Improving Application Security with Data Flow Assertions

Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
Massachusetts Institute of Technology – Computer Science and Artificial Intelligence Laboratory

ABSTRACT

RESIN is a new language runtime that helps prevent security vulnerabilities, by allowing programmers to specify application-level data flow assertions. RESIN provides *policy objects*, which programmers use to specify assertion code and metadata; *data tracking*, which allows programmers to associate assertions with application data, and to keep track of assertions as the data flow through the application; and *filter objects*, which programmers use to define data flow boundaries at which assertions are checked. RESIN's runtime checks data flow assertions by propagating policy objects along with data, as that data moves through the application, and then invoking filter objects when data crosses a data flow boundary, such as when writing data to the network or a file.

Using RESIN, Web application programmers can prevent a range of problems, from SQL injection and cross-site scripting, to inadvertent password disclosure and missing access control checks. Adding a RESIN assertion to an application requires few changes to the existing application code, and an assertion can reuse existing code and data structures. For instance, 23 lines of code detect and prevent three previously-unknown missing access control vulnerabilities in phpBB, a popular Web forum application. Other assertions comprising tens of lines of code prevent a range of vulnerabilities in Python and PHP applications. A prototype of RESIN incurs a 33% CPU overhead running the HotCRP conference management application.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Verification—*Assertion checkers*; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Languages, Design

1. INTRODUCTION

Software developers often have a plan for correct data flow within their applications. For example, a user u's password may flow out of a Web site only via an email to user u's email address. As another example, user inputs must always flow through a sanitizing function before flowing into a SQL query or HTML, to avoid SQL injection or cross-site scripting vulnerabilities. Unfortunately, today these plans are implemented implicitly: programmers try to insert code in *all* the appropriate places to ensure correct flow, but it is easy to miss some, which can lead to exploits. For example, one popular Web application, phpMyAdmin [38], requires sanitizing user input in 1,409 places. Not surprisingly, phpMyAdmin has suffered 60 vulnerabilities because some of these calls were forgotten [41].

This paper presents RESIN, a system that allows programmers to make their plan for correct data flow explicit using *data flow assertions*. Programmers can write a data flow assertion in one place to capture the application's high-level data flow invariant, and RESIN checks the assertion in all relevant places, even places where the programmer might have otherwise forgotten to check.

RESIN operates within a language runtime, such as the Python or PHP interpreter. RESIN tracks application data as it flows through the application, and checks data flow assertions on every executed path. RESIN uses runtime mechanisms because they can capture dynamic properties, like user-defined access control lists, while integration with the language allows programmers to reuse the application's existing code in an assertion. RESIN is designed to help programmers gain confidence in the correctness of their application, and is not designed to handle malicious code.

A key challenge facing RESIN is knowing when to verify a data flow assertion. Consider the assertion that a user's password can flow only to the user herself. There are many different ways that an adversary might violate this assertion, and extract someone's password from the system. The adversary might trick the application into emailing the password; the adversary might use a SQL injection attack to query the passwords from the database; or the adversary might fetch the password file from the server using a directory traversal attack. RESIN needs to cover every one of these paths to prevent password disclosure.

A second challenge is to design a generic mechanism that makes it easy to express data flow assertions, including common assertions like cross-site scripting avoidance, as well as application-specific assertions. For example, HotCRP [26], a conference management application, has its own data flow rules relating to password disclosure and reviewer conflicts of interest, among others. Can a single assertion API allow for succinct assertions for cross-site scripting avoidance as well as HotCRP's unique data flow rules?

The final challenge is to make data flow assertions coexist with each other and with the application code. A single application may have many different data flow assertions, and it must be easy to add an additional assertion if a new data flow rule arises, without having to change existing assertions. Moreover, applications are often written by many different programmers. One programmer may work on one part of the application and lack understanding of the application's overall data flow plan. RESIN should be able to enforce data flow assertions without all the programmers being aware of the assertions.

RESIN addresses these challenges using three ideas: *policy objects*, *data tracking*, and *filter objects*. Programmers explicitly annotate data, such as strings, with policy objects, which encapsulate the assertion functionality that is specific to that data. Programmers write policy objects in the same language that the rest of the application is written in, and can reuse existing code and data struc-

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

Vulnerability	Count	Percentage
SQL injection	1176	20.4%
Cross-site scripting	805	14.0%
Denial of service	661	11.5%
Buffer overflow	550	9.5%
Directory traversal	379	6.6%
Server-side script injection	287	5.0%
Missing access checks	263	4.6%
Other vulnerabilities	1647	28.6%
Total	5768	100%

Table 1: Top CVE security vulnerabilities of 2008 [41].

Vulnerability	Vulnerable sites among those surveyed
Cross-site scripting	31.5%
Information leakage	23.3%
Predictable resource location	10.2%
SQL injection	7.9%
Insufficient access control	1.5%
HTTP response splitting	0.8%

Table 2: Top Web site vulnerabilities of 2007 [48].

tures, which simplifies writing application-specific assertions. The RESIN runtime then tracks these policy objects as the data propagates through the application. When the data is about to leave the control of RESIN, such as being sent over the network, RESIN invokes filter objects to check the data flow assertions with assistance from the data's policy objects.

We evaluate RESIN in the context of application security by showing how these three mechanisms can prevent a wide range of vulnerabilities in real Web applications, while requiring programmers to write only tens of lines of code. One application, the MoinMoin wiki [31], required only 8 lines of code to catch the same access control bugs that required 2,000 lines in Flume [28], although Flume provides stronger guarantees. HotCRP can use RESIN to uphold its data flow rules, by adding data flow assertions that control who may read a paper's reviews, and to whom HotCRP can email a password reminder. Data flow assertions also help prevent a range of other previously-unknown vulnerabilities in Python and PHP Web applications. A prototype RESIN runtime for PHP has acceptable performance overhead, amounting to 33% for HotCRP.

The contributions of this work are the idea of an application-level data flow assertion, and a technique for implementing data flow assertions using filter objects, policy objects, and data tracking. Experiments with several real applications further show that data flow assertions are concise, effective at preventing many security vulnerabilities, and incrementally deployable in existing applications.

The rest of the paper is organized as follows. The next section discusses the specific goals and motivation for RESIN. Section 3 presents the design of the RESIN runtime, and Section 4 describes our implementation. Section 5 illustrates how RESIN prevents a range of security vulnerabilities. Sections 6 and 7 present our evaluation of RESIN's ease of use, effectiveness, and performance. We discuss RESIN's limitations and future work in Section 8. Section 9 covers related work, and Section 10 concludes.

2. GOALS AND EXAMPLES

RESIN's main goal is to help programmers avoid security vulnerabilities by treating exploits as data flow violations, and then using data flow assertions to detect these violations. This section explains how faulty data flows cause vulnerabilities, and how data flow assertions can prevent those vulnerabilities.

SQL Injection and Cross-Site Scripting

SQL injection and cross-site scripting vulnerabilities are common and can affect almost any Web application. Together, they account for over a third of all reported security vulnerabilities in 2008, as seen in Table 1. These vulnerabilities result from user input data flowing into a SQL query string or HTML without first flowing through their respective sanitization functions. To avoid these vulnerabilities today, programmers insert calls to the correct sanitization function on every single path on which user input can flow to SQL or HTML. In practice this is difficult to accomplish because there are many data flow paths to keep track of, and some of them are non-intuitive. For example, in one cross-site scripting vulnerability, phpBB queried a malicious *whois* server, and then incorporated the response into HTML without first sanitizing the response. A survey of Web applications [48] summarized in Table 2 illustrates how common these bugs are with cross-site scripting affecting more than 31% of applications, and SQL injection affecting almost 8%.

If there were a tool that could enforce a data flow assertion on an entire application, a programmer could write an assertion to catch these bugs and prevent an adversary from exploiting them. For example, an assertion to prevent SQL injection exploits would verify that:

DATA FLOW ASSERTION 1. *Any user input data must flow through a sanitization function before it flows into a SQL query.*

RESIN aims to be such a tool.

Directory Traversal

Directory traversal is another common vulnerability that accounts for 6.6% of the vulnerabilities in Table 1. In a directory traversal attack, a vulnerable application allows the user to enter a file name, but neglects to limit the directories available to the user. To exploit this vulnerability, an adversary typically inserts the ".." string as part of the file name which allows the adversary to gain unauthorized access to read, or write files in the server's file system. These exploits can be viewed as faulty data flows. If the adversary reads a file without the proper authorization, the file's data is incorrectly flowing to the adversary. If the adversary writes to a file without the proper authorization, the adversary is causing an invalid flow into the file. Data flow assertions can address directory traversal vulnerabilities by enforcing data flow rules on the use of files. For example, a programmer could encode the following directory traversal assertion to protect against invalid writes:

DATA FLOW ASSERTION 2. *No data may flow into directory d unless the authenticated user has write permission for d.*

Server-Side Script Injection

Server-side script injection accounts for 5% of the vulnerabilities reported in Table 1. To exploit these vulnerabilities, an adversary uploads code to the server and then fools the application into running that code. For instance, many PHP applications load script code for different visual themes at runtime, by having the user specify the file name for their desired theme. An adversary can exploit this by uploading a file with the desired code onto the server (many applications allow uploading images or attachments), and then supplying the name of that file as the theme to load.

