Secure Web Applications via Automatic Partitioning

Stephen Chong Jed Liu Andrew C. Myers Xin Qi
K. Vikram Lantian Zheng Xin Zheng
Department of Computer Science
Cornell University
{schong,liujed,andru,qixin,kvikram,zlt,xinz}@cs.cornell.edu

Abstract

Swift is a new, principled approach to building web applications that are *secure by construction*. In modern web applications, some application functionality is usually implemented as client-side code written in JavaScript. Moving code and data to the client can create security vulnerabilities, but currently there are no good methods for deciding when it is secure to do so.

Swift automatically partitions application code while providing assurance that the resulting placement is secure and efficient. Application code is written as Java-like code annotated with information flow policies that specify the confidentiality and integrity of web application information. The compiler uses these policies to automatically partition the program into JavaScript code running in the browser, and Java code running on the server. To improve interactive performance, code and data are placed on the client side. However, security-critical code and data are always placed on the server. Code and data can also be replicated across the client and server, to obtain both security and performance. A max-flow algorithm is used to place code and data in a way that minimizes client–server communication.

Categories and Subject Descriptors: D.4.6 [Security and Protection]: Information flow controls, D.3.3 [Language Constructs and Features]: Frameworks, I.2.2 [Automatic Programming]: Program transformation

General Terms: Security, Languages

Keywords: Information flow, security policies, compilers.

1. Introduction

Web applications are client–server applications in which a web browser provides the user interface. They are a critical part of our infrastructure, used for banking and financial management, email, online shopping and auctions, social networking, and much more. The security of information manipulated by these systems is crucial, and yet these systems are not being implemented with adequate security assurance. In fact, web applications are recently reported to comprise 69% of all Internet vulnerabilities [24]. The problem is that with current implementation methods, it is difficult to know whether an application adequately enforces the confidentiality or integrity of the information it manipulates.

Recent trends in web application design have exacerbated the security problem. To provide a rich, responsive user interface, application functionality is pushed into client-side JavaScript [8] code that executes within the web browser. JavaScript code is able to manipulate user interface components and can store information persistently on the client side by encoding it as cookies. These web applications are distributed applications, in which client- and server-side code exchange protocol messages represented as HTTP requests and responses. In addition, most browsers allow JavaScript code to issue its own HTTP requests, a functionality used in the Ajax development approach (Asynchronous JavaScript and XML).

With application code and data split across differently trusted tiers, the developer faces a difficult question: when is it secure to place code and data on the client? All things being equal, the developer would usually prefer to run code and store data on the client, avoiding server load and client–server communication latency. But moving information or computation to the client can easily create security vulnerabilities.

For example, suppose we want to implement a simple web application in which the user has three chances to guess a number between one and ten, and wins if a guess is correct. Even this simple application has subtleties. There is a confidentiality requirement: the user should not learn the true number until after the guesses are complete. There are integrity requirements, too: the match between the guess and the true number should be computed in a trustworthy way, and the guesses taken must also be counted correctly.

The guessing application could be implemented almost entirely as client-side JavaScript code, which would make the user interface very responsive and would offload the most work from the server. But it would be insecure: a client with a modified browser could peek at the true number, take extra guesses, or simply lie about whether a guess was correct. On the other hand, suppose guesses that are not valid numbers between one and ten do not count against the user. Then it is secure and indeed preferable to perform the bounds check on the client side. Currently, web application developers lack principled ways to make decisions about where code and data can be securely placed.

We introduce the Swift system, a way to write web applications that are *secure by construction*. Applications are written in a higher-level programming language in which information security requirements are explicitly exposed as declarative annotations. The compiler uses these security annotations to decide where code and data in the system can be placed securely. Code and data are partitioned at fine granularity, at the level of individual expressions and object fields. Developing programs in this way ensures that the resulting distributed application protects the confidentiality and

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

integrity of information. The general enforcement of information integrity also guards against common vulnerabilities such as SQL injection and cross-site scripting.

Swift applications are not only more secure, they are also easier to write: control and data do not need to be explicitly transferred between client and server through the awkward extralinguistic mechanism of HTTP requests. Automatic placement has another benefit. In current practice, the programmer has no help designing the protocol or interfaces by which client and server code communicate. With Swift, the compiler automatically synthesizes secure, efficient interfaces for communication.

Of course, others have noticed that web applications are hard to make secure and awkward to write. Prior research has addressed security and expressiveness separately. One line of work has tried to make web applications more secure, through analysis [11, 27, 12] or monitoring [10, 17, 28] of server-side application code. However, this work does not help application developers decide when code and data can be placed on the client. Conversely, the awkwardness of programming web applications has motivated a second line of work toward a single, uniform language for writing distributed web applications [9, 4, 21, 30, 29]. However, this work largely ignores security; while the programmer controls code placement, nothing ensures the placement is secure.

Swift thus differs from prior work by addressing both problems at once. Swift automatically partitions web application code while also providing assurance that the resulting placement enforces security requirements. Addressing both problems at the same time makes it possible to do a better job at each of them.

Prior work on program partitioning in the Jif/split language [32, 33] has explored using security policies to drive code and data partitioning onto a general distributed system. Applying this approach to the particularly important domain of web applications offers both new challenges and new opportunities. In the Swift trust model, the client is less trusted than the server. Code is placed onto the client in order to optimize interactive performance, which has not been previously explored. Swift has a more sophisticated partitioning algorithm that exploits new replication strategies. And because Swift supports a richer programming language with better support for dynamic security enforcement, it can control information flow even as a rich, dynamic graphical user interface is used to interact with security-critical information.

The remainder of the paper is structured as follows. Section 2 gives an overview of the Swift architecture. Section 3 describes the programming model, based on an extension of the Jif programming language [16] with support for browser-based user interfaces. Sections 4 and 5 explain how high-level Swift code is compiled into an intermediate language, WebIL, and then partitioned into Java and JavaScript code. Section 6 presents results and experience using Swift, Section 7 discusses related work, and Section 8 concludes.

2. Architecture

Figure 1 depicts the architecture of Swift. The system starts with annotated Java source code at the top of the diagram. Proceeding from top to bottom, a series of program transformations converts the code into a partitioned form shown at the bottom, with Java code running on the web server and JavaScript code running on the client web browser.

Jif source code.

The source language of the program is an extended version of the Jif 3.0 programming language [14, 16]. Jif extends the Java programming language with language-based mechanisms for information flow control and access control. Information security policies

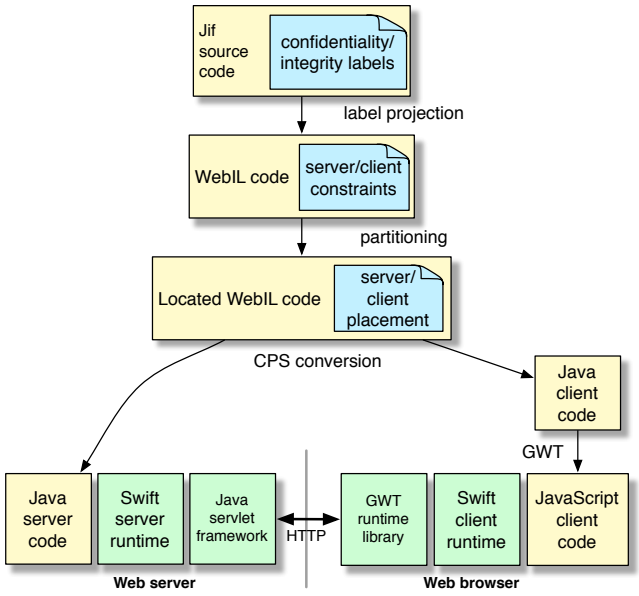

Figure 1: The Swift architecture

can be expressed directly within Jif programs, as *labels* on program variables. By statically checking a program, the Jif compiler ensures that these labels are consistent with flows of information in the program.

The original model of Jif security is that if a program passes compile-time static checking, and the program runs on a trustworthy platform, then the program will enforce the information security policies expressed as labels. For Swift, we assume that the web server can be trusted, but the client machine and browser may be buggy or malicious. Therefore, Swift must transform program code so that the application runs securely, even though it runs partly on the untrusted client.

WebIL intermediate code.

The first phase of program transformation converts Jif programs into code in an intermediate language we call WebIL. As in Jif, WebIL types can include annotations; however, the space of allowed annotations is much simpler, describing constraints on the possible locations of application code and data. For example, the annotation `S` means that the annotated code or data must be placed on the web server. The annotation `C?S` means that it must be placed on the server, and *may* optionally be replicated on the client as well. WebIL is useful for web application programming in its own right, although it does not provide security assurance.

WebIL optimization.

