To appear in *Proceedings of the 16th USENIX Security Symposium* (Security '07)

SIF: Enforcing Confidentiality and Integrity in Web Applications

Stephen Chong K. Vikram Andrew C. Myers
Department of Computer Science, Cornell University

Abstract

SIF (Servlet Information Flow) is a novel software framework for building high-assurance web applications, using language-based information-flow control to enforce security. Explicit, end-to-end confidentiality and integrity policies can be given either as compile-time program annotations, or as run-time user requirements. Compile-time and run-time checking efficiently enforce these policies. Information flow analysis is known to be useful against SQL injection and cross-site scripting, but SIF prevents inappropriate use of information more generally: the flow of confidential information to clients is controlled, as is the flow of low-integrity information from clients. Expressive policies allow users and application providers to protect information from one another.

SIF moves trust out of the web application, and into the framework and compiler. This provides application deployers with stronger security assurance.

Language-based information flow promises cheap, strong information security. But until now, it could not effectively enforce information security in highly dynamic applications. To build SIF, we developed new language features that make it possible to write realistic web applications. Increased assurance is obtained with modest enforcement overhead.

1 Introduction

Web applications are now used for a wide range of important activities: email, social networking, on-line shopping and auctions, financial management, and many more. They provide services to millions of users and store information about and for them. However, a web application may contain design or implementation vulnerabilities that compromise the confidentiality, integrity, or availability of information manipulated by the application, with financial, legal, or ethical implications. According to a recent report [33], web applications account for 69% of Internet vulnerabilities. Current techniques appear inadequate to prevent vulnerabilities in web applications.

In general, information security vulnerabilities arise from inappropriate information dependencies, so tracking information flows within applications offers a comprehensive solution. Confidentiality can be enforced by controlling information flow from sensitive data to clients; integrity can be enforced by controlling information flow from clients to trusted information—as a side effect, protecting against common vulnerabilities like SQL injection and cross-site scripting. In fact, recent work [14, 19, 37, 15] on static analysis of PHP and Java web applications has used dependency analyses to find many vulnerabilities in existing web applications and web application libraries. Dynamic tainting can detect some improper dependencies and has also proved useful in detecting vulnerabilities [39, 6]. However, static analyses have the advantage that they can conservatively identify information flows, providing stronger security assurance [28].

Therefore, we have developed Servlet Information Flow (SIF), a novel framework for building web applications that respect explicit confidentiality and integrity information security policies. SIF web applications are written in Jif 3.0, an extended version of the Jif programming language [21, 24] (which itself extends Java with information-flow control). The enforcement mechanisms of SIF and Jif 3.0 track the flow of information within a web application, and information sent to and returned from the client. SIF reduces the trust that must be placed in web applications, in exchange for trust in the servlet framework and the Jif 3.0 compiler—a good bargain because the framework and compiler are shared by all SIF applications.

The security policies used in SIF are both strong and expressive. Information flow is tracked through a type system that tracks all information flows, not merely explicit flows. Security enforcement is *end-to-end*, because policies are enforced on information from when it enters the web application, to when it leaves, even as information flows between different client requests. The security policies are expressive, allowing complex security requirements of multi-user systems to be enforced. Unlike prior frameworks for tracking information flow in web applications, policies can express fine-grained requirements for *both* confidentiality and integrity. Further, the interactions between confidentiality and integrity are controlled.

The end-to-end security provided by information-flow control has long been appealing, but much theoretical work on language-based information flow has not yet been successfully put into practice. We have identified limitations of existing security-typed languages for rea-

soning about security in a dynamic external environment, and we have extended the Jif language with new features supporting these dynamic environments, resulting in a new version of the language, Jif 3.0.

Information-flow control mechanisms work by labeling information. In previous information flow mechanisms, the space of labels is essentially static. In earlier versions of Jif, for example, labels are expressed in terms of principals, but the set of principals is fixed at compile time. This is a serious limitation for web applications, which often add new users at run time. Jif 3.0 adds the ability for applications to create their own principals, dynamically extending the space of information labels. Moreover, Jif 3.0 allows applications to implement their own authentication and authorization mechanisms for these application-specific principals—a necessity given the diversity of authentication schemes needed by different applications. Jif 3.0 also improves Jif's ability to reason about dynamic security policies, allowing, for example, web application users to specify their own security requirements at run time and have them enforced by the information flow mechanisms. These new mechanisms create new information channels, but Jif 3.0 tracks these channels and prevents their misuse.

To explore the performance and usability of SIF, we developed two web applications with non-trivial security requirements: an email application specialized for cross-domain communication, and a multiuser shared calendar. Both applications add new principals and policies at run time, and both allow users to define their own information security policies, which are enforced by the same mechanisms used for compile-time policies.

In summary, this paper makes three significant contributions:

- It shows how to use language-based information flow to construct a practical framework for high-assurance web applications, in which information flow is tracked to and from clients, and users can specify and reason about information security. To our knowledge, this is the first implemented web application framework to strongly enforce both confidentiality and integrity.
- It shows that application-defined mechanisms for access control and authentication, and a dynamically extensible space of labels, can be integrated securely with language-based information flow.
- It describes the experience using these new mechanisms to build realistic web applications.

The remainder of the paper is structured as follows. Section 2 gives an overview of the Servlet Information Flow framework, including some background on Jif. Section 3 introduces the new dynamic features in Jif 3.0, which enhance Jif's ability to express and enforce dynamic security requirements. Our experience with building web applications in SIF is described in Section 4. Section 5 covers related work, and Section 6 concludes.

2 Servlet Information Flow framework

SIF is built using the Java Servlet framework [7], but presents a higher-level interface to web applications. Through a combination of static and dynamic mechanisms, SIF ensures that web applications use data only in accordance with specified security policies, by tracking the flow of information in the server, and information sent to and from the client. Web applications in SIF are written entirely in Jif 3.0, an extended version of the *security-typed language* [36] Jif, in which types are annotated with information flow policies. Security policies are enforced on information as it flows through the system, giving stronger security assurance than ordinary (discretionary) access control.

In designing SIF, we faced two main challenges. The first was identifying information flows in web applications, including information that flows over multiple requests. For example, a request sent to a server by a user may contain information about the user's previous request and response. The second challenge was to restrict insecure information flows while providing sufficient flexibility to implement full-fledged web applications. The resulting framework is a principled approach to designing realistic, secure web applications.

SIF is implemented in about 4040 non-comment, non-blank lines of Java code. An additional 960 lines of Jif code provide signatures for the Java classes that web applications interact with. Jif signatures provide security annotations for Java classes, and expose only a subset of the actual methods and fields to clients. SIF web applications are compiled against the Jif signatures, but linked at run time against the Java classes. Some Java Servlet framework functionality makes reasoning about information security infeasible. Using signatures and wrapper classes, SIF necessarily limits access to this functionality, but without preventing implementation of full-fledged web applications.

In this section, we first describe the threat model that SIF addresses, and the security assurances that SIF provides. We present some background about Jif before describing the design of SIF.

2.1 Threat model and security assurance

Threat model. We assume that web application clients are potentially malicious, and that web application implementations are benign but possibly buggy. Thus, we aim to ensure that appropriate confidentiality and integrity security policies are enforced on server-side in-

formation regardless of the actions of clients, or the mistakes of well-meaning application programmers.

Although the Jif programming language prevents the unintentional violation of information security, it provides mechanisms for explicit intentional downgrading of security policies (see Section 4.3). While a well-meaning programmer will be unable to accidentally misuse these mechanisms, a malicious programmer may be able to subvert them, or use certain covert channels that Jif does not track (see Section 2.2).

We do not address network threats, such as denial of service attacks, or the interception and alteration of data sent over the network.

The Jif compiler and SIF are added to the trusted computing base, which already includes the servlet container, and the software stack required to run the servlet container. Note that SIF web applications are not part of the trusted computing base, whereas in standard servlet frameworks, web applications must be trusted.