Even if the application is careful to not include user-supplied file names, a more subtle problem can occur. If an adversary uploads a file with a `.php` extension, the Web server may allow the adversary to directly execute that file's contents by simply issuing an HTTP

request for that file. Avoiding such problems requires coordination between many parts of the application, and even the Web server, to understand which file extensions are "dangerous". This attack can be viewed as a faulty data flow and could be addressed by the following data flow assertion:

DATA FLOW ASSERTION 3. *The interpreter may not interpret any user-supplied code.*

Access Control

Insufficient access control can also be viewed as a data flow violation. These vulnerabilities allow an adversary to read data without proper authorization and make up 4.6% of the vulnerabilities reported in 2008. For example, a missing access control check in MoinMoin wiki allowed a user to read any wiki page, even if the page's access control list (ACL) did not permit the user to read that page [46]. Like the previous vulnerabilities, this data leak can be viewed as a data flow violation; the wiki page is flowing to a user who lacks permission to receive the page. This vulnerability could be addressed with the data flow assertion:

DATA FLOW ASSERTION 4. *Wiki page p may flow out of the system only to a user on p's ACL.*

Insufficient access control is particularly challenging to address because access control rules are often unique to the application. For example, MoinMoin's ACL rules differ from HotCRP's access control rules, which ensure that only paper authors and program committee (PC) members may read paper reviews, and that PC members may not view a paper's authors if the author list is anonymous. Ideally, a data flow assertion could take advantage of the code and data structures that an application already uses to implement its access control checks.

Password Disclosure

Another example of a specific access control vulnerability is a password disclosure vulnerability that was discovered in HotCRP; we use this bug as a running example for the rest of this paper. This bug was a result of two separate features, as follows.

First, a HotCRP user can ask HotCRP to send a password reminder email to the user's email address, in case the user forgets the password. HotCRP makes sure to send the email only to the email address of the account holder as stored in the server. The second feature is an *email preview* mode, in which the site administrator configures HotCRP to display email messages in the browser, rather than send them via email. In this vulnerability, an adversary asks HotCRP to send a password reminder for another HotCRP user (the victim) while HotCRP is in *email preview* mode. HotCRP will display the content of the password reminder email in the adversary's browser, instead of sending the password to that victim's email address, thus revealing the victim's password to the adversary.

A data flow assertion could have prevented this vulnerability because the assertion would have caught the invalid password flow despite the unexpected combination of the password reminder and *email preview* mode. The assertion in this case would have been:

DATA FLOW ASSERTION 5. *User u's password may leave the system only via email to u's email address, or to the program chair.*

2.1 Threat Model

As we have shown, many vulnerabilities in today's applications can be thought of as programming errors that allow faulty data flows. Adversaries exploit these faulty data flows to bypass the application's security plan. RESIN aims to prevent adversaries from exploiting these faulty data flows by allowing programmers to explicitly specify data flow assertions, which are then checked at runtime in all places in the application.

We expect that programmers would specify data flow assertions to prevent well-known vulnerabilities shown in Table 1, as well as existing application-specific rules, such as HotCRP's rules for password disclosure or reviewer conflicts of interest. As programmers write new code, they can use data flow assertions to make sure their data is properly handled in code written by other developers, without having to look at the entire code base. Finally, as new problems are discovered, either by attackers or by programmers auditing the code, data flow assertions can be used to fix an entire class of vulnerabilities, rather than just a specific instance of the bug.

RESIN treats the entire language runtime, and application code, as part of the trusted computing base. RESIN assumes the application code is not malicious, and does not prevent an adversary from compromising the underlying language runtime or the OS. In general, a buffer overflow attack can compromise a language runtime, but buffer overflows are less of an issue for RESIN because languages like PHP and Python are not susceptible to buffer overflows.

3. DESIGN

Many of the vulnerabilities described in Section 2 can be addressed with data flow assertions, but the design of such an assertion system requires solutions to a number of challenges. First, the system must enforce assertions on the many communication channels available to the application. Second, the system must provide a convenient API in which programmers can express many different types of data flow assertions. Finally, the system must handle several assertions in a single application gracefully; it should be easy to add new assertions, and doing so should not disrupt existing assertions. This section describes how RESIN addresses these design challenges, beginning with an example of how a data flow assertion prevents the HotCRP password disclosure vulnerability described in Section 2.

3.1 Design Overview

To illustrate the high-level design of RESIN and what a programmer must do to implement a data flow assertion, this section describes how a programmer would implement Data Flow Assertion 5, the HotCRP password assertion, in RESIN. This example does not use all of RESIN's features, but it does show RESIN's main concepts.

Conceptually, the programmer needs to restrict the flow of passwords. However, passwords are handled by a number of modules in HotCRP, including the authentication code and code that formats and sends email messages. Thus, the programmer must confine passwords by defining a data flow boundary that surrounds the entire application. Then the programmer allows a password to exit that boundary only if that password is flowing to the owner via email, or to the program chair. Finally, the programmer marks the passwords as sensitive so that the boundary can identify which data contains password information, and writes a small amount of assertion checking code.

RESIN provides three mechanisms that help the programmer implement such an assertion (see Figure 1):

- Programmers use *filter objects* to define data flow boundaries. A filter object interposes on an input/output channel or a function call interface.
- Programmers explicitly annotate sensitive data with *policy objects*. A policy object can contain code and metadata for checking assertions.
- Programmers rely on RESIN's runtime to perform *data tracking* to propagate policy objects along with sensitive data when the application copies that data within the system.

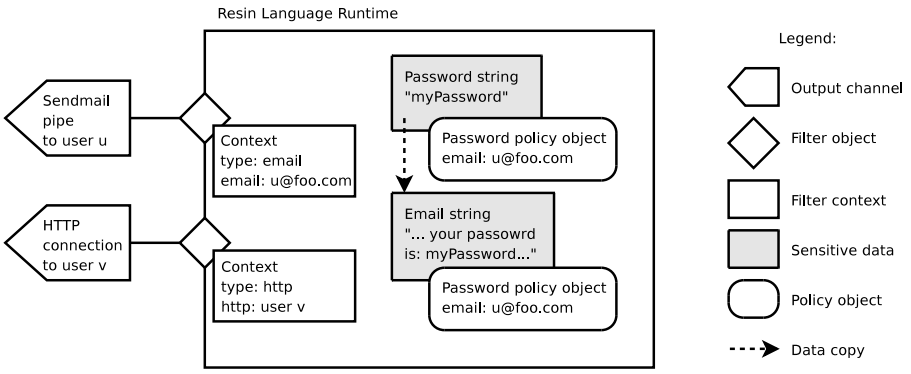

Figure 1: Overview of the HotCRP password data flow assertion in RESIN.

```
class PasswordPolicy extends Policy {
  private $email;
  function __construct($email) {
    $this->email = $email;
  }
  function export_check($context) {
    if ($context['type'] == 'email' &&
        $context['email'] == $this->email) return;
    global $Me;
    if ($context['type'] == 'http' &&
        $Me->privChair) return;
    throw new Exception('unauthorized disclosure');
  }
}

policy_add($password, new PasswordPolicy('u@foo.com'));
```

Figure 2: Simplified PHP code for defining the HotCRP password policy class and annotating the password data. This policy only allows a password to be disclosed to the user's own email address or to the program chair.

RESIN by default defines a data flow boundary around the language runtime using filter objects that cover all I/O channels, including pipes and sockets. By default, RESIN also annotates some of these default filter objects with *context* metadata that describes the specific filter object. For example, RESIN annotates each filter object connected to an outgoing email channel with the email's recipient address. The default set of filters and contexts defining the boundary are appropriate for the HotCRP password assertion, so the programmer need not define them manually.

In order for RESIN to track the passwords, the programmer must annotate each password with a policy object, which is a language-level object that contains fields and methods. In this assertion, a user's password will have a policy object that contains a copy of the user's email address so that the assertion can determine which email address may receive the password data. When the user first sets their password, the programmer copies the user's email address from the current session information into the password's policy object.

The programmer also writes the code that checks the assertion, in a method called *export_check* within the password policy object's class definition. Figure 2 shows the code the programmer must write to implement this data flow assertion, including the policy object's class definition and the code that annotates a password with a policy object. The policy object also shows how an assertion can benefit from the application's data structures; this assertion uses an existing flag, *$Me->privChair*, to determine whether the current user is the program chair.

Once a password has the appropriate policy object, RESIN's data tracking propagates that policy object along with the password data; when the application copies or moves the data within the system, the policy goes along with the password data. For example, after HotCRP composes the email content using the password data, the email content will also have the password policy annotation (as shown in Figure 1).

RESIN enforces the assertion by making each filter object call *export_check* on the policy object of any data that flows through the filter. The filter object passes its context as an argument to *export_check* to provide details about the specific I/O channel (e.g., the email's recipient).