The initial WebIL annotations are merely constraints on code and data placement. The second phase of compilation decides the exact placement and replication of code and data between the client and server, in accordance with these constraints. The system attempts to minimize the cost of the placement, in particular by avoiding unnecessary network messages. The minimization of the partitioning cost is expressed as an integer programming (IP) problem, and maximum flow methods are then used to find a good partitioning.

Splitting code.

Once code and data placements have been determined, the compiler transforms the original Java code into two Java programs,

one representing server-side computation and the other, client-side computation. This is a fine-grained transformation. Different statements within the same method may run variously on the server and the client, and similarly with different fields of the same object. What appeared as sequential statements in the program source code may become separate code fragments on the client and server that invoke each other via network messages. Because control transfers become explicit messages, the transformation to two separate Java programs is similar to a conversion to *continuation-passing style* [20, 22].

JavaScript output.

Although our compiler generates Java code to run on the client, this Java code actually represents JavaScript code. The Google Web Toolkit (GWT) [9] is used to compile the Java code down to JavaScript. On the client, this code then uses the GWT run-time library and our own run-time support. On the server, the Java application code links against Swift's server-side run-time library, which in turn sits on top of the standard Java servlet framework.

The final application code generated by the compiler uses an Ajax approach to securely carry out the application described in the original source code. The application runs as JavaScript on the client browser, and issues its own HTTP requests to the web server, which responds with XML data.

From the browser's perspective, the application runs as a single web page, with most user actions (e.g., clicking on buttons) handled by JavaScript code. This approach seems to be the current trend in web application design, replacing the older model in which a web application is associated with many different URLs. One result of the change is that the browser "back" and "forward" buttons no longer have the originally intended effect on the web application, though this can be largely hidden, as is done in the GWT.

Partitioning and replication.

Compiling a Swift application puts some code and data onto the client. Code and data that implement the user interface clearly must reside on the client. Other code and data are placed on the client to avoid the latency of communicating with the server. With this approach, the web application can have a rich, highly responsive user interface that waits for server replies only when security demands that the server be involved.

In order to enforce the security requirements in the Jif source code, information flows between the client and the server must be strictly controlled. In particular, confidential information must not be sent to the client, and information received from the client cannot be trusted. The Swift compilation process generates code that satisfies these constraints.

One novel feature of Swift is its ability to selectively replicate computation onto *both* the client and server, improving both responsiveness and security. For example, validation of form inputs should happen on the client so the user does not have to wait for the server to respond when invalid inputs are provided. However, client-side validation should not be trusted, so input validation must also be done on the server. In current practice, developers write separate validation code for the client and server, using different languages. This duplicates effort and makes it less likely that validation is done correctly and consistently. With Swift, the compiler can automatically replicate the same validation code onto both the server and the client. This replication is not a special-purpose mechanism; it is simply a result of applying a general-purpose algorithm for optimizing code placement.

In the next few sections, we more closely examine the various compilation phases illustrated in Figure 1.

3. Writing Swift applications

3.1 Labels and principals

Programming with Swift starts with a program written in the Jif programming language [14, 16], with a few extensions. A little background on Jif will therefore be helpful.

Information security requirements are expressed in Jif programs using labels from the decentralized label model (DLM) [15]. A label is a set of policies. For example, the confidentiality policy `alice→bob` says that principal `alice` owns the labeled information but permits principal `bob` to read it. Similarly, the integrity policy `alice←bob` means that `alice` permits `bob` to affect the labeled information. Because labels express security requirements explicitly in terms of principals, and keep track of *whose* security is being enforced, they are useful for systems where principals need to cooperate despite mutual distrust. Web applications are examples of such systems.

Labels can be attached to types, making Jif a *security-typed* language [26]. For example, the following declaration uses a label containing two policies separated by a semicolon:

```
int {alice→bob,alice; bob←alice} y;
```

It means that the information in `y` is considered sensitive by `alice`, who considers that it can be released securely only to `bob` and `alice`; and further that it is considered trustworthy by `bob`, who believes that only `alice` should be allowed to affect it.

The Jif compiler uses labels to statically check that information flows within Jif code are secure. For example, consider the following code fragment (using the same variable `y`):

```
int {bob→bob} x;
int {alice→bob; bob→alice} z;
if (x == 0)
   z = y;
```

This code causes an explicit information flow from `y` to `z`. For the code to be secure, the label on `z` must restrict the use of data in `z` as least as much as the label on `y` restricts the use of `y`. This is true if (1) for every confidentiality policy on `y`, there is one at least as restrictive on `z` (which is the case because `alice→bob` is at least as restrictive as `alice→bob,alice`) and (2) for every integrity policy on `z`, there is one at least as restrictive on `y` (which is the case because `z` has no integrity policy; the integrity of `y` (`bob←alice`) is extra.

More subtly, the code also causes an *implicit* information flow from `x` to `z`, because inspecting `z` after the code runs may impart information about `x`, even if the assignment from `y` never happens. Implicit flows are important. A failure to control this implicit flow would mean that an attacker could violate confidentiality by improperly learning about the value of `z`, or could violate integrity by changing `z` and improperly affecting the control flow of the program. Compared to purely dynamic taint tracking mechanisms, a static analysis of information flow can detect implicit flows with greater precision [6]. This precision is necessary for the applications described later in this paper.

Applying the above rule, the implicit flow from `x` to `z` is secure if `alice→bob` is at least as restrictive as `bob→bob`. In general, this condition does not hold, because the second policy is owned by `bob`, who would not trust any enforcement of the second policy on behalf of its owner (`alice`). However, the implicit flow *would* be secure if `alice` *acts for* `bob`, meaning that `bob` trusts `alice` completely, and as a result, `alice→bob` is at least as restrictive as `bob→bob`. Acts-for relationships increase the expressive power of labels and allow static information flow checking to work even though trust relationships change over time.

Two principals are already built into Swift programs. The principal * (also server) represents the maximally trusted principal in the system. The principal client represents the other end of the current session—in ordinary, non-malicious use, a web browser under the control of a user. When reasoning about security, we can only assume that the client is the other end of a network connection, possibly controlled by a malicious attacker. Because the server is trusted, the principal * acts for client. The client may see information whose confidentiality is no greater than *→client, and can produce information with integrity no greater than *←client.

A Swift program may use and even create additional principals, for example to represent different users of a web application. For a user to log in as principal bob, server-side application code trusted by bob must establish that the principal named by client acts for bob. Applications can define their own authentication methods for this purpose. Once the relationship exists, the client can act for bob; for example, information labeled alice→bob could be released to that client.

There are actually multiple principals denoted by client, whose identity is determined by which client initiated the current request. To prevent different session principals named as client in the code from being confused with each other, the Swift compiler requires that the types of static variables not reference the principal client, even indirectly. This works because different Swift sessions can only interact or access shared persistent state through static variables.

3.2 A sample application

The key features of the Swift programming model can be seen by studying a simple web application written using Swift. Figure 2 shows key fragments of the Jif source code of the number-guessing web application described in Section 1. Java programmers will recognize this code as similar to that of an ordinary single-machine Java application that uses a UI library such as Swing [23]. For example, it has a user interface dynamically constructed out of widgets such as buttons, text inputs, and text labels. Swift widgets are similar to those in the Google Web Toolkit [9], communicating via events and listeners. The crucial difference is that Swift controls how information flows through them.

The core application logic is found in the makeGuess method (lines 15–39). Aside from various security label annotations, this method is essentially straight-line Java code. To implement the same functionality with technologies such as JSP [1] or GWT requires more code, in a less natural programming style with explicit control transfers between the client and server.

The code contains various labels expressing security requirements. Because this example is very simple, just the principals client and * are used in these labels. For example, on line 3, the variable secret is declared to be completely secret (*→*) and completely trusted (*←*); the variable tries on the next line is not secret (*→client) but is just as trusted. Because Jif checks transitively how information flows within the application, the act of writing just these two label annotations constrains many of the other label annotations in the program. The compiler ensures that all label annotations are consistent with the information flows in the program.

The user submits a guess by clicking the button. A listener attached to the button passes the guess (line 50) to makeGuess. The listener reads the guess from a NumberTextBox widget that only allows numbers to be entered.

