Secure web applications via automatic partitioning

Stephen Chong Jed Liu Andrew C. Myers
Xin Qi K. Vikram Lantian Zheng Xin Zheng

{schong,liujed,andru,qixin,kvikram,zlt,xinz}@cs.cornell.edu

Department of Computer Science
Cornell University

Abstract

Web applications are now critical infrastructure. To improve the user interface, some application functionality is typically implemented as client-side JavaScript code. Currently there are no good methods for deciding when it is secure to move code and data to the client side. Swift is a new, principled approach to building web applications that are secure by construction. Application code is written as Java-like code annotated with information flow policies. This code is automatically partitioned between JavaScript code running in the browser, and Java code running on the server. Code and data are placed on the client side where possible. Security-critical code is placed on the server and user interface code is placed on the client. Code placement is constrained by high-level, declarative information flow policies that strongly enforce the confidentiality and integrity of server-side information.

Web applications are hard to build because code and data needs to be partitioned to make them responsive. They are also hard to build because code and data need to be partitioned for security. Because of the connection (and tension) between the two problems, Swift addresses both at once, automatically partitioning application code while also providing assurance that the resulting placement is secure and efficient.

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 [23]. 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 [7] 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 to fetch information, a functionality exploited by the Ajax (Asynchronous JavaScript and XML) development approach.

With application code and data split across differently trusted tiers, the developer is faced with 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 could 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. This approach would have the most responsive user interface and would offload the most work from the server. But it would be insecure: a client with a modified browser can 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 preferable) to perform the bounds

This work was supported in part by NSF 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 following organizations: the National Science Foundation (0424422), Cisco, British Telecom, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia and United Technologies.

check on the client side. Currently, web application developers lack principled ways to make these kinds of 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 style ensures that the resulting distributed application protects the confidentiality and 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 the client server through specialized extralinguistic mechanisms, such as 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 awkward to write and hard to make secure. Prior research has separately addressed the issues of expressiveness and security. One thread of work has studied the problem of making web applications more secure, through analysis [11, 26, 12] or monitoring [10, 17, 27] of server-side application code. However, this work does not help the application programmer decide when application code and data can be placed on the client. Conversely, the awkwardness of programming web applications has already motivated various attempts to provide a single, uniform language for writing distributed web applications [9, 4, 20, 29, 28]. However, this thread of work largely ignores security issues: the programmer has some control over code placement, but nothing checks that the placement is secure.

Swift thus differs from recent work on web applications, as it addresses both problems at once: it automatically partitions application code while also providing assurance that the resulting placement enforces security requirements. Addressing both problems simultaneously makes it possible to do a better job at each of them.

Prior work on program partitioning [31, 32] has explored using security policies to drive code and data partitioning onto a general distributed system. However, applying this general idea to the specific and crucially 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. And Swift can control information flow even

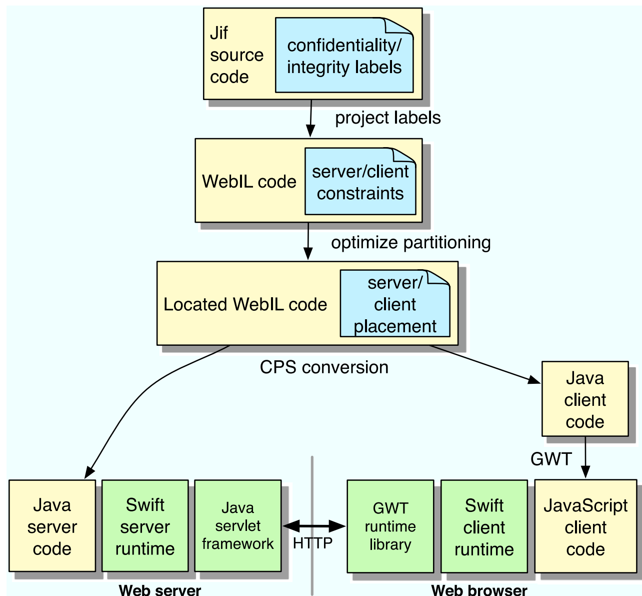

Figure 1: The Swift architecture

when a rich, dynamic user interface interacts 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 for security and performance. Section 6 presents results and experience using Swift to construct web applications, Section 7 discusses related work, and Section 8 concludes.

