the set-based solution, and help derive a valid typing.

3.1 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. 3. 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 SolveConstraint(c) for each constraint c. SolveConstraint(c) removes infeasible qualifiers from the set of variables that participate in c. It works as follows (for a more formal description, see [6]). 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 $q_\mathsf{y} \rhd \mathsf{tainted} <: q_\mathsf{x}$. 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 $S(\mathsf{x})$, 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 $\mathsf{x}\{\} <: \mathsf{y}\{\mathsf{safe}\}$).

The set-based solver iterates over the constraints and refines the sets until it reaches a fixpoint. There are two possible 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. 4. foo is safe when b1 and b2 refer to distinct StringBuffer

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

```
1  void doGet(A this, ServletRequest request, ServletResponse response) {
2    StringBuffer buf = ...;
3    this.foo(buf,buf,request,response);
4  }
5  void foo(A this, StringBuffer b1, StringBuffer b2,
6           ServletRequest req, ServletResponse resp) {
7    String url = req.getParameter("url");
8    b1.append(url);
9    String str = b2.toString();
10   PrintWriter writer = resp.getWriter();
11   writer.print(str);
12 }
```

$\text{buf} = \text{this}_{\text{doGet}} \triangleright \text{b1}$	$S(\text{buf}) = \{\text{tainted}\}$
$\text{buf} <: \text{this}_{\text{doGet}} \triangleright \text{b2}$	$S(\text{b2}) = \{\text{tainted}, \text{poly}\}$
$\text{req} \triangleright \textbf{tainted} <: \text{url}$	$S(\text{url}) = \{\text{tainted}\}$
$\text{url} <: \text{b1} \triangleright \textbf{poly}$	$S(\text{b1}) = \{\text{tainted}\}$
$\text{b2} \triangleright \textbf{poly} <: \text{str}$	$S(\text{str}) = \{\text{tainted}, \text{poly}\}$
$\text{str} <: \text{writer} \triangleright \textbf{safe}$	TYPE ERROR!

Fig. 4. 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).

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 $\{\text{tainted}, \text{poly}\}$.

3.2 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 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

```
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. 5. Computation of method summary constraints. C is the set of constraints, it is initialized to the set of constraints for program statements, derived as described in Sect. 3.1 (recall that each equality constraint is written as two subtyping constraints). S is initialized to the result of the set-based solver. Cases 1 and 2 add $q_x <: q_y$ into C because $q_y \triangleright \mathsf{poly}$ 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 $q_w <: q_a \triangleright q_x$ (flow from actual to formal) and $q_a \triangleright q_y <: q_z$ (flow from return value to left-hand-side), and also $q_x <: q_y$ (flow from formal to return value, usually discovered by Case 3), Case 4 adds $q_w <: q_z$. Note that line 4 calls SolveConstraint(c): the solver infers new constraints, which remove additional infeasible qualifiers from S. This process repeats until S stays unchanged.

is presented in Fig. 5. As an example, consider the following code snippet:

```
class A {
  String f;
  String get()
  {return this.f;}  this ▷ f <: ret
}

A y = ...;
PrintWriter writer = ...;
String x = y.get();  y <: y ▷ this   y ▷ ret <: x
writer.print(x);     x <: writer ▷ 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. 5 creates $\mathsf{this} <: \mathsf{ret}$. This entails $\mathsf{y} \triangleright \mathsf{this} <: \mathsf{y} \triangleright \mathsf{ret}$ since viewpoint adaptation preserves subtyping [11]. Case 3 combines this with constraints $\mathsf{y} <: \mathsf{y} \triangleright \mathsf{this}$ and $\mathsf{y} \triangleright \mathsf{ret} <: \mathsf{x}$, 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) with type errors, or (2) without 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* almost 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 benchmark, jugjobs. It is a theorem that even if it does not type-check, the program is still safe, i.e., there is no flow from sources to sinks. We confirmed this for the 2 constraints in jugjobs.

The inference is dominated by the algorithm in Fig. 5, which has worst-case complexity of $O(n^3)$, where n is the size of the program (see [7] for details).

4 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.