The makeGuess method receives a guess num from the client. The variable num is untrusted and not secret, as indicated by its label {*→client} on line 15. The label after the name of the

```
1  public class GuessANumber {
2    final label{*←*} cl = new label{*→client};
3    int{*→*; *←*} secret;
4    int{*→client; *←*} tries;
5    ...
6    private void setupUI{*→client}() {
7      guessbox = new NumberTextBox[cl, cl]("");
8      message = new Text[cl, cl]("");
9      button = new Button[cl, cl]("Guess");
10     ...
11     rootpanel.addChild(cl, cl, guessbox);
12     rootpanel.addChild(cl, cl, button);
13     rootpanel.addChild(cl, cl, message);
14   }
15   void makeGuess{*→client}(Integer{*→client} num)
16     where authority(*), endorse({*←*})
17     throws NullPointerException
18   {
19     int i = 0;
20     if (num != null) i = num.intValue();
21     endorse (i, {*←client} to {*←*})
22     if (i >= 1 && i <= 10) {
23       if (tries > 0 && i == secret) {
24         declassify ({*→*} to {*→client}) {
25           tries = 0;
26           finishApp("You win!");
27         }
28       } else {
29         declassify ({*→*} to {*→client}) {
30           tries--;
31           if (tries > 0) message.setText("Try again");
32           else finishApp("Game over");
33         }
34       }
35     } else {
36       message.setText("Out of range:" + i);
37     }
38   }
39 }
40 class GuessListener
41   implements ClickListener[{*→client}, {*→client}] {
42   ...
43   public void onClick{*→client} (
44         Widget[{*→client}, {*→client}]{*→client} w)
45     :{*→client} {
46     if (guessApp != null) {
47       NumberTextBox[cl, cl]{*→client} guessbox =
48         guessApp.guessbox;
49       if (guessbox != null)
50         guessApp.makeGuess(guessbox.getNumber());
51     }
52   }
53 }
```

Figure 2: Guess-a-Number web application

method, also {*→client}, is the *begin label* of the method. It bounds what might be learned from the fact that the method was invoked, by preventing callers from causing any greater implicit flow. Jif also keeps track of implicit flows out of methods using *end labels*; in the case of makeGuess, no additional annotations are required for this purpose because the end label is the same as the begin label. Jif aims to protect the confidentiality and integrity of program data rather than of program code. However, end labels and begin labels can be used to respectively protect the confidentiality and integrity of code.

The code of makeGuess checks whether the guess is correct, and either informs the user that he has won, or else decrements the remaining allowed guesses and repeats. Because the guess is untrusted, Jif will prevent it from affecting trusted variables such as tries, unless it is explicitly *endorsed* by trusted code. Therefore, lines 21–37 have a *checked endorsement* that succeeds only if num contains an integer between one and ten. If the check succeeds, the number i is treated as a high-integrity value within the "then" clause. If the check fails, the value of i is not endorsed, and

```
1  class Widget[label Out, label In] { ... }
2  class Panel[label Out, label In]
3   extends Widget[Out,In] {
4    void addChild{Out}(label wOut,
5                       label wIn,
6                       Widget[wOut,wIn]{Out} w)
7      where {*wOut} <= Out, {In;w} <= {*wIn};
8  }
9  class ClickableWidget[label Out, label In]
10  extends Widget[Out,In] {
11   void addListener{In}
12     (ClickListener[Out,In]{In} li);
13 }
14 class Button[label Out, label In]
15  extends ClickableWidget[Out,In] {
16   String{Out} getText();
17   void setText{Out}(String{Out} text);
18 }
19 interface ClickListener[label Out, label In] {
20   void onClick{In}(Widget[Out, In]{In} b);
21 }
```

Figure 3: UI framework signatures

the "else" clause is executed. Checked endorsements are a Swift-specific Jif extension that makes the common pattern of validating untrusted inputs both explicit and convenient.

By forcing the programmer to use `endorse`, the potential security vulnerability is made explicit. In this case, the endorsement of `i` is reasonable because it is intrinsically part of the game that the client is allowed to pick any value it wants (as long as it is between one and ten).

Similarly, some information about the secret value `secret` is released when the client is notified whether the guess `i` is equal to `secret`. Therefore, the bodies of both the consequent and the alternative of the `if` test on line 23 must use an explicit `declassify` to indicate that information transmitted by the control flow of the program may be released to the client. Without the `declassify`, client-visible events—showing messages, or updating the variable `tries`—would be rejected by the compiler.

The `declassify` and `endorse` operations are inherently dangerous. Jif controls the use of `declassify` and `endorse` by requiring that they occur in a code marked as trusted by the affected principals; hence the clauses `authority(*)` and `endorse({*←*})` on line 16. The latter, *auto-endorse* annotation means that an invocation of `makeGuess` is treated as trusted even if it comes from the client. Jif also enforces a security property of *robust declassification* [2], in which declassification cannot be performed without sufficient integrity. Untrusted information is not allowed to affect security-critical operations such as declassification, even indirectly.

3.3 Swift user interface framework

Swift programs interact with the user via a user interface framework. This framework abstracts away the details of the underlying HTML and JavaScript, allowing programming in a event-driven style familiar to users of UI frameworks such as Swing. The control of information flow in a rich, interactive, dynamically changing graphical user interface is a novel feature of Swift.

Figure 3 presents part of the signatures of several Swift UI framework classes. The class `Widget` is the ancestor of all user interface widgets, such as `TextBox` (which allows a user to enter text), `Button` (which represents a clickable button), and `Panel` (which contains other widgets).

All classes in the framework are annotated with security policies that track information flow that may occur within the framework. The framework ensures that the client is permitted to view all information that the user interface displays. Conversely, all information received from the user interface is annotated as having been tainted by the client.

The user interface classes demonstrate an important feature of Jif. Classes may be parameterized with respect to principals or labels, as indicated by the parameters in brackets following the name of each class. The Jif parameterization mechanism is superficially similar to the parameterized type mechanism in recent versions of Java, but differs in that parameter values are usable at run time.

All widget classes are parameterized on two security labels, `Out` and `In`. The parameter `Out` is an upper bound on the security labels of information that is contained in the widget, or its children. Thus, given labels ℓ and ℓ', the text displayed on a `Button[`ℓ`,`ℓ'`]` object must have a security label no more restrictive than ℓ. This restriction is evidenced by the annotations on the `getText` and `setText` methods, on lines 16–17. Similarly, given a `Panel[`ℓ`,`ℓ'`]` object to which we are adding a child `Widget[`ℓ_w`,`ℓ'_w`] w`, the label of the child's contents, ℓ_w, must be no more restrictive than the upper bound of the panel's content, ℓ. This requirement is expressed in the annotation "`where {*wOut} <= Out`" on the `addChild` method (line 7). This annotation means that the method can be called only if it is known at the call site that the label contained *in* the variable `wOut` is no more restrictive than the label `Out`. (Because `wOut` is a program variable, unlike `Out`, the label *in* `wOut` is written `{*wOut}` to distinguish it from the label *of* `wOut`, written `{wOut}`.)

The parameter `In` of a widget is an upper bound on information that may be gained by knowing an event occurred on the widget. Thus, if a `ButtonListener[`ℓ`,`ℓ'`]` is added as a listener to a `Button[`ℓ`,`ℓ'`]` object, ℓ' is an upper bound on information that the listener may learn by having the `onClick` method invoked. This is shown by the occurrences of the label `{In}` in the `addListener` and `onClick` method signatures on lines 12 and 20. For example, the first `{In}` in the `onClick` signature means that the method can be called only if the implicit information flow into the method is bounded above by `In`.

What information do we learn by knowing an event occurs on a widget? We can at least infer that the widget is displayed to the user, and thus that the widget is reachable from the root panel. For example, suppose an application creates button `bt` if the value of a secret boolean `v` is `true`, and button `bf` if the value is `false`; a listener to `bt` can then infer the value of `v` upon invocation of the `onClick` method. Thus, the `In` parameter for `bt` must be at least as restrictive as the security label for the boolean `v`. More generally, if a `Widget[`ℓ_w`,`ℓ'_w`] w` is added to a `Panel[`ℓ`,`ℓ'`] p`, the security label ℓ'_w must be at least as restrictive as the security label of widget `w`. In addition, since an event on `w` can only occur if the panel `p` is itself added to the UI, we also require that ℓ'_w is at least as restrictive as ℓ'. Both of these restrictions are expressed in the annotation "`where {In;w} <= {*wIn}`", on line 7.

4. WebIL

After the Swift compiler has checked information flows in the Jif program, the program is translated to the intermediate language WebIL. WebIL extends Java with placement annotations for both code and data. Placement annotations define constraints on where code and data may be replicated. These constraints may be due to security restrictions derived from the Jif code, or to architectural restrictions (for example, calls to a database must occur on the server, and calls to the UI must occur on the client).