This assertion catches HotCRP's faulty data flow before it can leak a password. When HotCRP tries to send the password data over an HTTP connection, the connection's filter object invokes the *export_check* method on the password's policy object. The *export_check* code observes that HotCRP is incorrectly trying to send the password over an HTTP connection, and throws an exception which prevents HotCRP from sending the password to the adversary. This solution works for all disclosure paths through the code because RESIN's default boundary controls all output channels; HotCRP cannot reveal the password without traversing a filter object.

This example is just one way to implement the password data flow assertion, and there may be other ways. For example, the programmer could implement the assertion checking code in the filter objects rather than the password's policy object. However, modifying filter objects is less attractive because the programmer would need to modify every filter object that a password can traverse. Putting the assertion code in the policy object allows the programmer to write the assertion code in one place.

3.2 Filter Objects

A filter object, represented by a diamond in Figure 1, is a generic interposition mechanism that application programmers use to create data flow boundaries around their applications. An application can associate a filter object with a function call interface, or an I/O channel such as a file handle, socket, or pipe.

RESIN aims to support data flow assertions that are specific to an application, so in RESIN, a programmer implements a filter object as a language-level object in the same language as the rest of the application. This allows the programmer to reuse the application's code and data structures, and allows for better integration with applications.

When an application sends data across a channel guarded by a filter object, RESIN invokes a method in that filter object with the

Function	Caller	Semantics
filter::filter_read(*data*, *offset*)	Runtime	Invoked when data comes in through a data flow boundary, and can assign initial policies for *data*; e.g., by de-serializing from persistent storage.
filter::filter_write(*data*, *offset*)	Runtime	Invoked when data is exported through a data flow boundary; typically invokes assertion checks or serializes policy objects to persistent storage.
filter::filter_func(*args*)	Runtime	Checks and/or proxies a function call.
policy::export_check(*context*)	Filter object	Checks if data flow assertion allows exporting data, and throws exception if not; *context* provides information about the data flow boundary.
policy::merge(*policy_object_set*)	Runtime	Returns set of policies (typically zero or one) that should apply to merging of data tagged with this policy and data tagged with *policy_object_set*.
policy_add(*data*, *policy*)	Programmer	Adds *policy* to *data*'s policy set.
policy_remove(*data*, *policy*)	Programmer	Removes *policy* from *data*'s policy set.
policy_get(*data*)	Programmer	Returns set of policies associated with *data*.

Table 3: The RESIN API. A::B(args) denotes method B of an object of type A. Not shown is the API used by the programmer to specify and access filter objects for different data flow boundaries.

```
class DefaultFilter(Filter):
  def __init__(self): self.context = {}
  def filter_write(self, buf):
    for p in policy_get(buf):
      if hasattr(p, 'export_check'):
        p.export_check(self.context)
    return buf
```

Figure 3: Python code for the default filter for sockets.

data as an argument. If the interposition point is an I/O channel, RESIN will invoke either *filter_read* or *filter_write*; for function calls, RESIN will invoke *filter_func* (see Table 3). *Filter_read* and *filter_write* can check or alter the in-transit data. *Filter_func* can check or alter the function's arguments and return value.

For example, in an HTTP splitting attack, the adversary inserts an extra CR-LF-CR-LF delimiter into the HTTP output to confuse browsers into thinking there are two HTTP responses. To thwart this type of attack, the application programmer could write a filter object that scans for unexpected CR-LF-CR-LF character sequences, and then attach this filter to the HTTP output channel. As a second example, a function that encrypts data is a natural data flow boundary. A programmer may choose to attach a filter object to the encryption function that removes policy objects for confidentiality assertions such as the *PasswordPolicy* from Section 3.1.

3.2.1 Default Filter Objects

RESIN pre-defines default filter objects on all I/O channels into and out of the runtime, including sockets, pipes, files, HTTP output, email, SQL, and code import. Since these default filter objects are at the edge of the runtime, data can flow freely within the application and the default filters will only check assertions before making program output visible to the outside world. This boundary should be suitable for many assertions because it surrounds the entire application. The default boundary also helps programmers avoid accidentally overlooking an I/O channel, which would result in an incomplete boundary that would not cover all possible flows.

The default filter objects check the in-transit data for policies, as shown in Figure 3. If a filter finds a policy that has an *export_check* method, the filter invokes the policy's *export_check* method. As described in Section 3.1, *export_check* typically checks the assertion and throws an exception if the flow would violate the assertion.

Since the policy's *export_check* method may need additional information about the filter's specific I/O channel or function call to check the assertion, RESIN attaches context information, in the form of a hash table, to some of the default filters as described in Section 3.1. RESIN also allows the application to add its own key-value pairs to the context hash table of default filter objects.

The context key-value pairs are likely specific to the I/O channel or function call that the filter guards, and the default filter passes the context hash table as an argument to *export_check*. In the HotCRP example, the context for a sendmail pipe contains the recipient of the email (as shown in Figure 1).

3.2.2 Importing Code

RESIN treats the interpreter's execution of script code as another data flow channel, with its own filter object. This allows programmers to interpose on all code flowing into the interpreter, and ensure that such code came from an approved source. This can prevent server-side script injection attacks, where an adversary tricks the interpreter into executing adversary-provided script code.

3.2.3 Write Access Control

In addition to runtime boundaries, RESIN also permits an application to place filter objects on persistent files to control write access, because data tracking alone cannot prevent modifications. In particular, RESIN allows programmers to specify access control checks for files and directories in a persistent filter object that's stored in the extended attributes of a specific file or directory. The runtime automatically invokes this persistent filter object when data flows into or out of that file, or modifies that directory (such as creating, deleting, or renaming files). This programmer-specified filter object can check whether the current user is authorized to access that file or directory. These persistent filter objects associated with a specific file or directory are separate from the filter objects associated by default with every file's I/O channel.

3.3 Policy Objects

Like a filter object, a policy object is a language-level object, and can reuse the application's existing code and data structures. A policy object can contain fields and methods that work in concert with filter objects; policy objects are represented by the rounded rectangles in Figure 1.

To work with default filter objects, a policy object should have an *export_check* method as shown in Table 3. As mentioned earlier, default filter objects invoke *export_check* when data with a policy passes through a filter, so *export_check* is where programmers implement an assertion check for use with default filters. If the assertion fails, *export_check* should throw an exception so that RESIN can prevent the faulty data flow.

The main distinction between policy objects and filter objects is that a policy object is specific to data, and a filter object is specific to a channel. A policy object would contain data specific metadata and code; for example, the HotCRP password policy contains the email address of the password's account holder. A filter object would contain channel specific metadata; for example, the email filter object contains the recipient's email address.

Even though RESIN allows programmers to write many different filter and policy objects, the interface between all filters and policies remains largely the same, if the application uses *export_check*. This limits the complexity of adding or changing filters and policies, because each policy object need not know about all possible filter objects, and each filter object need not know about all possible policy objects (although this does not preclude the programmer from implementing special cases for certain policy-filter pairs).

3.4 Data Tracking

RESIN keeps track of policy objects associated with data. The programmer attaches a policy object to a datum—a primitive data element such as an integer or a character in a string (although it is common to assign the same policy to all characters in a string). The RESIN runtime then propagates that policy object along with the data, as the application code moves or copies the data.

To attach a policy object to data, the programmer uses the *policy_add* function listed in Table 3. Since an application may have multiple data flow assertions, a single datum may have multiple policy objects, all contained in the datum's *policy set*.

RESIN propagates policies in a fine grained manner. For example, if an application concatenates the string "foo" (with policy p_1), and "bar" (with policy p_2), then in the resulting string "foobar", the first three characters will have only policy p_1 and the last three characters will have only p_2. If the programmer then takes the first three characters of the combined string, the resulting substring "foo" will only have policy p_1. Tracking data at the character level minimizes interference between different data flow assertions, whose data may be combined in the same string, and minimizes unintended policy propagation, when marshalling and un-marshalling data structures. For example, in the HotCRP password reminder email message, only the characters comprising the user's password have the password policy object. The rest of the string is not associated with the password policy object, and can potentially be manipulated and sent over the network without worrying about the password policy (assuming there are no other policies).

RESIN tracks explicit data flows such as variable assignment; most of the bugs we encountered, including all the bugs described in Sections 2 and 6, use explicit data flows. However, some data flows are implicit. One example is a control flow channel, such as when a value that has a policy object influences a conditional branch, which then changes the program's output. Another example of an implicit flow is through data structure layout; an application can store data in an array using a particular order. RESIN does not track this order information, and a programmer cannot attach a policy object to the array's order. These implicit data flows are sometimes surprising and difficult to understand, and RESIN does not track them. If the programmer wants to specify data flow assertions about such data, the programmer must first make this data explicit, and only then attach a policy to it.

3.4.1 Persistent Policies

RESIN only tracks data flow inside the language runtime, and checks assertions at the runtime boundary, because it cannot control what happens to the data after it leaves the runtime. However, many applications store data persistently in file systems and databases.