Security assurance. In a typical web application, security assurance consists of convincing each party with a stake in the system that the application enforces their security requirements. Obviously users would like to have assurance that information they input will be confidential, and information they view is not corrupted. The application provider (i.e., deployer) may also have confidentiality and integrity requirements for its information. Like other recent work on improving security of web applications (e.g., [14, 18, 37, 15]), we focus on providing assurance to deployers. The difference here is that SIF enforces rich policies for information integrity and confidentiality, including policies provided by the user.

Although we focus on providing assurance to deployers, it is worth considering security assurance from a web application user's perspective. Users must be convinced that they are communicating with an application that enforces their security requirements. The security validation offered by SIF effectively partitions the security assurance problem into two parts: first, ensuring that the application respects users' security requirements, and second, ensuring the server users communicate with is correctly running the application.

SIF addresses the first part of the assurance problem: verifying the security properties of web application code. SIF does not address the second part: convincing a remote client they are communicating with verified code. This step is important if the web application provider might be malicious. However, remote attestation methods [34, 10, 30] seem likely to be effective in solving this second problem. Attestation methods could be used to sign application code, or alternatively, to sign a verification certificate from a trusted SIF compiler that has checked the code. We leave integration of attestation mechanisms till future work.

In any case, concern about malicious application providers should not be exaggerated; users' willingness to spend money via web applications suggests they already place a modicum of trust in them. This work aims to ensure this trust is justified. At a minimum, this means application deployers can be more confident in making possibly legally binding representations to their users.

The SIF framework provides the following security assurances to deployers of web applications.

- SIF applications enforce explicit information security policies. In particular, SIF ensures that information sent to the client is permitted to be read by the client, thus ensuring that confidential information held on the server is not inadvertently released to the client. Further, information received from the client is marked as tainted by the client, helping prevent inappropriate use of low-integrity information. Thus, useful confidentiality and integrity restrictions are enforced in SIF applications.
- The information security policies of back-end systems (e.g., a database, file system, or legacy application) are also enforced, provided these systems have appropriate interfaces annotated with Jif 3.0 security policies. Thus, adding a web front-end to an existing system does not weaken the security assurance of that system, modulo the assumptions of our threat model.
- Jif ensures that security policies on information are not unintentionally weakened, or *downgraded*. However, many web applications that handle sensitive information intentionally downgrade information as part of their functionality. As discussed further in Section 4.3, SIF web applications must satisfy rules that enforce *selective downgrading* [22, 26] and *robustness against all attackers* [5], security conditions that provide strong information flow guarantees in the presence of downgrading.
- SIF web applications can produce only well-formed HTML. While cascading style sheets and JavaScript may be used, they cannot be dynamically generated, and must be explicitly specified in the deployment descriptor, where they can be more easily reviewed by the application deployer. The deployer thereby gains assurance that a web application does not contain malicious client-side code.

2.2 Background on Jif

SIF web applications are written in Jif 3.0, a new version of the Jif programming language. To understand the design of SIF, some background on the Jif programming language is helpful. Readers familiar with Jif may skip this subsection. Details of some of the new features of Jif 3.0 are given in Section 3.

Jif is a *security-typed language* [36]: a type has a security label L that describes restrictions on information at that type, which the compiler enforces. Security-type systems like that in Jif can enforce *noninterference*, ensuring that information labeled L can depend only on information labeled L or with a less restrictive label [28]. In other words, information cannot leak from higher to lower levels, nor can untrusted information affect trusted information. Proofs for noninterference exist for numerous security-typed languages, but not for any language as expressive as Jif. Jif labels are based on policies from the *decentralized label model* (DLM) [22], in which principals express ownership of information-flow policies.

A *principal* is an entity with security concerns, and the power to observe and change certain aspects of the system. In a web application, principals may be users of the application, user groups, or even the web application itself; SIF applications may choose which entities to model as principals. Web application principals may have different security concerns, and do not necessarily trust each other. By allowing principals to have different security policies, the DLM can express security concerns of mutually distrusting principals.

A principal `p` may delegate to another principal `q`, in which case `q` is said to *act for* `p`. The *acts-for* relation is reflexive and transitive, and is similar to the *speaks-for* relation [16]. The *acts-for* relation is needed to express trust relationships between principals, and can encode groups and roles. Jif supports a *top principal* $\top$ able to act for all principals, and a *bottom principal* $\bot$ that allows all principals to act for it. A principal may also grant its *authority* to code, meaning the code is trusted to perform actions such as declassification that could violate the principal's information security.

Jif labels are constructed from *reader policies* and *writer policies* [5]. A reader policy $o \rightarrow r_1, \ldots, r_n$ means that principal o owns the policy, and o permits any principal that can act for any r_i (or o itself) to read the data. For example, the reader policy $\top \rightarrow \mathtt{p}$ says that the top principal permits `p` to observe information. A writer policy $o \leftarrow w_1, \ldots, w_n$ is owned by principal o, and o has permitted any principal that can act for any of $w_1, \ldots, w_n$, or o to have influenced ("written") the data.

Reader policies restrict to which principals information may flow, whereas writer policies describe from which principals information may have flowed. Reader policies thus describe confidentiality, and writer policies describe integrity (provenance) of information.

A Jif label is a pair of a *confidentiality policy* and an *integrity policy*, written $\{c \ ; \ d\}$ for confidentiality policy c and integrity policy d. The set of *confidentiality policies* is formed by closing reader policies under conjunction and disjunction, denoted $\sqcup$ and $\sqcap$ respectively. The conjunction of two confidentiality policies, $c_1 \sqcup c_2$, enforces the restrictions of both c_1 and c_2. Thus, the readers permitted by $c_1 \sqcup c_2$ is the intersection of readers permitted by c_1 and c_2. Similarly, the readers permitted by the disjunction $c_1 \sqcap c_2$ is the union of readers permitted by c_1 and c_2. *Integrity policies* are formed by closing writer policies under conjunction and disjunction. Dually to confidentiality, conjunction and disjunction are respectively denoted $\sqcap$ and $\sqcup$.

For example, in the label $\{Alice \rightarrow Bob \sqcup Chuck \rightarrow Bob, Dave \ ; \ Alice \leftarrow \top\}$, the confidentiality policy is the join of two reader policies, $Alice \rightarrow Bob$ and $Chuck \rightarrow Bob, Dave$. Thus, information with this label can be read only by principals that can act for at least one of $Alice$ or Bob, *and* at least one of $Chuck$, Bob, or $Dave$; clearly, Bob is one such principal. The integrity policy of the label consists of a single writer policy, owned by $Alice$, stating that $Alice$ believes the data has been influenced only by principals able to act for $Alice$ or the top principal $\top$. SIF uses confidentiality policies to restrict what information is sent to the client, and integrity policies to restrict how information received from the client is used.

Secure information flow requires that the label on a piece of information can only become more restrictive as the information flows through the system. Given labels L and L', we write $L \sqsubseteq L'$ if the label L' restricts the use of information at least as much as L does. To handle computations that combine information from different sources, the label $L_1 \sqcup L_2$ imposes the restrictions of both L_1 and L_2.

The types of variables and expressions in Jif programs include labels. For example, a value with type $\mathtt{int}\{o \rightarrow r\mathtt{;} \bot \leftarrow \bot\}$ is an integer with label $\{o \rightarrow r \ ; \ \bot \leftarrow \bot\}$: it can be read only by principals that can act for r or o, and has the lowest possible integrity. A Jif programmer may annotate the type declarations of fields, variables, and methods with labels; use of fields, variables, and methods must comply with the label annotations. For types left unannotated, the Jif compiler either chooses default labels, or automatically infers labels, thus reducing the annotation burden on the programmer.

Although a Jif programmer may annotate a program with arbitrary labels, he does not have complete control over security. Labels must be internally consistent for the program to type-check, and moreover, the labels must be consistent with security policies from the external environment. In SIF, a web application interacts with the external environment through the SIF interfaces, as well as interfaces for back-end services (e.g., databases).