Reflective object creation Use of reflective object creation in web applications is widespread. Ignoring it, as some static analyses do, renders a static analysis useless. Consider the use of newInstance():

```
X 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 reflective object creation is difficult, expensive and often unsound.

We handle reflective object creation with newInstance() safely and effortlessly. The key is that SFlow tracks dependences between variables through subtyping, which *obviates the need to abstract heap objects*. It can be shown that, roughly speaking, if x flows to y, then x <: y holds. In the above example, x <: y holds and subsequently a <: b holds. SFlowInfer reports a type error caused by the flow from tainted a to safe b.

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 $\mathsf{poly}, \mathsf{poly} \to \mathsf{poly}$ 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 s1 <: s2. Thus, a tainted s1 propagates to s2, and a safe s2 propagates to s1.

We apply the $\mathsf{poly}, \mathsf{poly} \to \mathsf{poly}$ 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.

Frameworks Most Java web applications are built on top of one or more *frameworks* such as Struts, Spring, Hibernate, and etc. The problem with these frameworks is twofold. First, they 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, otherwise it may lead to an SQL injection vulnerability as described by Livshits and Lam [9]. 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 notorious issues for dataflow and points-to based analysis, which usually relies on reachability analysis. Our type-based analysis handles these features with 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 = (Action) Class.forName("inputClass").newInstance(); a instantiates one user-defined xxxAction class.
2. ActionForm f = (ActionForm) 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.

5 Empirical Results

SFlow and SFlowInfer are implemented within our type inference framework [6,8], which is built on top of the Checker Framework (CF) [15]. 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 [9, 18, 20]. We run SFlowInfer on these benchmarks on a

			[*Parameter*,*SQL*]			[*Parameter*,*XSS*]		
Benchmark	#Line	Time (s)	Type-1	Type-2	FP	Type-1	Type-2	FP
blojsom	12830	15.1	0	0	0 (0%)	0	0	0 (0%)
blueblog	4139	7.5	0	0	0 (0%)	0	0	0 (0%)
friki	1843	4.5	0	0	0 (0%)	0	0	0 (0%)
gestcv	7422	10.1	1	0	0 (0%)	0	8	2 (20%)
jboard	17405	22.2	3	0	0 (0%)	0	0	0 (0%)
jspwiki	83329	126.9	0	0	25 (100%)	73	12	20 (19%)
jugjobs	4044	18.7	0	0	0 (0%)	0	0	0 (0%)
pebble	42542	50.3	0	0	0 (0%)	2	0	0 (0%)
personalblog	9943	17.6	6	0	0 (0%)	3	21	2 (8%)
photov	126886	640.2	46	0	0 (0%)	0	0	0 (0%)
roller	81171	213.4	0	0	0 (0%)	21	2	0 (0%)
snipsnap	73295	87.3	0	0	3 (100%)	1	0	0 (0%)
webgoat	8474	9.6	10	0	0 (0%)	0	0	0 (0%)
Average					(**15%**)			(**4%**)

Fig. 6. Inference results for [*Parameter*, *SQL*] and [*Parameter*, *XSS*]. **Time** shows the running times of SFlowInfer for [*Parameter*, *SQL*] in seconds; running times for other configurations are essentially the same. The multicolumns show numbers of **Type-1**, **Type-2**, and False-positive (**FP**) type errors for the two configurations; note that a large number of benchmarks have 0 type errors, i.e., they are proven safe.

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.

Experiments We use the sources and sinks described in detail in Livshits and Lam [9, 10]. In addition, we use 59 sources and sinks in API methods of Struts, Spring, and Hibernate, discovered as described in Sect. 4. There are 3 categories of sources [9]: *Parameter manipulation*, *Header manipulation*, and *Cookie poisoning*. There are 4 categories of sinks [9]: *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. However, for space reasons, we report only on 2 configurations: [*Parameter manipulation*, *SQL*] and [*Parameter manipulation*, *XSS*].

Fig. 6 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 believe that these running times can be improved.

We examined the type errors reported by SFlowInfer, and classified them as **Type-1**, **Type-2**, or False-positive (**FP**). **Type-1** errors reflect direct flow from a source to a sink. The following code, adapted from webgoat, is a Type-1 error:

```
String u = request.getParameter(“user”);
String s = “SELECT * FROM users WHERE name = ’’ + u;
stat.executeQuery(s);
```

Tool Name	**AppScan Source**	**Fortify SCA**	**FlowDroid**	**SFlowInfer**
$\surd$, 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. 7. Summary of comparison with other taint analysis tools ($\surd$ = correct warning, $\times$ = false warning, $\bigcirc$ = missed flow).

Type-2 errors reflect key-value dependences. The following code, adapted from personalblog, is a Type-2 error:

```
HashMap map = ...; PrintWriter out = ...;
String id = request.getParameter(“id”);
User user = (User) map.get(id);
out.print(user.getName());
```

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 would not lead to flow violations. Most false positives are due to our conservative assumption about unknown libraries, e.g., that a tainted argument always propagate to the left-hand-side (see Sect. 4). The results are presented in Fig. 6. Additional results and nontrivial examples of type errors can be found in [7].

Comparison Direct comparison with TAJ [20], F4F [18], and ANDROMEDA [19] is impossible because the analysis tools are proprietary, and therefore unavailable. Instead, we run SFlowInfer on DroidBench [5], which is a suit of 39 Android apps, and compare with three other taint analysis tools – AppScan Source [2], Fortify SCA [1], and FlowDroid [5], using the results presented by Fritz et al. [5]. The comparison with AppScan Source is an indirect comparison with TAJ, F4F, and ANDROMEDA, because these analyses are built into AppScan Source.

For space reasons, Fig. 7, which borrows the format from Fritz et al. [5], only presents the summary of the comparison. Detailed comparison results can be found in our technical report [7]. 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.

6 Related Work

Unfortunately, we cannot include all related work on information flow control. More related work is discussed in the accompanying technical report [7].

The most closely related to ours is the work by Shankar et al. [17]. 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 [4]. CQual relies on dependence graphs built using points-to analysis. 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. [20] present TAJ, a points-to-based taint analysis for industrial applications. In order to handle Struts, TAJ 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, according to Sridharan et al. [18], 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 [9] present a static analysis based on a scalable and precise points-to analysis. In contrast, our inference analysis is type-based and modular. 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. [18] 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. [19] present ANDROMEDA, a demand-driven analysis that improves on F4F.

Volpano et al. [21] and Myers [12] present type systems for secure information flow. These systems are substantially more complex and powerful than SFlow. They focus on type checking and do not include type inference, or include only local type inference. In contrast, SFlowInfer handles large web applications.

7 Conclusions

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

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

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. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
4. J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, May 1999.
5. 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.
6. W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.
7. W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. Rensselaer Polytechnic Institute Technical Report RPI-CS-13-02. http://www.cs.rpi.edu/~huangw5/docs/RPI-CS-13-02.pdf, 2013.
8. 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.
9. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security*, 2005.
10. 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.
11. A. Milanova and W. Huang. Composing information flow type systems with reference immutability. In *FTfJP*, 2013.
12. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
13. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.
14. OWASP. Top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2013.
15. 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.
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. 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.
19. 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.
20. 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.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, pages 167–187, 1996.