Whereas Jif allows expression and enforcement of rich security policies from the decentralized label model (DLM) [15], the WebIL language is concerned only with the placement of code and data

```
1   auto void makeGuess(Integer num) {
2     C?S?: int i = 0;
3     C?S?: if (num != null)
4     C?S?:   i = num.intValue();
5     C?Sh: boolean b1 = (i >= 1);
6           boolean b2;
7     C?Sh: if (b1) b2 = (i <= 10); else b2 = false;
8     C?Sh: if (b2) {
9       Sh:   boolean c1 = (tries > 0);
10            boolean c2;
11      Sh:   if (c1) c2 = (i == secret);
12      Sh:     else c2 = false;
13      Sh:   if (c2) {
14    C?Sh:     tries = 0;
15    C?S?:     finishApp("You win!");
16              } else {
17    C?Sh:     tries--;
18    C?S?:     if (tries > 0) {
19      C :       message.setText("Try again");
20              } else {
21    C?S?:       finishApp("Game over");
22              }
23            }
24          } else {
25      C :   message.setText("Out of range:"+i);
26          }
27  }
```

Figure 4: Guess-a-Number web application in WebIL

onto two host machines, the server and the client. Thus, when translating to WebIL, the compiler projects annotations from the rich space of DLM security policies down to the much smaller space of *placement constraints*.

Using the placement constraint annotations, the compiler chooses a partitioning of the WebIL code. A partitioning is an assignment of every statement and field to a host machine or machines on which the statement will execute, or the field be replicated. To optimize performance, partitioning uses an efficient algorithm based on a reduction to the maximum flow problem. A novel feature of WebIL is that code or data may be replicated in order to improve the performance of the application. The partitioned code is then translated into two Java programs, one to run on the server, and the other to run on the client.

WebIL can be used as a source language in its own right, allowing programmers to develop web applications in a Java-like programming language with GUI support, while mostly ignoring issues of code and data placement, and client-server coordination. This approach has many benefits over traditional web application programming, but lacks the full security benefits of Swift.

4.1 Placement annotations

Each statement and field declaration in WebIL is preceded immediately by one of nine possible placement annotations, shown in Table 1: `C`, `S`, `Sh`, `C?Sh`, `C?S?`, `CS`, `CS?`, `C?S`, and `CSh`. Each placement annotation defines the possible placements for the field or statement, as shown in the table. There are three possible placements: *client*, *server*, and *both*. The intuition is that `C` and `S` mean the statement or field must be placed on the client and server respectively, whereas `C?` and `S?` mean it is optional. An `h` signifies high integrity. Figure 4 shows the result of translating Guess-a-Number into WebIL, including placement constraints.

The placement of a field declaration indicates onto which host or hosts the data stored in the field is replicated. For example, if a field has the placement *server*, that field is stored only on the server; if it has the placement *both*, it is replicated on both client and server.

Annotation	Possible placements	High integrity
`C`	{*client*}	N
`S`	{*server*}	N
`Sh`	{*server*}	Y
`CS`	{*both*}	N
`CSh`	{*both*}	Y
`CS?`	{*client*, *both*}	N
`C?S`	{*server*, *both*}	N
`C?Sh`	{*server*, *both*}	Y
`C?S?`	{*client*, *server*, *both*}	N

Table 1: WebIL placement constraint annotations

The placement of a statement indicates onto which host or hosts the computation of the statement is replicated. For compound statements such as conditionals and loops, the placement indicates the hosts for evaluating the test expression. On line 11 of Figure 4, the comparison of the guess to the secret number is given the annotation `Sh`, meaning that it must occur only on the server. Intuitively, this is the expected placement: the secret number cannot be sent to the client, so the comparison must occur on the server. On line 3, the annotation `C?S?` indicates that there is no constraint on where to test that `num` is non-null; that test may occur on the client, on the server, or on both.

For a statement that must execute on the server, the annotation may indicate that it is high-integrity. The annotations `Sh`, `C?Sh` and `CSh` denote high-integrity code. When translating to WebIL code, the Swift compiler will mark a statement as high-integrity if its execution may affect data that the client should not be able to influence. Thus, the client's ability to initiate execution of high-integrity statements must be restricted. As discussed in Section 5, run-time mechanisms prevent this.

Lines 5–14 of Figure 4 are annotated as high-integrity because the execution of these statements may alter or influence the values of the high-integrity variables `tries`, `b1`, `b2`, `c1`, and `c2`. Note that the start of the high-integrity statements, line 5, corresponds to the start of the `endorse` statement of the original Jif program of Figure 2; it is due to this endorsement that the temporary local variables `b1`, `b2`, `c1`, and `c2` are regarded as high-integrity, and they therefore need to be protected from malicious clients. Note that the ability of the client to cause execution of these high-integrity statements comes from the `endorse` annotation at line 16 in the source, reflected in the WebIL code by the `auto` annotation on `makeGuess`.

4.2 Translation from Jif to WebIL

When the compiler translates from Jif to WebIL code, it replaces DLM security policies with corresponding placement constraint annotations, and translates Jif-specific language constructs into Java code. Based on the security policies of the Jif code, the compiler chooses annotations that ensure code and data are placed on the client only if the security of the program will not be violated by a malicious client.

In particular, the translation ensures that data may be placed on a client only if the security policies indicate that the data may be read by the principal `client`; data may originate from the client only if the security policies indicate that the data is permitted to be written by the principal `client`. Similar restrictions apply to code: code may execute on the client only if the execution of the code reveals only information that the principal `client` may learn; the result of a computation on the client can be used on the server only if the

security policies indicate that the computation result is permitted to be written by the principal `client`.

The translation to WebIL also translates Jif-specific language features. Uses of the primitive Jif type `label` are translated to uses of a class `jif.lang.Label`. Declassifications and endorsements are removed, as they have no effect on the run-time behavior of the program. However, they do affect the labels of code and expressions, and therefore affect their placement annotations.

WebIL code is annotated at statement granularity. To allow fine-grained control over the placement of code, compound expressions are translated into a sequence of simple expressions whose results are stored in temporary local variables. Thus, subexpressions of the same source code expression may be computed on different hosts.

4.3 Goals and constraints

The compiler decides the partitioning by choosing a placement for every field and statement of the WebIL program. Placements are chosen to satisfy both the placement constraints and also certain consistency requirements. Once placements are chosen, the WebIL program is split into two communicating programs, one running on the client, and the other running on the server. The goal of choosing placements is to optimize overall performance without harming security. Since network latency is typically the most significant component of web application run time, fields and statements are placed in order to minimize latency arising from messages sent between the client and server. For example, it is desirable to give consecutive statements the same placement.

Replicating computation can also reduce the number of messages. Consider lines 5–8 of the Guess-a-Number application in Figure 4, which check that the user's input `i` is between 1 and 10 inclusive. To securely check that the client provides valid input, these statements must execute on the server. If the value entered by the user is not in the valid range, the server sends a message to the client to execute line 25, informing the user of the error. However, if lines 5–8 execute on *both* the client and server, no server–client message is needed, and the user interface is more responsive.

The placements of a field and of a statement that accesses the field must be consistent. In particular, if a statement writes to a field, then the statement and the field must have the same placement; if a statement reads a field, then the statement must be replicated on a subset of the hosts that the field is replicated on. These consistency requirements simplify the treatment of field accesses in the run-time system, ensuring that every replicated copy of a field is updated correctly, and that every read from a field occurs on a host on which the field is present. These requirements do not reduce the expressiveness of WebIL. Fields can be partitioned from their uses because a simple program transformation rewrites every field access as an assignment to or from a temporary local variable.

Figure 5 shows the `GuessANumber.makeGuess` method after partitioning. A placement has been chosen for each statement. The field `tries` has been replicated on both client and server, requiring all assignments to it to occur on both hosts (lines 14 and 17). Also, the compiler has replicated on both client and server the validation code to check that the user's guess is between 1 and 10 (lines 2–8). The validation code must be on the server for security, but placing it on the client allows the user to be informed of errors (on line 25) without waiting for a server response.

4.4 Partitioning algorithm

The compiler chooses placements for statements and fields in two stages. First, it constructs a weighted directed graph that approximates the control flow of the whole program. Each node in the graph is a statement, and weights on the graph edges are static

```
1  auto void makeGuess(Integer num) {
2     CS : int i = 0;
3     CS : if (num != null)
4     CS :   i = num.intValue();
5     CSh: boolean b1 = (i >= 1);
6          boolean b2;
7     CSh: if (b1) b2 = (i <= 10); else b2 = false;
8     CSh: if (b2) {
9      Sh:   boolean c1 = (tries > 0);
10           boolean c2;
11     Sh:   if (c1) c2 = (i == secret);
12     Sh:     else c2 = false;
13     Sh:   if (c2) {
14    CSh:     tries = 0;
15     S :     finishApp("You win!");
16            } else {
17    CSh:     tries--;
18    CS :     if (tries > 0) {
19     C :       message.setText("Try again");
20              } else {
21     S :       finishApp("Game over");
22              }
23            }
24         } else {
25     C :   message.setText("Out of range: "+i);
26         }
27 }
```

Figure 5: Guess-a-Number after partitioning

approximations of the frequency of execution following that edge. Second, the weighted directed graph and the annotations of the statements and field declarations are used to construct an instance of an integer programming problem, which is then reduced to an instance of the maximum flow problem. The solution for the integer programming problem directly yields the placements for fields and statements.