Jif's type system prevents labeled information from being unintentionally *downgraded*, or assigned a less-restrictive label. Downgrading confidentiality increases the set of principals permitted to read the information, whereas downgrading integrity reduces the set of prin-

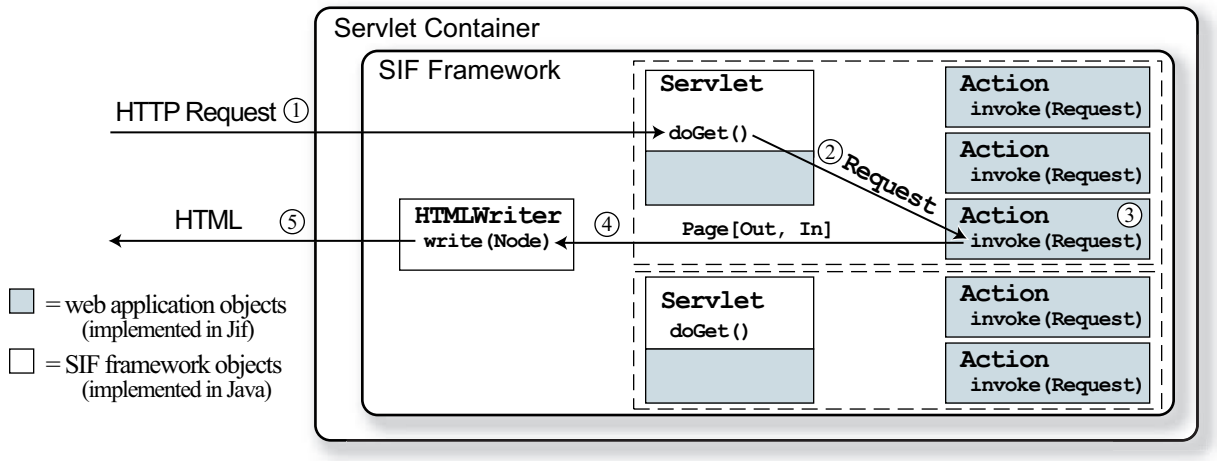

Figure 1: Handling a request in SIF.

cipals considered to have influenced the information. The type system prevents unintentional downgrading by tracking the data dependencies (information flow) in the program, including *implicit flows* [8]: covert storage channels that arise from program control structure. Jif does permit information to be intentionally downgraded, but any code that does so requires the authority of all principals whose reader or writer policies are weakened or removed as a result of the downgrading.

Timing and synchronization channels. Jif's type system does not track information flow via timing or termination channels. These covert channels are not a serious concern if web applications are not implemented by adversaries; we assume that application programmers are not malicious. Other work (e.g., [1, 31]) has investigated checking and transforming security-typed code to remove timing channels. Termination channels (which can be regarded as an extreme timing channel) are low-bandwidth, leaking at most one bit per interaction with the web application, that is, one bit per request.

Jif was developed assuming a single-threaded execution model. However, SIF web applications are multi-threaded Jif programs, and thread synchronization can create covert timing channels that transmit information. This risk can be mitigated by configuring the web server to handle at most one concurrent request per servlet, or by isolating concurrent requests or sessions in the protection domains offered by some Java run-time systems [12, 3].

2.3 Design of SIF

Like the Java Servlet framework, SIF allows application code to define how client requests are handled. However, there are some structural differences that facilitate the accurate tracking of information flow. Figure 1 presents an overview of how SIF handles a request from a web client:

1. An HTTP request is made from a web client to a servlet;
2. The HTTP request is wrapped in a `Request` object;
3. An appropriate `Action` object of the servlet is found to handle the request, and its `invoke` method called with the `Request` object;
4. The action's `invoke` method generates a `Page` object to return for the request;
5. The `Page` object is converted into HTML, which is returned to the client.

Step 1: HTTP request from web client to servlet. Web applications must extend the class `Servlet`, which is similar to the `HttpServlet` class of the Java Servlet framework. Figure 2 shows a simplified Jif signature for the `Servlet` class, as well as other key classes of SIF. The important aspects of these signatures are explained as they arise, but because of space limitations, the syntax of Jif methods and fields are not fully explained.

Web clients establish *sessions* with the servlet; sessions are tracked by the servlet container, as in the Java Servlet specification. The SIF framework creates a *session principal* for each session, which can be thought of as corresponding to the session key shared between the client and server [16], if such a key exists. The application would typically define its own user principals, which can delegate to the session principal.

Step 2: HTTP request wrapped in a `Request` object. The class `Request` is a SIF wrapper class for an HTTP request, providing restricted access to information in the request, via the `getParam` method. The restricted interface ensures that web applications are unable to circumvent the security policies on data contained in the request, as described below.

Step 3: An `Action` is found and invoked. Web applications implement their functionality in *actions*, which are

```
abstract class Servlet {
    // allows servlets to specify
    // a default action
    protected Action{req} defaultAction(Request req);

    // allows servlets to create a
    // servlet-specific SessionState object
    protected SessionState createSessionState();

    public void setReturnPage{*:req.session}(
                Request{*:req.session} req,
                label out, label in,
                Node[out,in]{*in} page)
          where {*out;*in} <= {*:req.session};
}

abstract class Action {
    public abstract void
          invoke{*lbl}(label{*lbl} lbl,
                       Request{*lbl} req)
         where caller(req.session);
}

// base class of HTML elements
abstract class Node[label Out, label In] {  }
```

```
final class Request {
    // principal representing the session
    // between client and server
    public final principal session;

    // reference to the Servlet
    public final Servlet servlet;

    // acquire a parameter value from the Request
    public String{*inp.L ⊔ inp ⊔ (⊥→⊥; ⊤←session)}
              getParam(Input inp);

    // obtain a reference to
    // the SessionState object
    public SessionState getSessionState();
}

final class Input {
    private final Nonce n;
    public final label L;
}
abstract class InputNode[label Out, label In]
    extends Node[Out,In] {
    // framework statically enforces Out ⊔ In ⊑ inp.L
    private final Input{L} inp;
}
```

Figure 2: Jif signatures for SIF classes

application-defined subclasses of the SIF class `Action`. A SIF servlet may have many action objects associated with it; each action object belongs to a single servlet.

Actions can be used as the targets of forms and hyperlinks. For example, the target of a form is an action object responsible for receiving and processing the data the user submits via the form. This mechanism differs from the standard Java servlet interface, which requires the application implementor to write explicit request dispatching code (the `doGet` method). However, explicit dispatch code in the application makes precise tracking of information flow difficult, as the dispatch code is executed for all requests, even though different requests may reveal different information. By avoiding dispatch code, the action mechanism permits more precise reasoning about the information revealed by client requests to the server, as discussed further in Section 2.4.

Action objects may be *session-specific actions*, which can only ever be used by a single session, or they may be *external actions* not specific to any given session. All action objects within a given servlet have a unique identifier. For session-specific actions, the identifier is a secure nonce, automatically generated by the framework on construction of the action. For external actions, the identifier is a (human-readable) string specified by the web application. Since external actions have fixed identifiers, they may be the target of external hyperlinks, such as a hyperlink in static HTML on a different web site.

When an HTTP request is received by a servlet, the framework finds a suitable action to handle it. Typically, the HTTP request contains a parameter value specifying the unique identifier of the appropriate action; for example, forms generated by the servlet identify the action to which the form is to be submitted. If the HTTP request does not contain an action's unique identifier, then a default action specified by the `Servlet.defaultAction` method is used to handle the request. This default is useful for handling the first request of a new session. If the HTTP request contains an invalid action identifier (e.g., the identifier of a session-specific action of an expired or invalidated session), an error page is returned, which then redirects the user to the default action.

Actions allow web applications to maintain control over application control flow. Because session-specific actions are named with a nonce, other sessions cannot invoke them. In addition, SIF tracks the *active set* of actions for each session. An error page is returned if a request tries to invoke an action that is not active. The active set contains all external actions, and all session-specific actions that were targets of hyperlinks and forms of the last response. Thus, a client by default cannot resubmit a form by replaying its (inactive) action identifier.

Once the appropriate action object has been found, the `invoke` method is called on it with a `Request` object as an argument. The `invoke` method executes with the authority of the session principal, as shown by the `where caller(req.session)` annotation in Figure 2.