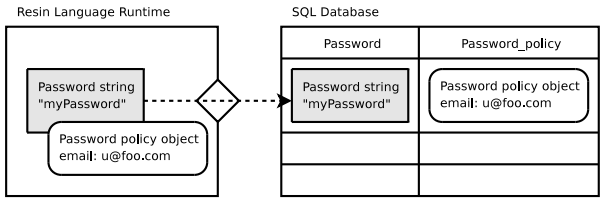

Figure 4: Saving persistent policies to a SQL database for HotCRP passwords. Uses symbols from Figure 1.

For example, HotCRP stores user passwords in a SQL database. It can be inconvenient and error-prone for the programmer to manually save metadata about the password's policy object when saving it to the database, and then reconstruct the policy object when reading the password later.

To help programmers of such applications, RESIN transparently tracks data flows to and from persistent storage. RESIN's default filter objects serialize policy objects to persistent files and database storage when data is written out, and de-serializes the policy objects when data is read back into the runtime.

For data going to a file, the file's default filter object serializes the data's policy objects into the file's extended attributes. Whenever the application reads data from the file, the filter reads the serialized policy from the file's extended attributes, and associates it with the newly-read data. RESIN tracks policies for file data at byte-level granularity, as it does for strings.

RESIN also serializes policies for data stored in a SQL database, as shown in Figure 4. RESIN accomplishes this by attaching a default filter object to the function used to issue SQL queries, and using that filter to rewrite queries and results. For a *CREATE TABLE* query, the filter adds an additional policy column to store the serialized policy for each data column. For a query that writes to the database, the filter augments the query to store the serialized policy of each cell's value into the corresponding policy column. Last, for a query that fetches data, the filter augments the query to fetch the corresponding policy column, and associates each de-serialized policy object with the corresponding data cell in the resulting set of rows.

Storing policies persistently also helps other programs, such as the Web server, to check invariants on file data. For example, if an application accidentally stores passwords in a world-readable file, and an adversary tries to fetch that file via HTTP, a RESIN-aware Web server will invoke the file's policy objects before transmitting the file, fail the *export_check*, and prevent password disclosure.

RESIN only serializes the class name and data fields of a policy object. This allows programmers to change the code for a policy class to evolve its behavior over time. For example, a programmer could change the *export_check* method of HotCRP's password policy object to disallow disclosure to the program chair without changing the existing persistent policy objects. However, if the application needs to change the fields of a persistent policy, the programmer will need to update the persistent policies, much like database schema migration.

3.4.2 Merging Policies

RESIN uses character-level tracking to avoid having to merge policies when individual data elements are propagated verbatim, such as through concatenation or taking a substring. Unfortunately, merging is inevitable in some cases, such as when string characters with different policies are converted to integer values and added up to compute a checksum. In many situations, such low-level data transformation corresponds to a boundary, such as encryption or

hashing, and would be a good fit for an application-specific filter object. However, relying on the programmer to insert filter objects in all such places would be error-prone, and RESIN provides a safety net by merging policy objects in the absence of any explicit actions by the programmer.

By default, RESIN takes the union of policy objects of source operands, and attaches them to the resulting datum. The *union* strategy is suitable for some data flow assertions. For example, an assertion that tracks user-supplied inputs by marking them with a *UserData* policy would like to label the result as *UserData* if any source operand was labeled as such. In contrast, the *intersection* strategy may be applicable to other policies. An assertion that tracks data authenticity by marking data with an *AuthenticData* policy would like to label the result as *AuthenticData* only if all source operands were labeled that way.

Because different policies may have different notions of a safe merge strategy, RESIN allows a policy object to override the *merge* method shown in Table 3. When application code merges two data elements, RESIN invokes the *merge* method on each policy of each source operand, passing in the entire policy set of the other operand as the argument. The *merge* method returns a set of policy objects that it wants associated with the new datum, or throws an exception if this policy should not be merged. The *merge* method can consult the set of policies associated with the other operand to implement either the *union* or *intersection* strategies. The RESIN runtime then labels the resulting datum with the union of all policies returned by all *merge* methods.

4. IMPLEMENTATION

We have implemented two RESIN prototypes, one in the PHP runtime, and the other in Python. At a high-level, RESIN requires the addition of a pointer, that points to a set of policy objects, to the runtime's internal representation of a datum. For example, in PHP, the additional pointer resides in the *zval* object for strings and numbers. For strings, each policy object contains a character range for which the policy applies. When copying or moving data from one primitive object to another, the language runtime copies the policy set from the source to the destination, and modifies the character ranges if necessary. When merging individual data elements, the runtime invokes the policies' merge functions.

The PHP prototype involved 5,944 lines of code. The largest module is the SQL parsing and translation mechanism at about 2,600 lines. The core data structures and related functions make up about 1,100 lines. Most of the remaining 2,200 lines are spent propagating and merging policy objects. Adding propagation to the core PHP language required changes to its virtual machine opcode handlers, such as variable assignment, addition, and string concatenation. In addition, PHP implements many of its library functions, such as *substr* and *printf*, in C, which are outside of PHP's virtual machine and require additional propagation code.

To allow the Web server to check persistent policies for file data, as described in Section 3.4.1, we modified the *mod_php* Apache module to de-serialize and invoke policy objects for all static files it serves. Doing so required modifying 49 lines of code in *mod_php*.

The Python prototype only involved 681 lines of code; this is fewer than the PHP prototype for two reasons. First, our Python prototype does not implement all the RESIN features; it lacks character-level data tracking, persistent policy storage in SQL databases, and Apache static file support. Second, Python uses fewer C libraries, so it required little propagation code beyond the opcode handlers.

5. APPLYING RESIN

```
def process_client(client_sock):
  req = parse_request(client_sock)
  client_sock.__filter.context['user'] = req.user
  ... process req ...

class PagePolicy(Policy):
  def __init__(self, acl): self.acl = acl
  def export_check(self, context):
    if not self.acl.may(context['user'], 'read'):
      raise Exception("insufficient access")

class Page:
  def update_body(self, text):
    text = policy_add(text, PagePolicy(self.getACL()))
    ... write text to page's file ...
```

Figure 5: Python code for a data flow assertion that checks read access control in MoinMoin. The *process_client* and *update_body* functions are simplified versions of MoinMoin equivalents.

RESIN's main goal is to allow programmers to avoid security vulnerabilities by specifying data flow assertions. Section 3.1 already showed how a programmer can implement a data flow assertion that prevents password disclosure in HotCRP. This section shows how a programmer would implement data flow assertions in RESIN for a number of other vulnerabilities and applications.

The following examples use the syntax described in Table 3. Additionally, these examples use `sock.__filter` to access a socket's filter object, and in the Python code, *policy_add* and *policy_remove* return a new string with the same contents but a different policy set, because Python strings are immutable.

5.1 Access Control Checks

As mentioned in Section 2, RESIN aims to address missing access control checks. To illustrate how a programmer would use RESIN to verify access control checks, this section provides an example implementation of Data Flow Assertion 4, the assertion that verifies MoinMoin wiki's read ACL scheme (see Section 2).

The MoinMoin ACL assertion prevents a wiki page from flowing to a user that's not on the page's ACL. One way for a programmer to implement this assertion in RESIN is to:

1. annotate HTTP output channels with context that identifies the user on the other end of the channel;
2. define a *PagePolicy* object that contains an ACL;
3. implement an *export_check* method in *PagePolicy* that matches the output channel against the *PagePolicy*'s ACL;
4. attach a *PagePolicy* to the data in each wiki page.

Figure 5 shows all the code necessary for this implementation. The *process_client* function annotates each HTTP connection's context with the current user, after parsing the user's request and credentials. *PagePolicy* contains a copy of the ACL, and implements *export_check*. The *update_body* method creates a *PagePolicy* object and attaches it to the page's data before saving the page to the file system. One reason why the *PagePolicy* is short is that it reuses existing application code to perform the access control check.

This example assertion illustrates the use of persistent policies. The *update_body* function associates a *PagePolicy* with the contents of a page immediately before writing the page to a file. As the page data flows to the file, the default filter object serializes the *PagePolicy* object, including the access control list, to the file system. When MoinMoin reads this file later, the default filter will de-serialize the *PagePolicy* and attach it to the page data in the runtime, so that RESIN will automatically enforce the same access control policy.

```
class CodeApproval extends Policy {
  function export_check($context) {}
}

function make_file_executable($f) {
  $code = file_get_contents($f);
  policy_add($code, new CodeApproval());
  file_put_contents($f, $code);
}

class InterpreterFilter extends Filter {
  function filter_read($buf) {
    foreach (policy_get($buf) as $p)
      if ($p instanceof CodeApproval)
        return $buf;
    throw new Exception('not executable');
  }
}
```

Figure 6: Simplified PHP code for a data flow assertion that catches server-side script injection. In the actual implementation, *filter_read* verifies that each character in *$buf* has the *CodeApproval* policy.