Control-flow graph.

For each method in the program, a control-flow graph (CFG) is constructed, and, assuming that the method is invoked n times, non-negative, real weights are assigned to edges in the method's CFG. Edge weights are multipliers of n, representing how often that edge is taken. To estimate n, each branch of an `if` statement is assumed to be taken the same number of times, and each loop is assumed to execute ten times before it exits. Exceptions, `break` and `continue` statements, and method calls are ignored.

An interprocedural analysis is then performed to construct a call graph of the whole program. For dynamically dispatched methods, the analysis conservatively finds all possible method bodies that may be invoked. Recursive methods are ignored, so the resulting call graph is acyclic. The application's `main` method and each UI event handler is assumed to be called exactly once, and the weights are propagated through the call graph using each method's CFG with edge weights. At method calls, every possible target is assumed to be invoked the same number of times. The result of the construction is a control flow graph of the entire program, with edge weights that approximate how often the edge is followed.

Integer programming problem.

Using the weighted directed graph and placement constraint annotations on field declarations and statements, the placement problem is expressed as an instance of an integer programming (IP) problem. A solution to the problem assigns all variables in the problem a value in $\{0, 1\}$. Each statement u is associated with two

variables, s_u and c_u. The variable s_u is 1 if the statement u is replicated on the server, and c_u is 1 if u is replicated on the client. For each u, the constraint $s_u + c_u \geq 1$ ensures that every statement has to be replicated somewhere. Also, linear constraints are used to ensure consistency in the annotations between statements that access the same field.

For each edge $e = (u, v)$ in the weighted directed graph, two variables x_e and y_e are used. The variable x_e is 1 if a message is sent from the client to the server when program execution transitions from statement u to statement v. This occurs when v executes on the server, but u does not, and therefore there is a constraint $x_e \geq s_v - s_u$. Similarly, y_e is 1 if a message is sent from the server to the client when program execution transitions on edge e; therefore, there is a constraint $y_e \geq c_v - c_u$.

Let w_e be the weight of edge e. The goal is to find an assignment to all variables that satisfies all constraints, and minimizes the cost of the messages sent. This cost is $\sum_e w_e(x_e + y_e)$.

Although integer programming problems are in general NP-complete, this particular problem has the nice property that its linear relaxation (obtained by replacing the constraint $s_v, c_v, x_e, y_e \in \{0, 1\}$ with $s_v, c_v, x_e, y_e \geq 0$) always has an integral optimal solution. Therefore, placement is polynomial-time solvable, because an integral optimal solution to the linear relaxation is an optimal solution to the IP problem, and linear programming problems are polynomial-time solvable.

An efficient algorithm for the placement problem is designed by reducing the integer programming problem to an instance of the maximum flow problem. The key to the algorithm is the construction of the flow graph H on which maximum flow is computed. It is constructed from the weighted directed graph G that approximates the control flow. Using the preflow-push method [5], the algorithm runs in $O(V^3)$, where V is the number of statements. The algorithm also implements the gap heuristic [7], and achieves a satisfactory performance for compiling the test cases in the paper.

The graph H is constructed from the graph G as follows. First, create G', which is a copy of G, with all edges reversed, i.e., for each edge $e = (u, v) \in G$, there is an edge (v', u') with weight w_e in G'. All nodes and edges of G and G' are in H. For each node u in G with corresponding node u' in G', add an edge (u, u') to H with infinite weight. Add two distinguished nodes to H, t_s and t_c, representing the server side and the client side respectively. Finally, add edges with infinite weights for every statement u that has a known placement: if u is only on the server, add an edge (t_s, u); if it is only on the client, add an edge (u', t_c); if it is on both sides, add two edges (t_s, u') and (u, t_c).

According to the max-flow min-cut theorem [5], there is a maximum flow from t_s to t_c on H, and a minimum cut (S, C), where S and C are two disjoint sets that cover all nodes in H, and $t_s \in S$, $t_c \in C$; the cost of the maximum flow equals that of the minimum cut, and it gives the optimal solution $s^*_u, c^*_u, x^*_e, y^*_e$ to the integer programming problem:

$$s^*_u = \begin{cases} 1 & u' \in S \\ 0 & \text{otherwise} \end{cases} \quad c^*_u = \begin{cases} 1 & u \in C \\ 0 & \text{otherwise} \end{cases}$$

$$x^*_{(u,v)} = \max\{0, s^*_v - s^*_u\}$$

$$y^*_{(u,v)} = \max\{0, c^*_v - c^*_u\}$$

The above solution is legal, because it disallows $s^*_u = c^*_u = 0$: if that were the case, it would imply that $u \in S$ and $u' \in C$, which is impossible as (u, u') has an infinite weight.

Of course, the accuracy of this approach is limited by how closely the weighted directed graph approximates actual run-time behavior. More sophisticated static analysis techniques or profiling data could yield more precise weighted directed graphs. However, in practice the current placements appear to be good.

```
1   // auto void makeGuess(Integer num)
2   block1: (CS)
3     int i = 0;
4     if (num != null) i = num.intValue();
5     goto block2;
6   block2: (CSh)
7     boolean b1 = (i >= 1);
8     boolean b2;
9     if (b1) b2 = (i <= 10); else b2 = false;
10    if (b2) goto block3; else goto block10;
11  block3: (Sh)
12    boolean c1 = (tries > 0);
13    boolean c2;
14    if (c1) c2 = (i == secret); else c2 = false;
15    if (c2) goto block4; else goto block6;
16  ...
17  block10: (C)
18    call message.setText("Out of range: "+i);
```

Figure 6: Guess-a-Number execution blocks

5. The Swift runtime

From a partitioning of a WebIL program, the Swift compiler produces two Java programs. One executes on the server, and the other on the client (after translation to JavaScript). Concurrent execution of these two programs simulates execution of the original Jif program while enforcing its security requirements.

Both programs rely on Swift's run-time support, which manages communication and synchronization. The client and the server have separate run-time systems, which are similar but not identical, since the trust model is asymmetric. The client's run-time system trusts all messages from the server, but the server does not trust any messages from the client.

This section describes the Swift run-time support and shows how WebIL code is translated into Java. It also explains how GWT is used to compile client-side code into JavaScript.

5.1 Execution blocks and closures

Methods in WebIL are divided into units called *execution blocks*, which are contiguous segments of code with the same placement annotation. Execution blocks have a single entry point and one or more exit points. Each execution block has a unique identifier. For example, Figure 6 shows the execution blocks of the Guess-a-Number `makeGuess` method, in which blocks `block2` and `block3` have two exit points and the other blocks have just one. In general, an execution block contains more than one basic block; a simple dataflow analysis finds execution blocks of maximal size.

Figure 7 shows the WebIL code for a more complex example, comprising two methods of a web application "Treasure Hunt." This game has a grid of cells, some of which contain bombs and others, treasure. The user explores the grid by digging in cells, exposing their contents. In the figure, different execution blocks in the code are shown separately boxed and numbered.

Execution blocks are executed sequentially, following branches from each block to the next. Suppose a branch is taken from execution block s to execution block t. If t is to run on a host that did not run s, the other host invokes it by sending a message containing the identifier of t. We call this a *control transfer message*, although the original host may also continue executing t. If s is placed on both hosts, no message need be sent.

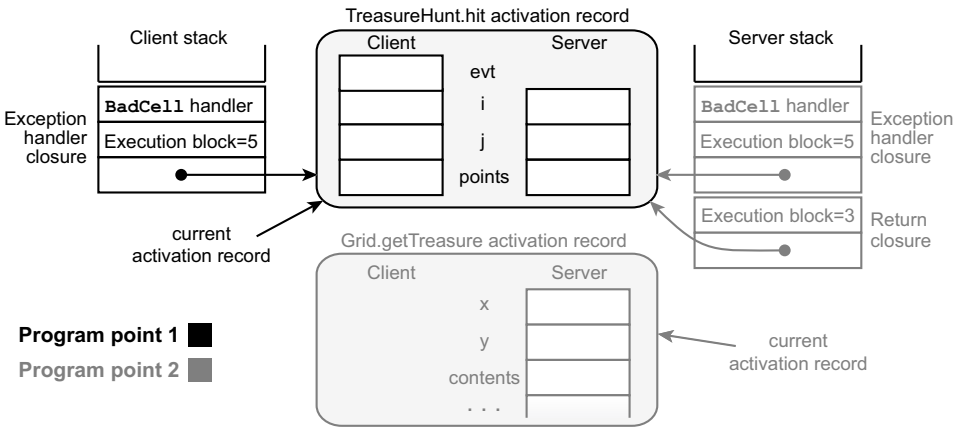

Figure 8: Run-time state at program points 1 (black) and 2 (gray) in Figure 7

Activation records.