Web applications implement their functionality in the action's `invoke` method, as Jif 3.0 code. If required, the `invoke` method can access back-end services (e.g., a database) provided that suitable Jif interfaces exist for the services. For example, web applications can access the file system since the Jif run-time library provides a Jif interface for it, which translates file system permissions into Jif security policies.

SIF web applications can provide secure web interfaces to legacy systems, by accessing the legacy systems as back-end services. The information security of these systems is not compromised by allowing SIF applications to access them, since all accesses from Jif code

must conform to the system's Jif interface.

Step 4: The `invoke` method generates a `Page` object. An object of the class `Page` is a representation of an HTML page. SIF uses the class `Node` to represent HTML elements; the class `Page`, and other HTML elements, such as `Paragraph` and `Hyperlink`, are subclasses of `Node`. Nodes may be composed to form trees, which represent well-formed HTML code. The class `Node` is parameterized by two labels, `Out` and `In`. The `Out` label is an upper bound on the labels of information contained in the node object and its children. For example, an HTML body may contain several paragraphs, each of which contains text and hyperlinks; the `Out` parameter of each `Paragraph` node is at least as restrictive as the `Out` parameters of its child `Nodes`. The `In` parameter is used to bound information that may be gained by from subsequent requests originating from this page, and is discussed further in Section 2.4.

The `Action.invoke` method must generate a `Page` object, and call `Servlet.setReturnPage` with that `Page` object as an argument. The signature for `Servlet.setReturnPage` ensures that the `Out` parameter of the `Page` is at most as restrictive as the label $\{\top \rightarrow \texttt{req.session}; \bot \leftarrow \bot\}$, where `req.session` is the session principal. This label is an upper bound on all labels that permit the principal `req.session` to read information, and thus the `Page` object returned for the request can contain only information that the session principal is permitted to view. This restriction is enforced statically through the type-system, and requires no runtime examination of labels by the SIF framework. Thus, assurance is gained prior to deployment that confidential information on the server is not inadvertently released.

In addition, by requiring the application to produce `Page` objects instead of arbitrary byte sequences, SIF can ensure that each input field on a page has an appropriate security policy associated with it (see Section 2.4), and that the web application serves only well-formed HTML that does not contain possibly malicious JavaScript.

Step 5: The `Page` is converted into HTML. SIF converts the `Page` object into HTML, which is sent to the client. The `Page` object may contain hyperlinks and forms whose targets are actions of the servlet; SIF ensures that the HTML output for these hyperlinks and forms contain parameter values specifying the appropriate actions' unique identifiers; if the user follows a hyperlink or submits a form, the appropriate action is invoked.

2.4 Information flow over requests

The Jif compiler ensures that security policies are enforced end-to-end within a servlet, that is, from when a request is submitted until a response is returned. However, information may flow over multiple requests within the same session, for example, by being stored in session state, or by being sent to a (well-behaved) client that returns it in the next request. SIF tracks information flow over multiple requests, to ensure that appropriate security labels are enforced on data at all times.

Information flow through parameter values. SIF requires each input field on a page to have an associated security label to be enforced on the input when submitted. This label is statically required to be at least as restrictive as the label of any default value for the input field, to prevent a default value from being sent back to the server with a less restrictive policy enforced on it.

SIF ensures that the submitted value of an input field has the correct label enforced on it by preventing applications from arbitrarily accessing the HTTP request's map from parameter keys to parameter values. Instead, when an input field is created in the outgoing `Page` object, an `Input` object is associated with it. An `Input` object is a pair $(\texttt{n}, \texttt{L})$, where `n` is a freshly generated nonce, and `L` is the label enforced on the input value. An application can retrieve a data value from an HTTP request only by presenting the `Input` object to the `Request.getParam(Input inp)` method, which checks the nonce, and returns the submitted value with label `inp.L` enforced on it. This "closes the loop," ensuring that data sent to the client has the correct security enforced on it when the client subsequently sends it back.

SIF does not try to protect against the user copying sensitive information from the web page, and pasting into a non-sensitive input field. That is impossible in general, and the application should define labels that prevent the user from seeing information that they are not trusted to see. By keeping track of input labels, SIF prevents web applications from laundering away security policies by sending information through the client. As discussed in Section 2.5, the user can also inspect the labels on inputs to see how the application will treat the information.

The `getParam` method signature also ensures that the label $\{\bot \rightarrow \bot\ ;\ \top \leftarrow \texttt{session}\}$ is enforced on values submitted by the user. This label indicates that the value has been influenced by the session principal. Thus, SIF ensures that the integrity policy of any value obtained from the client correctly reflects that the client has influenced it; the Jif 3.0 compiler then ensures that this "tainted," or low-integrity, information cannot be incorrectly used as if it were "untainted," or high-integrity. This helps avoid vulnerabilities such as SQL injection, where low-integrity information is used in a high-integrity context.

Information flow through session state. Java servlets typically store session state in the session map of the class `javax.servlet.http.HttpSession`. However, direct access to the session map would allow SIF applications to bypass the security policies that

should be enforced on values stored in the map. Instead, SIF web applications may store state in fields of session-specific actions, or in an application-defined subclass of `SessionState`. Since fields must have labels, the Jif compiler ensures that web applications honor labels associated with values stored in the state. Web applications may override the method `Servlet.createSessionState` to create an appropriate `SessionState` object; SIF ensures at run time that this method is called exactly once per session.

Information flow through action invocation. A subtlety of the framework is that the very act of invoking an action, by following a hyperlink or submitting a form, may reveal information to the web application. For example, if a hyperlink to some action a is generated if and only if some secret bit is 1, then knowing that a is invoked reveals the value of the secret bit.

To account for this information flow, the `Action.invoke` method takes two arguments: a `label lbl`, and a reference to the `Request` object. The label `lbl` is an upper bound on the information that may be gained by knowing which action has been invoked. This means that `lbl` must be at least as restrictive as the output information for the hyperlink or form used to invoke the action. In our example, the value of `lbl` when invoking a would be at least as restrictive as the label of the secret bit. In general, the value for `lbl` is the value of the `In` parameter of the `Node` that contains the link to the action; the constructors for the `Node` subclasses ensure that the parameter `In` correctly bounds the information that may be gained by knowing the node was present in the `Page` returned for the request.

The method signature for `Action.invoke` ensures that the security label `lbl` is enforced on the reference to the `Request` object ("...`Request{*lbl} req`...") and that `lbl` is a lower-bound for observable side-effects of the method ("`invoke{*lbl}(...)`"), meaning that any effects of the method (such as assignments to fields) must be observable only at security levels bounded below by `lbl`. These restrictions ensure that SIF correctly tracks the information that may be gained by knowing which actions were available for the user to invoke.

2.5 Deploying SIF web applications

SIF web applications may be deployed on standard Java Servlet containers, such as Apache Tomcat, and thus may be used in a multi-tier architecture wherever Java servlets are used. The SIF and Jif run-time libraries must be available on the class path, but deployment of SIF web applications is otherwise similar to deployment of ordinary Java servlets. The deployer of a SIF web application is free to specify configuration information in the application's deployment descriptor (the `web.xml` file). For example, the deployer may require all connections to use SSL, thus protecting the confidentiality and integrity of information in transit between client and server. Additionally, there are several SIF-specific options that a deployer may specify in the deployment descriptor.

Cascading style sheets. SIF applications must use the `Node` subclasses to generate responses to requests, which allows them to generate only well-formed HTML. To allow flexibility in presentation details such as colors and font attributes, SIF permits the deployment descriptor to specify a cascading style sheet (CSS) to use in the presentation of all HTML pages generated by the application; SIF adds this URL in the head of all generated HTML pages. `Node` objects can specify a `class` attribute, allowing style sheets to provide almost arbitrary formatting. While this allows great flexibility, care must be taken that the CSS does not contain misleading formatting. For example, inappropriate formatting might lead a user to enter sensitive information into a non-sensitive input field, such as a social security number into an address field. The deployer should review the CSS before deploying the application.