In this implementation, the *update_body* function provides a single place where MoinMoin saves the page to the file system, and a thus a single place to attach the *PagePolicy*. If, however, MoinMoin had multiple code paths that stored pages in the file system, the programmer could assign the policy to the page contents earlier, perhaps directly to the CGI input variables.

In addition to read access checks, the programmer can also define a data flow assertion that verifies write access checks. MoinMoin's write ACLs imply the assertion: *data may flow into wiki page p only if the user is on p's write ACL*. MoinMoin stores a wiki page as a directory that contains each version of the page as a separate file. The programmer can implement this assertion by creating a filter class that verifies the write ACL against the current user, and then attaching filter instances to the files and directory that represent a wiki page. The filters restrict the modification of existing versions, and also the creation of new versions based on the page's ACL.

5.2 Server-Side Script Injection

Another class of vulnerabilities that RESIN aims to address is server-side script injection, as described in Section 2, which can be addressed with Data Flow Assertion 3. One way for the programmer to implement this assertion is to:

1. define an empty *CodeApproval* policy object;[1]
2. annotate application code and libraries with *CodeApproval* policy objects;
3. change the interpreter's default input filter (see Section 3.2.2) to require a *CodeApproval* policy on all imported code.

This data flow assertion instructs RESIN to limit what code the interpreter may use. Figure 6 lists the code for implementing this assertion. When installing an application, the developer tags the application code and system libraries with a persistent *CodeApproval* policy object using *make_file_executable*. The *filter_read* method only allows code with a *CodeApproval* policy object to pass, ensuring that code from an adversary which would lack the *CodeApproval* policy, will not be executed, whether through *include* statements, *eval*, or direct HTTP requests.

The programmer must override the interpreter's filter in a global configuration file, to ensure the filter is set before any other code executes; PHP's *auto_prepend_file* option is one way to do this. If, instead, the application set the filter at the beginning of the application's own code, adversaries could bypass the check if they are able to upload and run their own `.php` files.

This example illustrates the need for programmer-specified filter objects in addition to programmer-specified context for default filters. The default filter calls *export_check* on all the policies that pass through, but the default filter always permits data that has no policy. The filter in this script injection assertion requires that data have a *CodeApproval* policy, and reject data that does not.

5.3 SQL Injection and Cross-Site Scripting

As mentioned in Section 2, the two most popular attack vectors in Web applications today are SQL injection and cross-site scripting. This section presents two different strategies for using RESIN to address these vulnerabilities.

To implement the first strategy, the programmer:

1. defines two policy object classes: *UntrustedData* and *SQLSanitized*;
2. annotates untrusted input data with an *UntrustedData* policy;
3. changes the existing SQL sanitization function to attach a *SQLSanitized* object to the freshly sanitized data;
4. changes the SQL filter object to check the policy objects on each SQL query. If the query contains any characters that have the *UntrustedData* policy, but not the *SQLSanitized* policy, the filter will throw an exception and refuse to forward the query to the database.

Addressing cross-site scripting is similar, except that it uses *HTMLSanitized* rather than *SQLSanitized*. This strategy catches unsanitized data because the data will lack the correct *SQLSanitized* or *HTMLSanitized* policy object. The reason for appending *SQLSanitized* and *HTMLSanitized* instead of removing *UntrustedData* is to allow the assertion to distinguish between data that may be incorporated into SQL versus HTML since they use different sanitization functions. This strategy ensures that the programmer uses the correct sanitizer (e.g., the programmer did not accidentally use SQL quoting for a string used as part of an HTML document).

The second strategy for preventing SQL injection and cross-site scripting vulnerabilities is to use the same *UntrustedData* policy from the previous strategy, but rather than appending a policy like *SQLSanitized*, the SQL filter inspects the final query and throws an exception if any characters in the query's structure (keywords, white space, and identifiers) have the *UntrustedData* policy. The HTML filter performs a similar check for *UntrustedData* on JavaScript portions of the HTML to catch cross-site scripting errors, similar to a technique used in prior work [34].

A variation on the second strategy is to change the SQL filter's tokenizer to keep contiguous bytes with the *UntrustedData* policy in the same token, and to automatically sanitize the untrusted data in transit to the SQL database. This will prevent untrusted data from affecting the command structure of the query, and likewise for the HTML tokenizer. These two variations require the addition of either tokenizing or parsing to the filter objects, but they avoid relying on trusted quoting functions.

We have experimented with both of these strategies in RESIN, and find that while the second approach requires more code for the parsers, many applications can reuse the same parsing code.

A SQL injection assertion is complementary to the other assertions we describe in this section. For instance, even if an application has a SQL injection vulnerability, and an adversary manages to execute the query `SELECT user, password FROM userdb`,

[1]The *CodeApproval* policy does not need to take the intersection of policies during merge because RESIN's character-level data tracking avoids having to merge file data.

the policy object for each password will still be de-serialized from the database, and will prevent password disclosure.

5.4 Other Attack Vectors

Finally, there are a number of other attack vectors that RESIN can help defend against. For instance, to address the HTTP response splitting attack described in Section 3.2, a developer can use a filter to reject any CR-LF-CR-LF sequences in the HTTP header that came from user input.

As Web applications use more client-side code, they also use more JSON to transport data from the server to the client. Here, much like in SQL injection, an adversary may be able to craft an input string that changes the structure of the JSON's JavaScript data structure, or worse yet, include client-side code as part of the data structure. Web applications can use RESIN's data tracking mechanisms to avoid these pitfalls as they would for SQL injection.

5.5 Application Integration

One potential concern when using RESIN is that a data flow assertion can duplicate data flow checks and security checks that already exist in an application. As a concrete example, consider HotCRP, which maintains a list of authors for each paper. If a paper submission is anonymous, HotCRP must not reveal the submission's list of authors to the PC members. HotCRP already performs this check before adding the author list to the HTML output. Adding a RESIN data flow assertion to verify read access to the author list will make HotCRP perform the access check a second time within the data flow assertion, duplicating the check that already exists.

If a programmer implements an application with RESIN in mind, the programmer can use an exception to indicate that the user may not read certain data, thereby avoiding duplicate access checks. For example, we modified the HotCRP code that displays a paper submission to always try to display the submission's author list. If the submission is anonymous, the data flow assertion raises an exception; the display code catches that exception, and then displays the string "Anonymous" instead of the author list. This avoids duplicate checks because the page generation code does not explicitly perform the access control check. However, if the application sends HTML output to the browser during a *try* block and then encounters an exception later in the *try* block, the previously released HTML might be invalid because the *try* block did not run to completion.

RESIN provides an *output buffering* mechanism to assist with this style of code. To use output buffering, the application starts a new *try* block before running HTML generation code that might throw an exception. At the start of the *try* block, the application notifies the outgoing HTML filter object to start buffering output. If the *try* block throws an exception, the corresponding *catch* block notifies the HTML filter to discard the output buffer, and potentially send alternate output in its place (such as "Anonymous" in the example). However, if the *try* block runs to completion, the *try* block notifies the HTML filter to release the data in the output buffer.

Using exceptions, instead of explicit access checks, frees the programmer from needing to know exactly which checks to invoke in every single case, because RESIN invokes the checks. Instead, programmers need to only wrap code that might fail a check with an appropriate exception handler, and specify how to present an exception to the user.

6. SECURITY EVALUATION

The main criteria for evaluating RESIN is whether it is effective at helping a programmer prevent data flow vulnerabilities. To provide a quantitative measure of RESIN's effectiveness, we focus on three areas. First, we determine how much work a programmer must do to implement an existing implicit data flow plan as an explicit data flow assertion in RESIN. We then evaluate whether each data flow assertion actually prevents typical data flow bugs, both previously-known and previously-unknown bugs. Finally, we evaluate whether a single high-level assertion can be general enough to cover both common and uncommon data flows that might violate the assertion, by testing assertions against bugs that use surprising data paths.

6.1 Programmer Effort

To determine the level of effort required for a programmer to use RESIN, we took a number of existing, off-the-shelf applications and examined some of their implicit security-related data flow plans. We then implemented a RESIN data flow assertion for each of those implicit plans. Table 4 summarizes the results, showing the applications, the number of lines of code in the application, and the number of lines of code in each data flow assertion.

The results in Table 4 show that each data flow assertion requires a small amount of code, on the order of tens of lines of code. The assertion that checks read access to author lists in HotCRP requires the most changes, 32 lines. This is more code than other assertions because our implementation issues database queries and interprets the results to perform the access check, requiring extra code. However, many of the other assertions in Table 4 reuse existing code from the application's existing security plan, and are shorter.

Table 4 also shows that the effort required to implement a data flow assertion does not grow with the size of the application. This is because implementing an assertion only requires changes where sensitive data first enters the application, and/or where data exits the system, not on every path data takes through the application; RESIN's data tracking handles those data paths. For example, the cross-site scripting assertion for phpBB is only 22 lines of code even though phpBB is 172,000 lines of code.