An execution block runs in the context of an *activation record*, which stores the state of local variables, including method arguments. For a given activation record, the client and server have distinct views that share the same unique activation record identifier. Each host's view stores only the variables used on that host.

Hosts can forward local variable values to each other, to update the other host's view of the activation record. Activation record updates are piggybacked onto control transfer messages. Importantly for security, there are restrictions on which variables will be sent and received in activation record updates: the server will only forward variables that the client is permitted to read, and only accept updates for variables the client is permitted to write.

Figure 8 shows the state of the run-time system on the client and server during execution of the Treasure Hunt code. The activation record for the method `TreasureHunt.hit` has variables `evt`, `i`, `j`, and `points`. All of these variables are in the client view, but the server view does not contain `evt`, as the server never needs to use that variable. Also, since the variable `points` is high-integrity, the server never accepts updates to it from the client.

Closures.

When one host invokes an execution block on the other, it supplies both the identifier of the next execution block and the identifier for the appropriate activation record. This pair of identifiers forms a *closure*, a self-contained executable unit.

The client and server run-time systems each maintain a stack of closures, which serves two purposes: to correctly simulate the execution of method calls and exceptions, and to enforce the integrity of control flow. The use of the closure stack for control-flow integrity is discussed further in Section 5.2. The closure stacks of the client and server are synchronized by piggybacking stack updates onto control transfer messages.

For method calls and exceptions, two kinds of closures are kept on the stack: *return closures* and *exception handler closures*. A return closure is pushed onto the stack when a method is called, and popped and invoked when it returns. Exception handler closures are pushed on entry to `try...catch` blocks, and popped on exit. They are invoked if a matching exception is thrown within the block.

Closure results.

When a closure s runs, it produces a *result* that identifies the closure t to run next. There are four kinds of results: *simple results*, *exception results*, *method call results*, and *method return results*. A simple result identifies a closure t within the same method as the closure s that returned it, in which case s and t share the same activation record. The run-time system simply invokes t, sending a control transfer message to the other host if needed.

A method call result means that a method is to be invoked next. This result identifies the execution block of the appropriate method body, the receiver object, a new activation record containing the method arguments, and a return closure. The run-time system pushes the return closure onto the stack, makes the new activation record the current one, and transfers control.

Method return results and exception results indicate, respectively, a return from the current method and the throwing of an exception. When a closure returns a method return or exception result, the run-time system walks up the closure stack, popping off closures, until it finds an appropriate return closure or exception handler closure. Once found, the run-time system invokes the appropriate closure, passing it the return value or exception object as appropriate.

Example.

Figure 8 shows the state of the client and server closure stacks at program points 1 and 2 of Figure 7. Figure elements in black are present at program point 1, and elements in gray are added by program point 2.

At program point 1, the client has finished execution of execution block 1. The `try` statement of line 10 has just been entered, and the client has pushed an exception handler closure onto its stack. The handler has three fields: the type of exception it handles (`BadCell`), the execution block identifier, and the activation record identifier of the current call to `TreasureHunt.hit`. If a `BadCell` exception is thrown within the `try...catch` block, the run-time system walks up the stack, finds the handler closure, and invokes it.

At program point 1, the client is just about to return a simple result requesting the invocation of execution block 2. This results in the sending of a control transfer message to the server. Along with the control transfer message, the client sends an activation record update (for the values of variables `i` and `j`) and a stack update (informing the server of the `BadCell` exception handler closure). The server runs execution block 2, which returns a method call result. This causes the return closure for the caller (`TreasureHunt.hit`) to be pushed onto the server's closure stack, a new activation record to be created for `Grid.getTreasure`, and start the execution of the `Grid.getTreasure` method.

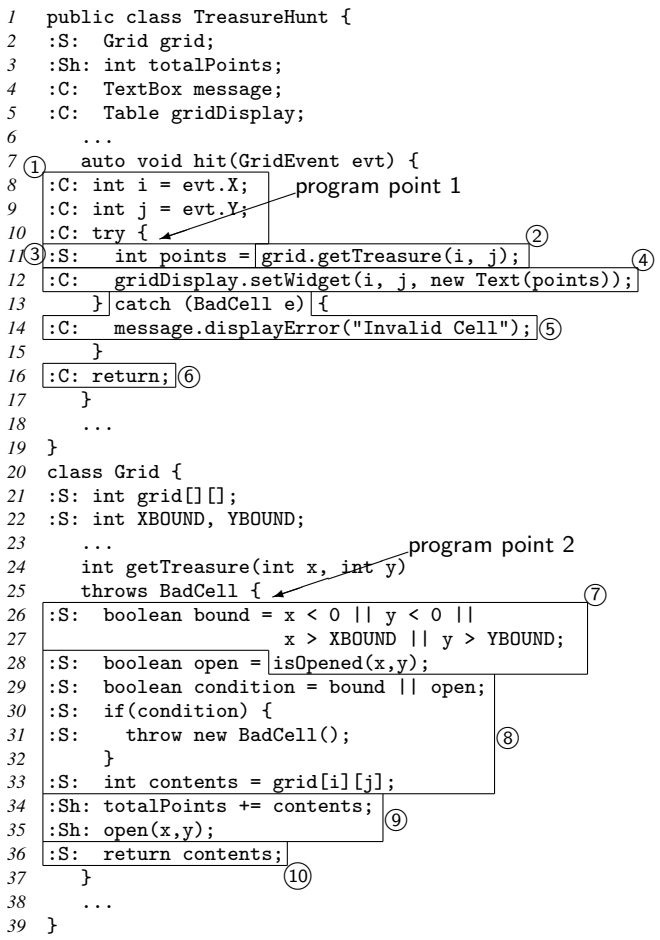

Figure 7: Part of the Treasure Hunt application, in WebIL

At program point 2, the `Grid.getTreasure` method has just begun executing, in execution block 7. The return closure for the caller is on the top of the stack. The return closure contains the closure to execute when the method call returns: execution block 3, and the activation record for the latest invocation of `hit`. The current activation record at program point 2 is the activation record for `Grid.getTreasure`.

5.2 Integrity of control flow

A *high-integrity closure* is one whose execution block has high-integrity side effects, and is therefore annotated `Sh` or `CSh`. A misbehaving client might try to send a control transfer message specifying a high-integrity execution block, and thereby compromise the integrity of variables affected by that execution block. A simple-minded approach would be to prevent the client from invoking high-integrity closures. However, in some situations, the client must legitimately invoke a high-integrity closure on the server. Consider the following WebIL code, after partitioning:

```
1  Sh: this.f = 7;
2  C : this.g = 8;
3  Sh: m(this.f);
```

Lines 1 and 3 are both server-only high-integrity execution blocks, but line 2 must execute on the client. Here, correct control flow of the program requires the client to invoke the high-integrity server closure for line 3.

The server pushes high-integrity closures onto the closure stack, to give the client a controlled way to execute high-integrity closures. A client may invoke a high-integrity closure only if it is at the top of the closure stack. For example, the execution of line 1 pushes a closure for line 3 onto the closure stack, which allows the client execution block at line 2 to invoke line 3, but no other high-integrity closure. Further, a client cannot pop a high-integrity closure without executing it. The server checks that control transfer messages and stack updates received from the client obey these rules. As a result, a misbehaving client cannot control the execution of high-integrity closures, even if it throws arbitrary exceptions and invokes arbitrary closures on the server.

A dataflow analysis is used to statically determine when high-integrity closures should be pushed onto the closure stack. When control flow may pass from a low-integrity execution block u to a high-integrity execution block t, the analysis finds the high-integrity execution blocks s that immediately precedes the low-integrity execution leading to u. The execution of s then pushes the closure for t onto the closure stack. Because the WebIL code was generated from a Jif program with secure information flows, a suitable execution block s exists for each such u and t.

5.3 Classes and objects

A Jif class C is translated into two Java classes: C_s, for use by the server program, and C_c, for the client. For each field f of class C, the placement of f determines whether the field declaration should be placed in C_s or C_c (or both). Each object has a unique identifier. An object o of class C is represented by a pair of objects o_s and o_c that share the same identifier, where o_s of class C_s is on the server, and o_c of class C_c is on the client.

When an object reference is sent from one machine to the other (for example, when forwarding the value of a local variable), it suffices to send the object identifier. If the receiving machine is not aware of the object identifier, a *heap update* is also sent, informing the receiving machine of the run-time class of the object; the receiving machine's run-time system creates an object of the appropriate class, with the specified object identifier.