JavaScript. Dynamically generated JavaScript can provide rich user interfaces, but introduces new possibilities for security violations and covert channels. SIF does not allow web applications to send dynamic JavaScript to the client. However, as with CSSs, SIF allows deployment descriptors to specify a URL containing (static) JavaScript code to be included on all generated HTML pages. Explicit inclusion of JavaScript permits easy review by the deployer. Ideally, SIF should automatically check included JavaScript code (or perhaps an extension of JavaScript with information-flow control); we leave this to future work.

Policy visualization. User awareness of security policies is an important aspect of secure systems. Since SIF tracks the policies of information sent to the user, SIF can augment the user interface to inform the user of the security policies of data they view and supply. Provided the user trusts the interface (see Section 2.1), this helps prevent, for instance, a user from inappropriately copying sensitive information from the browser into an email, or from following an untrusted hyperlink.

Web applications may opt to allow SIF to automatically color-code information sent to the client, based on policy annotations. When the user presses a hotkey combination, JavaScript code recolors the page elements to reflect their confidentiality, varying from red (highly confidential) to green (low confidentiality). Both displayed information and inputs are colored appropriately. An additional hotkey colors the page based on the integrity policies of information. A third hotkey shows a legend of colors and corresponding labels so the user can identify

the precise security policy for each page element.

3 Language extensions

Web applications have diverse, complicated, and dynamic security requirements. For example, web applications display a plethora of authentication schemes, including various password schemes, password recovery schemes, biometrics, and CAPTCHAs to identify human users. Web applications often enforce dynamic security policies, such as allowing users to specify who may view and update their information. Moreover, the security environment of a web application is dynamic: new users are being created, users are starting and ending sessions, and authenticating themselves.

In order both to accommodate diverse, complicated, and dynamic security requirements, and to provide assurance that these requirements are met, we have produced Jif 3.0, a new version of Jif. Section 2.2 describes the previous version of Jif; this section presents new features that support dynamic security requirements: integration of information flow with application-defined authentication and authorization, and improved ability to reason about and compute with dynamic security labels and principals.

Care was needed in the design and implementation of these language extensions, since there is always a tension in language-based security between expressiveness and security. In particular, the new dynamic security mechanisms in Jif 3.0 create new information channels, complicating static analysis of information flow. Importantly, Jif 3.0 tracks these channels to prevent their misuse.

3.1 Application-specific principals

Principals are entities with security concerns. Applications may choose which entities to model as principals. Principals in Jif are represented at run time, and thus can be used as values by programs during execution. Jif gives run-time principals the primitive type `principal`. Jif 3.0 introduces an open-ended mechanism that allows applications great flexibility in defining and implementing their own principals.

Applications may implement the Jif 3.0 interface `jif.lang.Principal`, shown in simplified form in Figure 3. Any object that implements the `Principal` interface is a principal; it can be cast to the primitive type `principal`, and used just as any other principal. The `Principal` interface provides methods for principals to delegate their authority and to define authentication.

Delegation is crucial. For example, user principals must be able to delegate their authority to session principals, so that requests from users can be executed with their authority. The method call `p.delegatesTo(q)` returns `true` if and only if principal `p` delegates its authority to principal `q`. The implementation of a principal's `delegatesTo` method is the sole determiner of whether its authority is delegated. An *acts-for proof* is a sequence of principals $p_1, \ldots, p_n$, such that each p_i delegates its authority to p_{i+1}, and is thus a proof that p_n can act for p_1. Acts-for proofs are found using the methods `findProofUpTo` and `findProofDownTo` on the `Principal` interface, allowing an application to efficiently guide a proof search. Once an acts-for proof is found, it is verified using `delegatesTo`, cleanly separating proof search from proof verification.

```
interface Principal {
    String name();

    // does this principal delegate authority to q?
    boolean delegatesTo(principal q);

    // is this principal prepared to authorize the
    // closure c, given proof object authPrf?
    boolean isAuthorized(Object authPrf,
                         Closure[this] c);

    // methods to guide search for acts-for proofs
    ActsForProof findProofUpTo(Principal p);
    ActsForProof findProofDownTo(Principal q);
}
interface Closure[principal P] authority(P) {
    // authority of P is required to
    // invoke a Closure
    Object invoke() where caller(P);
}
```

Figure 3: Signatures for application-specific principals

The authority of principals is required for certain operations. For example, the authority of the principal $Alice$ is required to downgrade information labeled $\{Alice \rightarrow Bob \ ; \top \leftarrow \top\}$ to the label $\{Alice \rightarrow Bob, Chuck \ ; \top \leftarrow \top\}$ since a policy owned by $Alice$ is weakened. The authority of principals whose identity is known at compile time may be obtained by these principals approving the code that exercises their authority. However, for dynamic principals, whose identity is not known at compile time, a different mechanism is required. We have extended Jif with a mechanism for dynamically authorizing closures.

An *authorization closure* is an implementation of the interface `jif.lang.Closure`, shown in Figure 3. The `Closure` interface has a single method `invoke`, and is parameterized on a principal `P`. The `invoke` method can only be called by code that possesses the authority of principal `P`, as indicated by the annotation `where caller(P)`. Code that does not have the authority of principal `P` can request the Jif run-time system to execute a closure for `P`; the run-time system will do so only if `P` authorizes the closure.

The `Principal` interface provides a method for authorizing closures, `isAuthorized`. It takes two arguments: a `Closure` object instantiated with the principal represented by the `this` object, and an application-specific proof of authentication and/or authorization. For example, the proof might be a password, a checkable proof that the closure satisfies certain safety re-

quirements, or a collection of certificates or capabilities. The application-specific implementation of the `isAuthorized` method examines the closure and the proof object, and returns `true` if the principal grants its authority to the closure.

The `Principal` interface and authorization closures provide a flexible mechanism for web applications to implement their own authentication and authorization mechanisms. For example, in the case studies of Section 4, closures are used to obtain the authority of application users after they have authenticated themselves with a password. Other implementations of principals are free to choose other authentication and authorization mechanisms, such as delegating the authorization decision to a XACML service. Dynamic authorization tests introduce new information flows that are tracked using Jif's security-type system. To prevent the usurpation of a principal's authority, the Jif run-time library cannot execute a closure unless appropriately authorized.

Legacy systems may have their own abstractions for users, authentication, and authorization. Application-specific principals allow legacy-system security abstractions to be integrated with web applications. For example, when integrating with a database with access controls, database users can be represented by suitable implementations of the `Principal` interface; web applications can then execute queries under the authority of specific database users, rather than executing all queries using a distinguished web server user.

3.2 Dynamic labels and principals

Jif can represent labels at run time, using the primitive type `label` for run-time label values. Following work by Zheng and Myers [42], Jif 3.0's type system has been extended with more precise reasoning about run-time labels and principals. It is now possible for the label of a value (or a principal named in a label) to be located via a *final access path expression*. A final access path expression is an expression of the form $\mathtt{r.f_1.\ldots.f_n}$, where `r` is either a final local variable (including final method arguments), or the expression `this`, and each $\mathtt{f_i}$ is an access to a final field. For example, in Figure 2, the signature for the method `Request.getParam(Input inp)` indicates that the return value has the label `inp.L` enforced on it. Therefore, the Jif 3.0 compiler can determine that the label of the result of the `getParam` method is found in the object `inp`. The additional precision of Jif 3.0 is needed to capture this relationship.

This additional precision allows SIF web applications to express and enforce dynamic security requirements, such as user-specified security policies. SIF web applications can also statically control information received from the currently authenticated user, whose identity is unknown at compile time.

The use of dynamic labels and principals introduces new information flows, because which label is enforced on information may itself reveal information. Jif 3.0's type system tracks such flows, and prevents dynamic labels and principals from introducing covert channels.

3.3 Caching dynamic tests

To allow efficient dynamic tests of label and principal relations, the Jif 3.0 runtime system caches the results of label and principal tests. Separate caches are maintained for positive and negative results of acts-for and label tests. Care must be taken that the use of caches does not introduce unsoundness. When a principal delegation is added, the negative acts-for and label caches are cleared, as the new delegation may now enable new relationships. When a principal delegation is removed, entries in the positive acts-for and label caches that depend upon that delegation are removed, as the relationship may no longer hold.