As a point of comparison for programmer effort, consider the MoinMoin access control scheme that appeared in the Flume evaluation [28]. MoinMoin uses ACLs to limit who can read and write a wiki page. To implement this scheme under Flume, the programmer partitions MoinMoin into a number of components, each with different privileges, and then sets up the OS to enforce the access control system using information flow control. Adapting MoinMoin to use Flume requires modifying or writing about 2,000 lines of application code. In contrast, RESIN can check the same MoinMoin access control scheme using two assertions, an eight line assertion for reading, and a 15 line assertion for writing, as shown in Table 4. Most importantly, adding these checks with RESIN requires no structural or design changes to the application.

Although Flume provides assurance against malicious server code and RESIN does not, the RESIN assertions catch the same two vulnerabilities (see Section 6.2) that Flume catches, because they do not involve binary code injection. By focusing on a weaker threat model, RESIN's lightweight and easy-to-use mechanisms provide a compelling choice for programmers that want additional security assurance without much extra effort.

6.2 Preventing Vulnerabilities

To evaluate whether RESIN's data flow assertions are capable of preventing vulnerabilities, we checked some of the assertions in Table 4 against known vulnerabilities that the assertion should be able to prevent. The results are shown in Table 4, where the number of previously-known vulnerabilities is greater than zero.

The results in Table 4 show that each RESIN assertion does prevent the vulnerabilities it aims to prevent. For example, the phpBB access control assertion prevents a known missing access control check listed in the CVE [41], and the HotCRP password protec-

Application	Lang.	App. LOC	Assertion LOC	Known vuln.	Discovered vuln.	Prevented vuln.	Vulnerability type
MIT EECS grad admissions	Python	18,500	9	0	3	3	SQL injection
MoinMoin	Python	89,600	8 15	2 0	0 0	2 0	Missing read access control checks Missing write access control checks
File Thingie file manager	PHP	3,200	19	0	1	1	Directory traversal, file access control
HotCRP	PHP	29,000	23 30 32	1 0 0	0 0 0	1 0 0	Password disclosure Missing access checks for papers Missing access checks for author list
myPHPscripts login library	PHP	425	6	1	0	1	Password disclosure
PHP Navigator	PHP	4,100	17	0	1	1	Directory traversal, file access control
phpBB	PHP	172,000	23 22	1 4	3 0	4 4	Missing access control checks Cross-site scripting
many [3, 11, 16, 23, 36]	PHP	–	12	5	0	5	Server-side script injection

Table 4: Results from using RESIN assertions to prevent previously-known and newly discovered vulnerabilities in several Web applications.

tion assertion shown in Section 3.1 prevents the password disclosure vulnerability described in Section 2. The assertion to prevent server-side script injection described in Section 5.2 prevents such vulnerabilities in five different applications [3, 11, 16, 23, 36].

Since we implemented these assertions with knowledge of the previously-known vulnerabilities, it is possible that the assertions are biased to thwart only those vulnerabilities. To address this bias, we tried to find new bugs, as an adversary would, that violate the assertions in Table 4. These results are shown in Table 4 where the number of newly discovered vulnerabilities is greater than zero.

These results show that RESIN assertions can prevent vulnerabilities, even if the programmer has no knowledge of the specific vulnerabilities when writing the assertion. For example, we implemented a generic data flow assertion to address SQL injection vulnerabilities in MIT's EECS graduate admissions system. Although the original programmers were careful to avoid most SQL injection vulnerabilities, the assertion revealed three previously-unknown SQL injection vulnerabilities in the admission committee's internal user interface.

As a second example, File Thingie and PHP Navigator are Web based file managers, and both support a feature that limits a user's write access to a particular home directory. We implemented this behavior as a write access filter as described in Section 3.2.3. Again, both applications have code in place to check directory accesses, but after a careful examination, we discovered a directory traversal vulnerability that violates the write access scheme in each application. The data flow assertions catch both of these vulnerabilities.

As a final example, phpBB implements read access controls so that only certain users can read certain forum messages. We implemented an assertion to verify this access control scheme. In addition to preventing a previously-known access control vulnerability, the assertion also prevents three previously-unknown read access violations that we discovered. These results confirm that data flow assertions in RESIN can thwart vulnerabilities, even if the programmer does not know they exist. Furthermore, these assertions likely eliminate even more vulnerabilities that we are not aware of.

The three vulnerabilities in phpBB are not in the core phpBB package, but in plugins written by third-party programmers. Large-scale projects like phpBB are a good example of the benefit of explicitly specifying data flow assertions with RESIN. Consider a situation where a new programmer starts working on an existing application like HotCRP or phpBB. There are many implicit rules that programmers must follow in hundreds of places, such as who is responsible for sanitizing what data to prevent SQL injection and cross-site scripting, and who is supposed to call the access control function. If a programmer starts writing code before understanding all of these rules, the programmer can easily introduce vulnerabilities, and this turned out to be the case in the phpBB plugins we examined. Using RESIN, one programmer can make a data flow rule explicit as an assertion and then RESIN will check that assertion for all the other programmers.

These results also provide examples of a single data flow assertion thwarting more than one instance of an entire class of vulnerabilities. For example, the single read access assertion in phpBB thwarts four specific instances of read access vulnerabilities (see Table 4). As another example, a single server-side script injection assertion that works in all PHP applications catches five different previously-known vulnerabilities in the PHP applications we tested (see Table 4). This suggests that when a programmer inevitably finds a security vulnerability and writes a RESIN assertion that addresses it, the assertion will prevent the broad class of problems that allow the vulnerability to occur in the first place, rather than only fixing the one specific instance of the problem.

6.3 Generality

To evaluate whether RESIN data flow assertions are general enough to cover the many data flow paths available to an adversary, we checked whether the assertions we wrote detect a number of data flow bugs that use surprising data flow channels.

The results indicate that a high-level RESIN assertion can detect and prevent vulnerabilities even if the vulnerability takes advantage of an unanticipated data flow path. For example, a common way for an adversary to exploit a cross-site scripting vulnerability is to enter malicious input through HTML form inputs. However, there was a cross-site scripting vulnerability in phpBB due to a more unusual data path. In this vulnerability, phpBB requests data from a *whois* server and then uses the response without sanitizing it first; an adversary exploits this vulnerability by inserting malicious JavaScript code into a *whois* record and then requesting the *whois* record via phpBB. The RESIN assertion that protects against cross-site scripting in phpBB, listed in Table 4, prevents vulnerabilities at a high-level; the assertion treats *all* external input as untrusted and makes sure that the external input data flows through a sanitizer before phpBB may use the data in HTML. This assertion is able to prevent both the more common HTML form attack as well as the less common *whois* style attack because the assertion is general enough to cover many possible data flow paths.

A second example is in the read access controls for phpBB's forum messages. The common place to check for read access is

before displaying the message to a user, but one of the read access vulnerabilities, listed in Table 4, results from a different data flow path. When a user replies to a message, phpBB includes a quotation of the original message in the reply message. In the vulnerable version, phpBB also allows a user to reply to a message even if the user lacks permission to read the message. To exploit this vulnerability, an adversary, lacking permission to read a message, replies to the message using its message ID, and then reads the content of the original message, quoted in the reply template. The RESIN assertion that checks the read access controls prevents this vulnerability because the assertion detects data flow from the original message to the adversary's browser, regardless of the path taken.

A final example comes from the two password disclosure vulnerabilities shown in Table 4. As described in Section 5, the HotCRP disclosure results from a logic bug in the email preview and the email reminder features. In contrast, the disclosure in the myPHPscripts login library [33] results from the library storing its users' passwords in a plain-text file in the same HTTP-accessible directory that contains the library's PHP files [35]. To exploit this, an adversary requests the password file with a Web browser. Despite preventing password disclosure through two different data flow paths, the assertions for password disclosure in HotCRP and myPHPscripts are very similar (the only difference is that HotCRP allows email reminders and myPHPscripts does not). This shows that a single RESIN data flow assertion can prevent attacks through a wide range of attack vectors and data paths.

7. PERFORMANCE EVALUATION

Although the main focus of RESIN is to improve application security, application developers may be hesitant to use these techniques if they impose a prohibitive performance overhead. In this section, we show that RESIN's performance is acceptable. We first measure the overhead of running HotCRP with and without the use of RESIN, and then break down the low-level costs that account for the overhead using microbenchmarks. The overall result is that a complex Web application like HotCRP incurs a 33% CPU overhead for generating a page, which is unlikely to be noticeable by end-users.

The following experiments were run on a single core of a 2.3GHz Xeon 5140 server with 4GB of memory running Linux 2.6.22. The unmodified PHP interpreter is version 5.2.5, the same version that the RESIN PHP interpreter is based on.

7.1 Application Performance

To evaluate the system-level overhead of RESIN, we compare a modified version of HotCRP running in the RESIN PHP interpreter against an unmodified version of HotCRP 2.26 running in an unmodified PHP interpreter. We measured the time to generate the Web page for a specific paper in HotCRP, including the paper's title, abstract, and author list (if not anonymized), as if a PC member requested it through a browser. The measured runtime includes the time taken to parse PHP code, recall the session state, make SQL queries, and invoke the relevant data flow assertions. In this example, RESIN invoked two assertions: one protected the paper title and abstract (and the PC member was allowed to see them), and the other protected the author list (and the PC member was not allowed to see it, due to anonymization). We used the output buffering technique from Section 5.5 to present a consistent interface even when the author list policy raised an exception. The resulting page consisted of 8.5KB of HTML.