Label checking on the original Jif source program ensures that heap updates do not violate confidentiality of information: if the server needs to send a heap update to the client for a particular object, then the client is permitted to know about the existence of that object. Conversely, before applying a heap update received from the client, the server checks it for consistency; for example, it checks uniqueness of the object identifier. Object fields are never read before initialization.

5.4 Other security considerations

The fact that WebIL programs are generated from Jif programs with secure information flows is important to ensuring translated code is secure. For example, the client does not learn any secret information by knowing which closures the server requests it to execute. Static checking of the Jif program prevents these implicit flows. Similarly, stack updates, activation record updates and heap updates do not leak information covertly.

Care must also be taken in the run-time system to ensure that no new information channels are introduced by the code translation. For example, the unique identifiers used for activation records and objects form a potential information channel. If identifiers followed a predictable sequence, and confidential information affected the number of objects or activation records created on the server, then information would leak to the client. To prevent this, a cryptographic hash function is used to generate unpredictable identifiers for computation in server-only closures. Thus, sending the identi-

fier of an object or activation record to the client does not reveal any confidential details of the server's execution history.

5.5 GWT and Ajax

We use the Google Web Toolkit [9] (GWT) compiler and framework to translate the client Java programs (and the Swift client runtime system) to JavaScript. GWT provides browser-independent support for Ajax and JavaScript user interfaces. This implementation choice facilitates the development of the Swift run-time system and compiler, but is not fundamental to the design of Swift.

Ajax permits an elegant implementation of the Swift run-time protocol. Communication between client and server occurs mostly invisibly to the user. The server provides a service interface that accepts requests for closure invocations. GWT automatically generates asynchronous proxies that the client can access, and provides marshaling of data sent over the network.

The Ajax model is asymmetric: only the client is able to initiate a dialogue with the server. Any message sent from the server to the client (such as a request to invoke a closure) must be a response to a previous client request. With minor modifications to our runtime system, we can ensure that whenever the server needs to send a message to the client, the client has an outstanding request.

6. Evaluation

The Swift compiler extends the Jif compiler with about 20,000 lines of non-comment non-blank lines of Java code. Both the Swift and Jif compilers are written using the Polyglot compiler framework [18]. The Swift server and client run-time systems together comprise about 2,600 lines of Java code. The UI framework is implemented in 1,400 lines of WebIL code and an additional 560 lines of Java code that adapt the GWT UI library. We also ported the Jif run-time system from Java to WebIL, resulting in about 3,900 lines of WebIL code. The Jif run-time system provides support for run-time representations of labels and principals.

Although Swift shares some ideas and techniques with previous work on Jif/split [33], no compiler or run-time code was reused from Jif/split, because of significant differences between the systems. These differences include a richer source language, use of the intermediate language WebIL, simplified protocols for field access and control transfer, and the optimization of partitioning.

To evaluate our system, we implemented six web applications with varying characteristics. None of these applications is large, but because they test the functionality of Swift in different ways, they suggest that Swift will work for a wide variety of web applications. Because the applications are written in a higher-level language than is usual for web applications, they provide much functionality (and contain many security issues) per line of code. Overall, the performance of these web applications is comparable to what can be obtained by writing applications by hand. Therefore, we do not see any barrier to using this system on much larger web applications.

6.1 Example web applications

Guess-a-Number.

This running example demonstrates how Swift uses replication to avoid round-trip communication between client and server. Figure 5, lines 5–8, show that the compiler automatically replicates the range check onto the client and server, thus saving a network message from the server to the client at line 25. Potential insecurities are also avoided by automatically placing the `tries` field on the server so a malicious client cannot corrupt it, and by placing `secret` on the server where it cannot be leaked or corrupted.

Shop.

This program models an important class of real-world web applications, and is the largest Swift program written to date. It is an online shopping application with a back-end PostgreSQL database. Items may be added to and removed from a shopping cart (automatically updating the total cost), orders can be placed, and users can update their billing information. Users must log in before shopping; new users can register themselves. The database contains both confidential authentication and billing information for each user, and high-integrity inventory information.

Poll.

This application is an online poll that allows users to vote for one of three options and view the current winner. Server-side static fields are used to provide persistence and sharing across multi-user Swift applications. The current count for each choice is kept as a secret on the server, and an explicit declassification makes the result available to users who request to see it.

Secret Keeper.

This simple application allows users to store a secret on the server and retrieve the secret later by logging in. In the source program, the secret of a user has a strong confidentiality policy that only allows that user principal to read it. Once the user logs in, the acts-for relationship established between the client and the user principal permits the secret to be released securely without declassification. This example shows that Swift can handle complex policies with application-defined principals, and that it can automatically generate protocols for password-based authentication and authorization from high-level information security policies.

Treasure Hunt.

This game is described in Section 5.1. It has a relatively rich user interface that is dynamically and incrementally updated as the user discovers what lies beneath cells in the secret grid. Because the grid is secret, it is placed on the server and accessed via Ajax calls as it is explored.

Auction.

This online auction application allows users to list items for sale and bid on items from other users. Once a seller starts an auction, it is visible to other users, and the current bid as well as the bidder's username is shown. The application automatically polls the server to retrieve auction status updates and updates the display. Buyers can enter higher bids until the seller ends the auction. Information about each auction is considered public to users but is maintained with high integrity on the server.

6.2 Code size results

Table 2 shows the code size of the example applications and the generated target code. Generated code size is reported in non-comment tokens rather than in lines, as line counts are not meaningful. However, as a point of comparison, the Jif source programs use 9–11 tokens per line. The "Java target code" columns report the size of the Java output for the server and client. Note that this does not include the Swift run-time systems, nor the UI framework and Jif runtime. (Recall that the UI framework and Jif runtime are both implemented in WebIL.) The "JavaScript All" column reports the size of the code generated by GWT compiling the client Java target code, including the parts of the UI framework and Jif runtime that are partitioned onto the client, and the Swift client runtime; the "JavaScript Framework" column gives the size of code produced by using GWT to compile just the Swift client runtime and the

Example	Jif	Java target code		JavaScript		
		Server	Client	All	Framework	App
Null program	6 lines	0.7k tokens	0.6k tokens	73 kB	70 kB	3 kB
Guess-a-Number	142 lines	12k tokens	25k tokens	267 kB	104 kB	162 kB
Shop	1094 lines	139k tokens	187k tokens	1.21 MB	323 kB	889 kB
Poll	113 lines	8k tokens	17k tokens	242 kB	104 kB	137 kB
Secret Keeper	324 lines	38k tokens	38k tokens	639 kB	332 kB	307 kB
Treasure Hunt	92 lines	11k tokens	11k tokens	211 kB	99 kB	112 kB
Auction	502 lines	46k tokens	77k tokens	503 kB	116 kB	387 kB

Table 2: Code size of example applications

Example	Task	Actual		Optimal	
		Server→Client	Client→Server	Server→Client	Client→Server
Guess-a-Number	guessing a number	1	2	1	1
Shop	adding an item	0	0	0	0
Poll	casting a vote	1	1	0	1
Secret Keeper	viewing the secret	1	1	1	1
Treasure Hunt	exploring a cell	1	2	1	1
Auction	bidding	1	1	1	1

Table 3: Network messages required to perform a core UI task

parts of the UI framework and Jif runtime placed on the client. The difference, in the "JavaScript App" column, indicates how much JavaScript code is specific to the application.

The size of the application JavaScript code is approximately linear in the size of the Jif source. For these applications, about 800 bytes of JavaScript is generated per line of application Jif code. Much of the expansion occurs when Java code is compiled to JavaScript by GWT, so translating WebIL directly to JavaScript might reduce code size.

6.3 Performance results

We studied the performance of the example applications from the user's perspective. We expect network latency to be the primary factor affecting application responsiveness, so we measured the number of network round trips required to carry out the core user interface task in each application. For example, the core user interface task in Guess-a-Number is submitting a guess. We also compared the number of actual round trips to the optimum that could be achieved by writing a secure web application by hand.

Table 3 gives the number of round trips required for each of the applications. To count the number of round trips, we measure the number of messages sent from the server to the client. These messages are the important measure of responsiveness because it is these messages that the client waits for. The table also reports the number of messages sent from the client to the server. Because the client does not block when these messages are sent, the number of messages from client to server is not important for responsiveness.

The total number of round trips in the example applications is always optimal or nearly so. For example, in the Shop application, it is possible to update the shopping cart without any client–server communication. The optimum number of round trips is not achieved for Poll because the structure of Swift applications currently requires that the client hear a response to its vote request. For Guess-a-Number and Treasure Hunt, there are extra client–server messages triggering server-side computations that the client does not wait for, but server–client messages remain optimal.