When principals add or remove delegations, they should notify the Jif 3.0 runtime system, which updates the caches appropriately. Although an incorrectly or maliciously implemented principal p may fail to notify the runtime system, lack of notification can hurt only the principal p, since p (and only p) determines to whom its authority is delegated.

4 Case studies

Using SIF, we have designed and implemented two web applications. The first is a cross-domain information sharing system that permits multiple users to exchange messages. The second is a multi-user calendar application that lets users create, edit, and view events.

This section describes the key functionality of these applications, their information security requirements, and how we reflected these requirements in the implementations. Real applications must release information, reducing its confidentiality. In SIF, this is implemented by *downgrading* to a lower security label. We discuss and categorize downgrades that occur in the applications. Based on our experience, we make some observations about programming with information-flow control.

4.1 Application descriptions

Cross-domain information sharing (CDIS). CDIS applications involve exchange of information between different entities with varying levels of trust between them. For example, organizational policy may require the approval of a manager to share information between members of certain departments. Many CDIS systems provide an automatic process; for example, they determine what approval is needed, and delay information delivery until approval is obtained.

We have designed and implemented a prototype CDIS system. The interface is similar to a web-based email application. The application allows users to log in and compose messages to each other. A message may require review and approval by other users before it is available to its recipients. The review process is driven by a set of system-wide mandatory rules: each rule specifies for a unique sender-recipient pair which users need to review and approve messages. Once all appropriate reviewers have approved a message, it appears in the recipient's inbox. Each user also has a "review inbox," for messages requiring their approval or rejection. In this prototype, all messages are held centrally on the web server; a full implementation would be integrated with an SMTP server.

Calendar. We have also implemented a multi-user calendar system. Authenticated users may create, edit, and view events. Events have a time, title, list of attendees, and description. Events are controlled by expressive security policies, customizable by application users. A user can edit an event only if the user acts for the creator of the event (recall that the *acts-for* relation is reflexive). A user may view the details of an event (title, attendees, and description) if the user acts for either the creator or an attendee. An event may specify a list of additional users who are permitted to view the time of the event—to view an event, a user must act for the creator, for an attendee, or for a user on this list.

A user's calendar is defined to be the set of all events for which the user is either the creator or an attendee. When a user u views another user v's calendar, u will see only the subset of events on v's calendar for which u is permitted to see the details or time. If the user is permitted to view the time, but not the details of an event, the event is shown as "Busy."

Measurements. Measurements of the applications' code are given in Figure 4, including non-blank non-comment lines of code, lines with label annotations, and the number of `declassify` and `endorse` annotations, which indicate intentional downgrading of information (see Section 4.3).

Performance tests indicate that the overhead due to the SIF framework is modest. We compared the calendar case study application to a Java servlet we implemented with similar functionality, using the same backend database; the Java servlet does not offer the security assurances of the SIF servlet. Tests were performed using Apache Tomcat 5.5 in Redhat Linux, kernel version 2.6.17, running on a dual-core 2.2GHz Opteron processor with 3GB of memory. As the number of concurrent sessions varies between 1 and 245, the SIF servlet exhibits at most a 29% reduction in requests processed per second, showing that SIF does not dramatically affect scalability. At peak throughput, the Java servlet processes 2010 requests per second, compared with 1503 for the SIF servlet. Of the server processing time for a request to the SIF servlet, about 17% is spent rendering the `Page` object into HTML, and about 9% is spent performing dynamic label and principal tests.

4.2 Implementing security requirements

Many of the security requirements of both applications can be expressed using Jif's security mechanisms, including dynamic principals and security labels, and thus automatically enforced by Jif and SIF's static and runtime mechanisms. Other security requirements are enforced programmatically.

Principals. Users of the applications are application-specific principals (see Section 3.1). We factored out much functionality from both applications relating to user management, such as selecting users and logging on and off. The sharing of code across both case studies shows that SIF permits the design and implementation of reusable components. Figure 4 also shows measurements of the reusable user library.

The login process works as follows: a user and password are specified on the login screen, and if the password is correct, the authority of the user is dynamically obtained via a closure; the closure is used to delegate the user's authority to the session principal, who can then act on behalf of the now logged-in user.

In addition to user principals, the two applications define principals `CDISApp` and `CalApp`, representing the applications themselves. These model the security of sensitive information that is not owned by any one user, such as the set of application users. This information is labeled $\{p \rightarrow \top \; ; \; p \leftarrow \top\}$, where p is one of `CDISApp` or `CalApp`, and relevant portions are downgraded for use as needed. In particular, information in the database has this label. Since all information sent to and from the database (including data used in SQL queries) must have this label, the authority of the application principal (`CDISApp` or `CalApp`) is required to endorse information sent to the database and to declassify information received from it. This provides a form of access control, ensuring that only code authorized by the application principal is able to access the database. The need to explicitly endorse data used in SQL queries also helps to prevent SQL injection attacks, by making the programmer aware of exactly what information may be used in SQL queries.

Dynamic security labels. The security labels of Jif 3.0 are expressive enough to capture the case studies' information-sharing requirements. In particular, we are able to model the confidentiality and review requirements for CDIS messages by enforcing appropriate labels on the messages. For instance, suppose sender s is sending a message to recipient t. The confidentiality policy $s \rightarrow t$ would allow both s and t to read the mes-

	Lines	Annotated Lines	Downgrade Annotations	Functional downgrades			
				Access control	Imprecision	Application	Total
CDIS	1325	277	76	11	0	3	14
Calendar	1779	443	73	12	0	5	17
User	925	283	31	3	1	4	8

Figure 4: Summary of case studies.

sage. However, before t is permitted to read the message, it may need to be reviewed. Suppose reviewers $r_1, r_2, ..., r_n$ must review all messages sent from s to t. When s composes the message, it initially has the following confidentiality policy: $(s \rightarrow t, r_1, \ldots, r_n) \sqcup (r_1 \rightarrow r_1, \ldots, r_n) \sqcup \ldots \sqcup (r_n \rightarrow r_1, \ldots, r_n)$. In this policy, s permits t and all reviewers to read the message, and each reviewer permits all other reviewers to read the message. This label allows the message to be read by each reviewer, but prevents t from reading it. As each reviewer reviews and approves the message, their authority is used to remove their reader policy from the confidentiality policy using `declassify` annotations. Eventually the message is declassified to the policy $s \rightarrow t, r_1, \ldots, r_n$, which permits t to read it.

The calendar application also enforces user-defined security requirements by labeling information with appropriate dynamic labels. Event details have the confidentiality policy $c \rightarrow a_1, \ldots, a_n$ enforced on them, where c is the creator of the event and $a_1, \ldots, a_n$ are the event attendees. The time of an event has confidentiality policy $c \rightarrow a_1, \ldots, a_n \sqcap c \rightarrow t_1, \ldots, t_m$, where $t_1, \ldots, t_m$ are the users explicitly given permission by c to view the event time. Event labels ensure that times and details flow only to users permitted to see them; run-time label tests are used to determine which events a user can see.

4.3 Downgrading

Jif prevents the unintentional downgrading of information. However, most applications that handle sensitive information, including the case study applications, need to downgrade information as part of their functionality. Jif provides a mechanism for deliberate downgrading of information: *selective declassification* [22, 26] is a form of access control, requiring the authorization of the owners of all policies weakened or removed by a downgrade. Authorization can be acquired statically if the owner of a policy is known at compile time; or authorization can be acquired at run time through a closure (see Section 3).

Jif 3.0 programs must also satisfy typing rules to enforce *robust declassification* [40, 23, 5]. In the context of Jif, robustness ensures that no principal p (including attackers) is able to influence either *what* information is released to p (a *laundering attack*), or *whether* to release information to p. For a web application, robustness implies that users are unable to cause the incorrect release of information. Selective declassification and robust declassification are orthogonal, providing different guarantees regarding the downgrading of information.

In Jif programs, downgrading is marked by explicit program annotations. A `declassify` annotation allows confidentiality to be downgraded, whereas an `endorse` annotation downgrades integrity.