The unmodified version of HotCRP generates the page in 66ms (15.2 requests per second) and the RESIN version uses 88ms (11.4 requests per second), averaged over 2000 trials. The performance of this benchmark is CPU limited. Despite our unoptimized RESIN prototype, its performance is likely to be adequate for many real world applications. For example, in the 30 minutes before the SOSP submission deadline in 2007, the HotCRP submission system logged only 390 user actions. Even if there were 10 page requests for each logged action (likely an overestimate), this would only average to 2.2 requests per second and a CPU utilization of 14.3% without RESIN, or 19.1% with RESIN on a single core. Adding a second CPU core doubles the throughput.

Operation	Unmodified PHP	RESIN no policy	RESIN empty policy
Assign variable	0.196 μs	0.210 μs	0.214 μs
Function call	0.598 μs	0.602 μs	0.619 μs
String concat	0.315 μs	0.340 μs	0.463 μs
Integer addition	0.224 μs	0.247 μs	0.384 μs
File open	5.60 μs	7.05 μs	18.2 μs
File read, 1KB	14.0 μs	16.6 μs	26.7 μs
File write, 1KB	57.4 μs	60.5 μs	71.7 μs
SQL SELECT	134 μs	674 μs	832 μs
SQL INSERT	64.8 μs	294 μs	508 μs
SQL DELETE	64.7 μs	114 μs	115 μs

Table 5: The average time taken to execute different operations in an unmodified PHP interpreter, a RESIN PHP interpreter without any policy, and a RESIN PHP interpreter with an empty policy.

7.2 Microbenchmarks

To determine the source of RESIN's overhead, we measured the time taken by individual operations in an unmodified PHP interpreter, and a RESIN PHP interpreter both without any policy and with an empty policy. The results of these microbenchmarks are shown in Table 5.

For operations that simply propagate policies, such as variable assignments and function calls, RESIN incurs a small absolute overhead of 4-21ns, but percentage wise, this is about a 10% overhead. This overhead is due to managing the policy set objects.

The overhead for invoking a filter object's interposition method (*filter_read*, *filter_write*, and *filter_func*) is the same as for a standard function call, except that RESIN calls the interposition method once for every call to *read* or *write*. Therefore the application programmer has some control over how much interposition overhead the application will incur. For example, the programmer can control the amount of computation the interposition method performs, and the number of times the application calls *read* and *write*.

For operations that track byte-level policies, such as string concatenation, the overhead without any policy is low (8%), but increases when a policy is present (47%). This reflects the cost of propagating byte-level policies for parts of the string at runtime as well as more calls to *malloc* and *free*. A more efficient implementation of byte-level policies could reduce these calls.

Operations that merge policies (such as integer addition, which cannot do byte-level tracking) are similarly inexpensive without a policy (10%), but are more expensive when a policy is applied (71%). This reflects the cost of invoking the programmer-supplied merge function. However, in all the data flow assertions we encountered, we did not need to apply policies to integers, so this might not have a large impact on real applications.

For file open, read, and write, RESIN adds potentially noticeable overhead, largely due to the cost of serializing, de-serializing, and invoking policies and filters stored in a file's extended attributes. Caching file policies in the runtime will likely reduce this overhead.

The INSERT operation listed in Table 5 inserts 10 cells, each into a different column, and the SELECT operation reads 10 cells, each from a different column. When there is an empty policy, each

datum has the policy. The overhead without any policy is 229–540μs (354%–403%), and that with an empty policy is 443–698μs (521%–684%). RESIN's overhead is related to the size of the query, and the number of columns that have policies; reducing the number of columns returned by a query reduces the overhead for a query. For example, a SELECT query that only requests six columns with policies takes 578μs in RESIN compared to 109μs in unmodified PHP. The DELETE operation has a lower overhead because it does not require rewriting queries or results.

RESIN's overhead for SQL operations is relatively high because it parses and translates each SQL query in order to determine the policy object for each data item that the query stores or fetches. Our current implementation performs much of the translation in a library written in PHP; we expect that converting all of it to C would offer significant speedup. Note that, even with our high overhead for SQL queries, the overall application incurs a much smaller performance overhead, such as 33% in the case of HotCRP.

8. LIMITATIONS AND FUTURE WORK

RESIN currently has a number of limitations which we plan to address in future work. First, we would like to provide better support for data integrity invariants. Instead of requiring programmers to specify what writes are allowed using filter objects, we envision using transactions to buffer database or file system changes, and checking a programmer-specified assertion before committing them.

Second, we would like to add the ability to construct internal data flow boundaries within an application. For example, an assertion could prevent clear-text passwords from flowing out of the software module that handles passwords. Attaching filter objects to function calls helps with these boundaries, but languages like PHP and Python allow code to read and write data in another module's scope as if they were global variables. An internal data flow boundary would need to address these data flow paths.

We would also like to extend RESIN to allow data flow assertions to span multiple runtimes, possibly including Javascript and SQL. We are considering a few possible approaches, including a generic representation of policy objects, or mechanisms to invoke the policy object's original runtime for performing policy checks. Also, we are interested in extending RESIN to propagate policies between machines in a distributed system similar to the way DStar [52] does with information flow labels.

We are looking for easier ways to implement data tracking in a language runtime, perhaps with OS or VMM support. Adding data tracking to PHP required modifying the interpreter in 103 locations to propagate policies; ideally, applying these techniques to new runtimes would require fewer changes. For example, it might be possible to implement RESIN without modifying the language runtime, given a suitable object-oriented system. The implementation would override all string operations to propagate policy objects, and override storage system interfaces to implement filter objects.

Finally, dynamic data tracking adds runtime overheads and presents challenges to tracking data through control flow paths. We would like to investigate whether static analysis or programmer annotations can help check RESIN-style data flow assertions at compile time.

9. RELATED WORK

RESIN makes a number of design decisions regarding how programmers specify policies and how RESIN tracks data. This section relates RESIN's design to prior work.

9.1 Policy Specification

In RESIN, programmers define a data flow assertion by writing policy objects and filter objects in the same language as the rest of the application. Previous work in policy description languages focuses on specifying policies at a higher level, to make policies easier to understand, manage [6, 12, 14], analyze [20], and specify [2]. While these policy languages do not enforce security directly, having a clearly defined policy specification allows reasoning about the security of a system, performing static analysis [17, 18], and composing policies in well-defined ways [1, 7, 43]. Doing the same in RESIN is challenging because programmers write assertions in general-purpose code. In future work, techniques like program analysis could help formalize RESIN's policies [4], to bring some of these benefits to RESIN, or to allow performance optimizations.

Lattice-based label systems [9, 10, 13, 15, 28, 32, 51] control data flow by assigning labels to objects. Expressing policies using labels can be difficult [14], and can require re-structuring applications. Once specified, labels objectively define the policy, whereas RESIN assertions require reasoning about code. For more complex policies, labels are not enough, and many applications use trusted *declassifiers* to transform labels according to application-specific rules (e.g. encryption declassifies private data). Indeed, a large part of re-structuring an application to use labels involves writing and placing declassifiers. RESIN's design can be thought of as specifying the declassifier (policy object) in the label, thus avoiding the need to place declassifiers throughout the application code.

Since RESIN programmers define their own policy and filter objects, programmers can implement data flow assertions specific to an application, such as ensuring that every string that came from one user is sanitized before being sent to another user's browser. RESIN's assertions are more extensible than specialized policy languages [19], or tools designed to find specific problems, such as SQL injection or cross-site scripting [21, 22, 29, 30, 34, 39, 44, 47, 49].

PQL [30] allows programmers to run application-specific program analyses on their code at development time, including analyses that look for data flow bugs such as SQL injection. However, PQL is limited to finding data flows that can be statically analyzed, with the help of some development-time runtime checks, and cannot find data flows that involve persistent storage. This could miss some subtle paths that an attacker might trigger at runtime, and would not prevent vulnerabilities in plug-ins added by end-users.

FABLE [40] allows programmers to customize the type system and label transformation rules, but requires the programmer to define a type system in a specialized language, and use the type system to implement the applications' data flow schemes. RESIN, on the other hand, implements data tracking orthogonal to the type system, requiring fewer code modifications, and allowing programmers to reuse existing code in their assertions.

Systems like OKWS [27] and Privman [25] enforce security by having programmers partition their application into less-privileged processes. By operating in the language runtime, RESIN's policy and filter objects track data flows and check assertions at a higher level of abstraction, avoiding the need to re-structure applications. However, RESIN cannot protect against compromised server processes.