6.4 Automatic repartitioning

One advantage of Swift is that the compiler can repartition the application when security policies change. We tested this feature with the Guess-a-Number example: if the number to guess is no longer required to be secret, the field that stores the number and the code that manipulates it can be replicated to the client for better responsiveness. Lines 9–13 of Figure 5 all become replicated on both server and client, and the message for the transition from line 13 to 14 is no longer needed. The only source-code change is to replace the label `{*→*; *←*}` with `{*→client; *←*}` on line 3 of Figure 2. Everything else follows automatically.

7. Related work

In recent years there have been a number of attempts to improve web application security. At the same time, there has been increasing interest in unified frameworks for web application development. The goals of these two lines of work are in tension, since moving code to the client affects security. Because it provides a unified programming framework that enforces end-to-end information security policies, Swift is at the confluence of these two lines of work.

7.1 Information flow in web applications

Several previous systems have used information flow control to enforce web application security. This prior work is mostly concerned with tracking information integrity, rather than confidentiality, with the goal of preventing the client from subverting the application by providing bad information (e.g., that might be used in an SQL query). Some of these systems use static program analysis (of information flow and other program properties) [11, 27, 12], and some use dynamic taint tracking [10, 17, 28], which usually has the weakness that the untrusted client can influence control flow. Concurrent work uses a combination of static and dynamic information flow tracking and enforces both confidentiality and integrity policies [3]. Unlike Swift, none of this prior work addresses client-side computation or helps decide which information and computation can be securely placed on the client. Most of the prior work (ex-

cept [3]) only controls information flows arising from a single client request, and not information flow arising across multiple client actions or across sessions.

Instrumenting JavaScript with dynamic security checks [31] has been proposed to protect sensitive client information from cross-site scripting attacks and similar vulnerabilities. In these attacks, a malicious website attempts to retrieve information from another browser window or session to which it should not have access. The usual avenue of attack is via JavaScript's ability to interpret and execute user-provided input as unchecked code, using the `eval` operation. Because Swift does not expose these "higher-order scripting" capabilities of JavaScript, it is not vulnerable to these attacks.

7.2 Uniform web application development

Several recently proposed languages provide a unified programming model for implementing applications that span the multiple tiers found in web applications. However, none of these languages helps the user automatically satisfy security requirements, nor do they support replication for improved interactive performance.

Links [4] and Hop [21] are functional languages for writing web applications. Both allow code to be marked as client-side code, causing it to be translated to JavaScript. Links does this at the coarse granularity of individual functions, whereas Hop allows individual expressions to be partitioned. Links supports partitioning program code into SQL database queries, whereas Hop and Swift do not. Swift does not have language support for database manipulation, though a back-end database can be made accessible by wrapping it with a Jif signature. To keep server resource consumption low, Links stores all state on the client, which may create security vulnerabilities. Neither Links nor Hop helps the programmer decide how to partition code securely.

Hilda [30, 29] is a high-level declarative language for developing data-driven web applications. The most recent version [29] also supports automatic partitioning with performance optimization based on linear programming. Hilda does not support or enforce security policies, or replicate code or data. Hilda's programming model is based on SQL and is only suitable for data-driven applications, as opposed to Swift's more general Java-based programming model. Swift partitions programs on a much finer granularity than on Hilda's "Application Units", which are roughly comparable to classes; fine-grained partitioning is critical to resolve the tension between security and performance. The performance optimization problem in Hilda is NP-complete, and is solved with a bicriteria approximation algorithm, while Swift has a problem that is solvable in polynomial time, and an efficient algorithm is presented.

A number of popular web application development environments make web application development easier by allowing a higher-level language to be embedded into HTML code. For example, JSP [1] embeds Java code, and PHP [19] and Ruby on Rails [25] embed their respective languages. None of these systems help to manage code placement, or help to decide when client-server communication is secure, or provide fully interactive user interfaces (unless JavaScript code is used directly). Programming is still awkward, and reasoning about security is challenging.

The Google Web Toolkit [9] makes construction of client-side code easier by compiling Java to JavaScript, and provides a clean interface for Ajax requests. However, GWT neither unifies programming across the client–server boundary, nor addresses security.

7.3 Security by construction

An important aspect of Swift is that it provides security by construction: the programmer specifies security requirements, and the system transforms the program to ensure that these requirements are met. Prior work has explored this idea in other contexts.

The Jif/split system [32, 33] also uses Jif as a source language and transforms programs by placing code and data onto sets of hosts in accordance with the labels in the source code. Jif/split addresses the general problem of distributed computation in a system incorporating mutual distrust and arbitrary host trust relationships. Swift differs in exploring the challenges and opportunities of web applications. Web applications have a specialized trust model, and therefore specialized construction techniques are used to exploit this trust relationship. In particular, replication is used by Jif/split to boost integrity, whereas Swift uses replication to improve performance and responsiveness. In addition, Swift uses a more sophisticated algorithm to determine the placement and replication of code and data to the available hosts. Swift applications support dynamic user interfaces (represented as complex, compositional data structures) and control the information flows that result. No Jif/split applications contain data structures or control flow of comparable complexity. Jif's label parameterization is needed to reason about information flow in complex data structures, as in Figure 3, but Jif/split lacks the necessary support for label parameters.

Program transformation has also been applied to implementing secure function evaluation in a distributed system, in Fairplay [13]. Its compiler translates a two-party secure function specified in a high-level language into a Boolean circuit. Fairplay provides strong, precise security guarantees for simple computations, but does not scale to general programs. However, its techniques might be applicable within a larger framework such as Swift.

8. Conclusions

We have shown that it is possible to build web applications that enforce security by construction, resulting in greater security assurance. Further, Swift automatically takes care of some awkward tasks: partitioning application functionality across the client–server boundary, and designing protocols for exchanging information.

Writing Swift code does require writing security label annotations. These annotations are mostly found on method declarations, where they augment the information specified in existing type annotations. In our experience, the annotation burden is clearly less than the current burden of managing client–server communication explicitly, even ignoring the effort that should be expended on manually reasoning about security. More sophisticated type inference algorithms might further lessen the annotation burden, but we leave this to future work.

Swift satisfies three important goals: enforcement of information security; a dynamic, responsive user interface; and a uniform, general-purpose programming model. No prior system delivers these capabilities. Because web applications are being used for so many important purposes by so many users, better methods are needed for building them securely. Swift appears to be a promising solution to this important problem.

Acknowledgments

We would like to thank David P. Williamson for the helpful discussion on the optimization problem and its algorithm. Thanks also to the SOSP reviewers and our shepherd David Mazières, for very useful comments and suggestions.

This work was supported in part by the National Science Foundation under grants 0430161 and 0627649, and in part by AF-TRUST (Air Force Team for Research in Ubiquitous Secure Technology for GIG/NCES), which receives support from the DAF Air Force Office of Scientific Research (FA9550-06-1-0244) and the NSF

(0424422). Stephen Chong was awarded an SOSP student travel scholarship, supported by the NSF.

9. References

[1] Hans Bergsten. *JavaServer Pages*. O'Reilly & Associates, Inc., 3rd edition, 2003.

[2] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proc. 19th IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.

[3] Stephen Chong and K. Vikram. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symposium*, August 2007. To appear.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proc. 5th International Symposium on Formal Methods for Components and Objects*, November 2006.

[5] Thomas A. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[6] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[7] U. Derigs and W. Meier. Implementing Goldberg's max-flow algorithm—a computational investigation. *Methods and Models of Operations Research (ZOR)*, 33:383–403, 1989.

[8] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, 2002.

[9] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

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

[11] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proc. 13th International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.

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

[13] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proc. 13th Usenix Security Symposium*, pages 287–302, San Diego, CA, August 2004.

[14] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[15] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

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

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

[18] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[19] PHP: hypertext processor. `http://www.php.net`.

[20] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

[21] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proc. 1st Dynamic Languages Symposium*, October 2006.

[22] Guy L. Steele, Jr. RABBIT: A compiler for Scheme. Technical Report AITR-474, MIT AI Laboratory, Cambridge, MA, May 1978.

[23] Java Swing (Java Foundation Classes) . `http://java.sun.com/javase/technologies/desktop`.

[24] Symantec Internet security threat report, volume X. Symantec Corporation, September 2006.

[25] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, 2nd edition, 2004. ISBN 0-974-51405-5.

[26] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. 7th International Joint Conference on the Theory and Practice of Software Development*, pages 607–621, 1997.

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

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

[29] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applictions with automatic client-server partitioning. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007.

[30] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven web applications. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 32, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proc. 34th ACM Symp. on Principles of Programming Languages (POPL)*, pages 237–249, January 2007.

[32] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[33] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003.