Downgrading annotations are typically clustered together in code, with several annotations needed to accomplish a single "functional downgrade." For example, declassifying a data structure requires declassification of each field of the structure [2]. The two applications had a combined total of 39 functional downgrades, with an average of 4.6 annotations per functional downgrade.

Figure 4 shows a more detailed breakdown of the use of downgrading in each case study. (Details of each downgrade appear in Appendix A.) We found that downgrading could be divided into three broad categories: access control, imprecision, and application requirements.

The first category is downgrades associated with discretionary access control. Discretionary access control is used as a mechanism to mediate information release between different application components; any information release requires explicit downgrading. For example, in the calendar application, the set of all events has the label $\{\texttt{CalApp} \rightarrow \top \; ; \; \texttt{CalApp} \leftarrow \top\}$; thus, downgrading is required both to extract events to display to the user, and to update events edited by the user; the authority of `CalApp` is required for these downgrades, and thus the downgrades serve as a form of discretionary access control to the event set. The choice of the label $\{\texttt{CalApp} \rightarrow \top \; ; \; \texttt{CalApp} \leftarrow \top\}$ for the event set necessitates these downgrades; using other labels may result in fewer downgrades, but without the benefits of this discretionary access control.

Imprecision is another reason for downgrading: sometimes the programmer can reason more precisely than the compiler about security labels and information flows. For example, suppose a method is always called with a non-null argument: Jif 3.0 has no ability to express this precondition, and conservatively assumes that accessing the argument may result in a `NullPointerException`. Since the exception may reveal information, a spurious information flow is introduced, which may require explicit downgrading later. Few downgrades fall into this category, giving confidence that Jif 3.0 is sufficiently expressive. Some imprecision could be removed entirely by extending the compiler to accept and reason about ad-

ditional annotations, as in JML [17].

Security requirements of the application provide the third category of downgrade reasons. These downgrades are inherent in the application, and cannot and should not be avoided. For example, in the calendar application, when users are added to the list of event attendees, more users are able to see the details of the event, an information release that requires explicit downgrading.

4.4 Programming with information flow

During the case studies' development, we obtained several insights into the design and implementation of applications with information flow control.

Abstractions and information flow. Information flow analysis tends to reveal details of computations occurring behind encapsulation boundaries, making it important to design abstractions carefully. Unless sufficient care is taken during design, abstractions will need to be modified during implementation. For example, we sometimes needed to change a method's signature several times, both while implementing the method body (and discovering flows we hadn't considered during design), and while calling the method in various contexts (as method invocation may reveal information to the callee, which we hadn't considered when designing the signature).

Coding idioms. We found that certain coding idioms simplified reasoning about information flow, by putting code in a form that either allowed the programmer to better understand it, or allowed Jif's type system to reason more precisely about it. As a simple example, consider the following (almost) equivalent code-snippets for assigning the result of method call `o.m()` to `x`, followed by an assignment to `y`:

1. `x = o.m(); y = 42;`
2. `if (o != null) { x = o.m(); } y = 42;`

The first snippet throws a `NullPointerException` if `o` is null, and thus information about the value of `o` flows to `x`, and also to `y` (since the assignment to `y` is executed only in the absence of an exception). The information flow to `y` is subtle, and a common trap for new Jif programmers. In the second snippet, no exception can be thrown (the compiler detects this with a data-flow analysis), and so information about `o` does not flow to `y`. This snippet avoids the subtle implicit flow to `y`. More generally, making implicit information flow explicit simplifies reasoning about information flow.

Declarative security policies. Many of the case studies' security requirements were expressed using Jif labels. SIF and the Jif compiler ensure that these labels (and thus the security requirements) are enforced end-to-end. In general, Jif's declarative security policies can relieve the programmer of enforcing security requirements programmatically, and give greater assurance that the requirements are met. This argues for even greater expressiveness in security policies, to allow more application security requirements to be captured, and to verify that programs enforce these requirements.

5 Related work

The most closely related work is Li and Zdancewic's [18], which proposes a security-typed PHP-like scripting language to address information-flow control in web applications. Their system has not been implemented. It assumes a strongly-typed database interface, and, like SIF, ensures that applications respect the confidentiality and integrity policies on data sent to and from the database. Their security policies can express *what* information may be downgraded; in contrast, the decentralized label model used in Jif specifies *who* needs to authorize downgrading. In a multi-user web application with mutually distrusting users, the concept of *who* a session or process is executing on behalf of is crucial to security. We believe that practical information-flow control will ultimately need to specify multiple aspects of downgrading [29]; extending the decentralized label model to reason about other downgrading aspects is ongoing work.

Huang et al. [14], Xie and Aiken [37], and Jovanovic et al. [15] all present frameworks for statically analyzing information flow in PHP web applications. Xie and Aiken, and Jovanovic et al. track information integrity using a dataflow analysis, while Huang et al. extend PHP's type system with type state. Livshits and Lam [19] use a precise static analysis to detect vulnerabilities in Java web applications. Each of these frameworks has found previously unknown bugs in web applications. Xu et al. [38], Halfond and Orso [11] and Nguyen-Tuong et al. [25] use dynamic information-flow control to prevent attacks in web applications. All of these approaches use a simple notion of integrity: information is either *tainted* or *untainted*. While this suffices to detect and prevent certain web application vulnerabilities, such as SQL injection, it is insufficient for modeling more complex, application-level integrity requirements that arise in applications with multiple mutually distrusting principals. Also, they do not address confidentiality information flows, and thus do not control the release of sensitive server-side information to web clients.

Xu et al. [39] propose a framework for analyzing and dynamically enforcing client privacy requirements in web services. They focus on web service composition, assuming that individual services correctly enforce policies. Their policies do not appear suitable for reasoning about the security of mutually distrusting users. Otherwise, this work is complementary, as we provide assurance that web applications enforce security policies.

While there has been much recent work on language-

based information flow (see [28, 29] for recent surveys), comparatively little has focused on creating real systems with information flow security, or on languages and techniques to enable this. No prior work has built real applications that enforce both confidentiality and integrity policies while dealing securely with their interactions.

The most realistic prior application experience is that of Hicks et al. [13], who use an earlier version of Jif to implement a secure CDIS email client, JPmail. Although there are similarities between JPmail and the CDIS mail application described here, SIF is a more convincing demonstration of information flow control in three ways. First, SIF is a reusable application framework, not just a single application. Second, SIF applications enforce integrity, not just confidentiality, and they ensure that declassification is robust [5]. Third, SIF applications can dynamically extend the space of principals and labels and define their own authentication mechanisms; JPmail relies on mechanisms for principal management and authentication that lie outside the scope of the application.

Askarov and Sabelfeld [2] use Jif to implement cryptographic protocols for mental poker. They identify several useful idioms for (and difficulties with) writing Jif code; recent extensions to Jif should assuage many of the difficulties.

Praxis High Integrity System's language SPARK [4] is based on a subset of Ada, and adds information-flow analysis. SPARK checks simple dependencies within procedures. FlowCaml [27] extends the functional language OCaml with information-flow security types. Like SPARK, it does not support features needed for real applications: downgrading, dynamic labels, and dynamic and application-defined principals.

Asbestos [9], Histar [41], and SELinux [20] are operating systems that track information flow for confidentiality and integrity. To varying degrees, they provide flexible security labels and application-defined principals. However, these systems are coarse-grained, tracking information flow only between processes. Information flow is controlled only dynamically, which is imprecise, and creates additional information flows from runtime label checking. By contrast, Jif checks information flow mostly statically, at the granularity of program variables, providing increased precision and greater assurance that a program is secure prior to deployment. Asbestos has a web server that allows web applications to isolate users' data from one another, using one process per user. All downgrades are performed by trusted processes. Unlike Jif, this granularity of information flow tracking does not permit different security policies for different data owned by a single user.

Tse and Zdancewic [35] present a monadic type system for reasoning about dynamic principals, and certificates for authority delegation and downgrading. Jif 3.0's dependent type system for dynamic labels and principals allows similar reasoning. Tse and Zdancewic assume that certificates are contained in the external environment, and do not provide a mechanism to dynamically create them. Closures in Jif 3.0 can be dynamically authorized, and may perform arbitrary computation, whereas Tse and Zdancewic's certificates permit only authority delegation and downgrading.