9.2 Data Tracking

Once the assertions are in place, RESIN tracks explicit flows of application data at runtime, as it moves through the system. RESIN does not track data flows through implicit channels, such as program control flow and data structure layout, because implicit flows can be difficult to reason about, and often do not correspond to data flow plans the programmer had in mind. Implicit data flows can lead to "taint creep", or increasingly tainted program control flow, as the application executes, which can make the system difficult to use

in practice. In contrast, systems like Jif [32] track data through all channels, including program control flow, and can catch subtle bugs that leak data through these channels. By relying on a well-defined label system, Jif can also avoid runtime checks in many cases, and rely purely on compile-time static checking, which reduces runtime overhead.

RESIN's data tracking is central to its ability to implement data flow assertions that involve data movement, like SQL injection or cross-site scripting protection. Other program checkers, like Spec# [5, 6], check program invariants, but focus on checking function pre- and post-conditions and do not track data. Aspect-oriented programming (AOP) [45] provides a way to add functionality, including security checks, that cuts across many different software modules, but does not perform data tracking. However, AOP does help programmers add new code throughout an application's code base, and could be used to implement RESIN filter objects.

By tracking data flow in a language runtime, RESIN can track data at the level of existing programming abstractions—variables, I/O channels, and function calls—much like in Jif [32]. This allows programmers to use RESIN without having to restructure their applications. This differs from OS-level IFC systems [15, 28, 50, 51] which track data flowing between processes, and thus require programmers to expose data flows to the OS by explicitly partitioning their applications into many components according to the data each component should observe. On the other hand, these OS IFC systems can protect against compromised server code, whereas RESIN assumes that all application code is trusted; a compromise in the application code can bypass RESIN's assertions.

Some bug-specific tools use data tracking to prevent vulnerabilities such as cross-site scripting [24], SQL injection [34, 44], and untrusted user input [8, 37, 42]. While these tools inspired RESIN's design, they effectively hard-code the assertion to be checked into the design of the tool. As a result, they are not general enough to address application-specific data flows, and do not support data flow tracking through persistent storage. One potential advantage of these tools is that they do not require the programmer to modify their application in order to prevent well-known vulnerabilities such as SQL injection or cross-site scripting. We suspect that with RESIN, one developer could also write a general-purpose assertion that can be then applied to other applications.

10. CONCLUSION

Programmers often have a plan for correct data flow in their applications. However, today's programmers often implement their plans implicitly, which requires the programmer to insert the correct code checks in many places throughout an application. This is difficult to do in practice, and often leads to vulnerabilities.

This work takes a step towards solving this problem by introducing the idea of a data flow assertion, which allows a programmer to explicitly specify a data flow plan, and then have the language runtime check it at runtime. RESIN provides three mechanisms for implementing data flow assertions: *policy objects* associated with data, *data tracking* as data flows through an application, and *filter objects* that define data flow boundaries and control data movement.

We evaluated RESIN by adding data flow assertions to prevent security vulnerabilities in existing PHP and Python applications. Results show that data flow assertions are effective at preventing a wide range of vulnerabilities, that assertions are short and easy to write, and that assertions can be added incrementally without having to restructure existing applications. We hope these benefits will entice programmers to adopt our ideas in practice.

Acknowledgments

We thank Michael Dalton, Eddie Kohler, Butler Lampson, Robert Morris, Neha Narula, Jerry Saltzer, Jacob Strauss, Jeremy Stribling, and the anonymous reviewers for their feedback. This work was supported by Nokia Research.

References

[1] G. Ahn, X. Zhang, and W. Xu. Systematic policy analysis for high-assurance services in SELinux. In *Proc. of the 2008 POLICY Workshop*, pages 3–10, Palisades, NY, June 2008.

[2] A. H. Anderson. An introduction to the web services policy language (WSPL). In *Proc. of the 2004 POLICY Workshop*, pages 189–192, Yorktown Heights, NY, June 2004.

[3] J. Bae. Vulnerability of uploading files with multiple extensions in phpBB attachment mod. `http://seclists.org/fulldisclosure/2004/Dec/0347.html`. CVE-2004-1404.

[4] S. Barker. The next 700 access control models or a unifying meta-model? In *Proc. of the 14th ACM Symposium on Access Control Models and Technologies*, pages 187–196, Stresa, Italy, June 2009.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. of the 4th International Symposium on Formal Methods for Components and Objects*, pages 364–387, Amsterdam, The Netherlands, November 2005.

[6] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69, Marseille, France, March 2004.

[7] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the 2005 PLDI*, pages 305–314, Chicago, IL, June 2005.

[8] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proc. of the 15th CCS*, pages 39–50, Alexandria, VA, October 2008.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of the 21st SOSP*, pages 31–44, Stevenson, WA, October 2007.

[10] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. of the 16th USENIX Security Symposium*, pages 1–16, Boston, MA, August 2007.

[11] CWH Underground. Kwalbum arbitrary file upload vulnerabilities. `http://www.milw0rm.com/exploits/6664`. CVE-2008-5677.

[12] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. of the 2001 POLICY Workshop*, pages 18–38, Bristol, UK, January 2001.

[13] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[14] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *Proc. of the 3rd ACM EuroSys conference*, pages 301–313, Glasgow, UK, April 2008.

[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating

system. In *Proc. of the 20th SOSP*, pages 17–30, Brighton, UK, October 2005.

[16] Emory University. Multiple vulnerabilities in AWStats Totals. `http://userwww.service.emory.edu/~ekenda2/EMORY-2008-01.txt`. CVE-2008-3922.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the 4th OSDI*, pages 1–16, San Diego, CA, October 2000.

[18] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[19] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proc. of the 15th National Computer Security Conference*, pages 554–563, Baltimore, MD, October 1992.

[20] S. Garriss, L. Bauer, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proc. of the 13th ACM Symposium on Access Control Models and Technologies*, pages 185–194, Estes Park, CO, June 2008.

[21] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proc. of the 20th ACM International Conference on Automated Software Engineering*, pages 174–183, Long Beach, CA, November 2005.

[22] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proc. of the 14th FSE*, pages 175–185, Portland, OR, November 2006.

[23] N. Hippert. phpMyAdmin code execution vulnerability. `http://fd.the-wildcat.de/pma_e36a091q11.php`. CVE-2008-4096.

[24] S. Kasatani. Safe ERB plugin. `http://agilewebdevelopment.com/plugins/safe_erb`.

[25] D. Kilpatrick. Privman: A library for partitioning applications. In *Proc. of the 2003 USENIX Annual Technical Conference, FREENIX track*, pages 273–284, San Antonio, TX, June 2003.

[26] E. Kohler. Hot crap! In *Proc. of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.

[27] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. of the 2004 USENIX Annual Technical Conference*, pages 185–198, Boston, MA, June–July 2004.

[28] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st SOSP*, pages 321–334, Stevenson, WA, October 2007.

[29] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of the 14th USENIX Security Symposium*, pages 271–286, Baltimore, MD, August 2005.

[30] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the 2005 OOPSLA*, pages 365–383, San Diego, CA, October 2005.

[31] MoinMoin. The MoinMoin wiki engine. `http://moinmoin.wikiwikiweb.de/`.

[32] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOCS*, 9(4):410–442, October 2000.

[33] myPHPscripts.net. Login session script. `http://www.myphpscripts.net/?sid=7`. CVE-2008-5855.

[34] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proc. of the 20th IFIP International Information Security Conference*, pages 295–307, Chiba, Japan, May 2005.

[35] Osirys. myPHPscripts login session password disclosure. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-5855`. CVE-2008-5855.

[36] Osirys. wPortfolio arbitrary file upload exploit. `http://www.milw0rm.com/exploits/7165`. CVE-2008-5220.

[37] Perl.org. Perl taint mode. `http://perldoc.perl.org/perlsec.html`.

[38] phpMyAdmin. phpMyAdmin 3.1.0. `http://www.phpmyadmin.net/`.

[39] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proc. of the 8th International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, Seattle, WA, September 2005.

[40] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. of the 2008 IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, CA, May 2008.

[41] The MITRE Corporation. Common vulnerabilities and exposures (CVE) database. `http://cve.mitre.org/data/downloads/`.

[42] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

[43] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proc. of the 11th ACM Symposium on Access Control Models and Technologies*, pages 160–169, Lake Tahoe, CA, June 2006.

[44] W. Venema. Taint support for PHP. `http://wiki.php.net/rfc/taint`.

[45] J. Viega, J. T. Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, February 2001.

[46] T. Waldmann. Check the ACL of the included page when using the rst parser's include directive. `http://hg.moinmo.in/moin/1.6/rev/35ff7a9b1546`. CVE-2008-6548.

[47] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proc. of the 2007 PLDI*, pages 32–41, San Diego, CA, June 2007.

[48] Web Application Security Consortium. 2007 web application security statistics. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[49] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of the 15th USENIX Security Symposium*, pages 179–192, Vancouver, BC, Canada, July 2006.

[50] A. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *Proc. of the 4th NSDI*, pages 159–172, Cambridge, MA, April 2007.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, November 2006.

[52] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. of the 5th NSDI*, pages 293–308, San Francisco, CA, April 2008.