Swamy et al. [32] consider dynamic policy updates, and introduce a transactional mechanism to prevent *unintentional transitive flows* that may arise from policy updates. In Jif, policies are updated dynamically by adding and removing principal delegations, and unintentional transitive flows may occur. Their techniques are complementary to our work, and should be applicable to Jif to stop these flows.

6 Conclusion

We have designed and implemented Servlet Information Flow (SIF), a novel framework for building high-assurance web applications. Extending the Java Servlet framework, SIF addresses trust issues in web applications, moving trust out of web applications and into SIF and the Jif compiler.

SIF web applications are written entirely in the Jif 3.0 programming language. At compile time, applications are checked to see if they respect the confidentiality and integrity of information held on the server: confidential information is not released inappropriately to clients, and low-integrity information from clients is not used in high-integrity contexts. SIF tracks information flow both within the handling of a single request, and over multiple requests—it closes the loop of information flow between client and server.

Jif 3.0 extends Jif in several ways to make web applications possible. It adds sophisticated dynamic mechanisms for access control, authentication, delegation, and principal management, and shows how to integrate these features securely with language-based, largely static, information-flow control.

We have used SIF to implement two applications with interesting information security requirements. These web applications are among the first to statically enforce strong and expressive confidentiality and integrity policies. Many of the applications' security requirements were expressible as security labels, and are thus enforced by the Jif 3.0 compiler.

As language-based information-flow control becomes more mature, and information-flow tools become more useful and robust, we expect the task of writing and understanding programs with information-flow control to become easier. This work makes an important step towards wider use of information-flow control by providing a framework in which useful applications can be de-

signed, implemented, and deployed. The Jif 3.0 compiler and run-time system and the SIF framework are all publicly available.

Acknowledgments

We thank Nate Nystrom, Fred Schneider, Lantian Zheng, Xin Qi, and Jed Liu for useful suggestions. The research was supported in part by NSF awards 0430161 and 0627649 and by an Alfred P. Sloan Research Fellowship, and in part by TRUST (The Team for Research in Ubiquitous Secure Technology) and AF-TRUST (Air Force Team for Research in Ubiquitous Secure Technology for GIG/NCES), which receive support from the NSF (award 0424422) and from AFOSR (FA9550-06-1-0244), Cisco, British Telecom, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia and United Technologies.

References

[1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, January 2000.

[2] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. 10th European Symposium on Research in Computer Security (ESORICS)*, September 2005.

[3] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. 4th Symposium on Operating Systems Design and Implementation*. USENIX, October 2000.

[4] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, April 2003.

[5] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop*, July 2006.

[6] Lap chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. 22st Annual Computer Security Applications Conference (ACSAC 2006)*, December 2006.

[7] Danny Coward and Yutaka Yoshida. Java servlet specification, version 2.4, November 2003. JSR-000154.

[8] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7), July 1977.

[9] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[10] T. Garfinkel, B. Bfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine based platform for trusted computing. In *Proc. 19th ACM Symp. on Operating System Principles (SOSP)*, 2003.

[11] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.

[12] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proc. 1998 USENIX Annual Technical Conference*, June 1998.

[13] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[14] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proc. 13th International Conference on World Wide Web*. ACM Press, 2004.

[15] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy*, May 2006.

[16] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, October 1991. *Operating System Review*, 253(5).

[17] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, 2000.

[18] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proc. 18th IEEE Computer Security Foundations Workshop*, 2005.

[19] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. 14th USENIX Security Symposium (USENIX'05)*, pages 271–286, August 2005.

[20] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[21] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, January 1999.

[22] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[23] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proc. 17th IEEE Computer Security Foundations Workshop*, June 2004.

[24] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release, at http://www.cs.cornell.edu/jif, July 2001–.

[25] A. Nguyen-Tuong, S. Guarneri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th International Information Security Conference*, pages 372–382, May 2005.

[26] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[27] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[28] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[29] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, June 2005.

[30] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, pages 1–15, October 2005.

[31] Geoffrey Smith. A new type system for secure information flow. In *Proc. 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2001.

[32] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 202–216, July 2006.

[33] Symantec internet security threat report, volume IX. Symantec Corporation, March 2006.

[34] Trusted Computing Group. *TCG TPM Specification Version 1.2*

Revision 94, March 2006.

[35] Stephen Tse and Steve Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. 14th European Symposium on Programming*, 2005.

[36] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development*, 1997.

[37] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Conference*, July 2006.

[38] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, August 2006.

[39] Wei Xu, V.N. Venkatakrishnan, R. Sekar, and I.V. Ramakrishnan. A framework for building privacy-conscious composite web services. In *4th IEEE International Conference on Web Services (ICWS'06)*, September 2006.

[40] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[41] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, November 2006.

[42] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

A Downgrading in case studies

These tables describe the case studies' functional downgrades.

CDIS application

Description	Category
Error composing message. If an error is made when composing a message (e.g., leaving Subject field empty), the user is sent back to message composition. Downgrading this information flow reveals very little about the message data.	Application
Message approval. When a reviewer approves a message, he downgrades his confidentiality restriction. Once all reviewers have approved the message, the recipient may view it.	Application
Database access. Access to the database is done with the authority of the principal `CDISApp`. There are 11 functional downgrades for database accesses, releasing info from `CDISApp` to the user.	Access control
Delegation to `CDISRoot`. All users delegate authority to a root user for the CDIS application, `CDISRoot`, to perform operations that affect all users. This delegation requires user endorsement.	Application

User library

Description	Category
Unsuccessful login. When user enters a password on the login page, he learns if the password was correct. If incorrect, the user is returned to the login page with an error message. This information release about the password is acceptable.	Application
Successful login. When the user logs in successfully, he learns that the password was correct. This information flow is secure.	Application
Delegation to session principal. When the user logs in, he delegates authority to the session principal, using a closure. The decision to authorize the delegation closure must be declassified.	Application
Delegation to session principal. Delegating authority from a newly logged in user to the session principal requires the trust of the user, and thus an endorsement.	Application
Retrieving users from the database. When selecting one or more users, info must be retrieved from the database, and returned to the caller of the Select User(s) page. This transfer requires a total of 3 functional downgrades during user selection.	Access control
Error selecting user(s). A user making an error on the Select User(s) page (e.g., no user id entered) is returned to the Select User(s) page. As this page is a reusable component, its label is set conservatively. A declassification is needed for the error message, from the conservative label to the actual label used for a given page invocation.	Imprecision

Calendar application

Description	Category
Update session state with date to display. The display date must be trusted by the session principal. The date input by the user is trusted by the user, but must be endorsed by the session principal before it's stored in session state.	Access control
Update session state with which user's calendar to display. Similarly, the user selects a user's calendar to display. This downgrade ensures that the session principal authority is required to update session state.	Access control
Fresh id for new event. A new event requires a fresh unique id. The unique id may act as a covert channel, revealing info about the order in which events are created. Since ids are generated randomly, downgrading the fresh id is secure.	Application
Update and retrieve info from database. When info needs to be updated in the database (e.g., edit an event) or retrieved (e.g., fetch user details, or events) information must be transferred between the current user and the application principal `CalApp`. There are 10 such functional downgrades, for different database accesses.	Access control
Go to View/Edit Event page. An event's name is displayed as a hyperlink to the View Event or Edit Event page (depending on user's permissions). Since the link contains the event's name, the info gained by invoking View/Edit Event action is at least as restrictive as the event detail's label. This reveals little about *which* event is being viewed/edited.	Application
Error editing event. The user who makes an error editing an event (e.g., end time before start) is sent back to the Edit Event page. Like the "Go to View/Edit Event" downgrade, this reveals little about the data input.	Application
Changing attendees or viewers of an event. When the user edits an event and changes the attendees or viewers of an event, the labels to enforce on the event time and details change. This requires a downgrade.	Application
Delegation to `CalRoot`. All users delegate their authority to a root user for the calendar application, `CalRoot`, whose authority is needed to perform operations that affect all users. This requires an endorsement from each user.	Application