2 Architecture

Figure 1 depicts the architecture of the Swift approach. 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. Jif is a *security-typed* language [25], in which the types of declared variables are annotated with decentralized security labels [15]. For example, the confidentiality label `{alice→bob}` says that principal `alice` owns the labeled information but permits principal `bob` to read it. Similarly, the integrity label `{alice←bob}` means that `alice` permits `bob` to affect the labeled information. Because labels express security requirements explicitly in terms

of principals, they are useful for systems where principals need to cooperate yet are mutually distrusting. Web applications are examples of such systems.

The model of security in Jif is that if a program passes compile-time static checking, the program will enforce the information security policies expressed as labels in an end-to-end fashion, assuming that the code is running on a trustworthy underlying platform. For Swift, we assume that the web server is trusted, but the client machine and browser are not. The client may be buggy or malicious. Therefore, the program transformations performed by Swift must ensure that application executes securely even though the client uses some application data and performs some application computations.

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 the annotated code or data must be placed on the server, and may optionally be placed on the client.

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 linear programming (LP) methods are then used to find a good partitioning.

Splitting code. Given the location of code and data, the compiler then 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* [19, 21],

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 runtime library and our own runtime support. On the server, the Java application code links against a Swift server-side runtime 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; this JavaScript 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, but this can be hidden largely transparently, as is done in the GWT.

Partitioning and replication. Compiling a Swift application puts some code and data onto the client. The code and data that implement the user interface clearly must reside on the client. Other code and data are placed on the client in order to make the application more responsive, avoiding the latency of communicating with the server. This approach has the advantage that the web application is highly responsive, providing a rich 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 the Swift compiler is its ability to replicate some computation onto *both* the client and server, obtaining 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 validation also needs to be performed on the server. Where appropriate, the Swift compiler automatically replicates validation code onto both the server and the client. This is not a special-purpose mechanism; it happens because of the integrity policies on the data and the optimization algorithms used. This approach is a significant improvement over current practice, in which web application programmers aiming for security and responsiveness write separate validation code for the client and server, typically in different languages. Replicating validation by hand makes it less likely that validation is done correctly and consistently.

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

3 Writing Swift applications

3.1 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 using a UI library such as Swing [22]. For example, it dynamically constructs the user interface out of widgets such as buttons, text inputs, and text labels. Swift widgets are similar to those in the Google Web Toolkit [9], with the difference that Swift controls how information flows through them.

The core application logic occurs in the `makeGuess` method (lines 17–44). 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. Two principals are mentioned in these labels: a distinguished principal `client` representing the client browser, and a Jif-defined principal `*` representing the maximally trusted principal, which in this case is the web server. For example, on line 2, the variable `secret` is declared to be completely secret (`*→*`) and completely trusted (`*←*`), whereas the variable `tries` declared on the next line is not secret (`*→client`) but is just as trusted. Because Jif checks how information flows within the application, just writing these two label annotations constrains many of the other label annotations in the program. Labels must be consistent with the flows of information within the program. In general, Swift code may also contain labels that mention other principals, which is useful for multiuser applications.

The user submits a guess by clicking the button. A listener attached to the button passes the guess (line 53) to `makeGuess`. The listener reads the guess from a `NumberBox` 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 17. The application checks whether the guess is correct, and either informs the user that he has won if so, or else decrements the number of remaining allowed guesses and repeats.

Because the guess is untrusted, Jif will prevent it from being used in a way that affects trusted variables such as `tries`, unless it is explicitly *endorsed*. Therefore, lines 25–42 has 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 the "else" clause is executed. This checked endorsement construct is an extension to Jif that makes the common

```
1  public class GuessTheNumber {
2    int{*→*; *←*} secret;
3    int{*→client; *←*} tries;
4    ...
5    private void setupUI{*→client}() {
6      final label cl = new label{*→client};
7      guessbox = new NumberBox[cl, cl]("");
8      message = new Text[cl, cl]("");
9      button = new Button[cl, cl]("Guess");
10     input = new HorizontalPanel[cl, cl]();
11     input.addChild(cl, cl, textbox);
12     input.addChild(cl, cl, button);
13     input.addChild(cl, cl, message);
14     ...
15     rootpanel.addChild(input);
16   }
17   void makeGuess{*→client}(Integer{*→client}
18                                            num)
19     : {*→client}
20     where authority(*), endorse({*←*})
21     throws NullPointerException
22   {
23     int i = 0;
24     if (num != null) i = num.intValue();
25     endorse (i, {*←client} to {*←*})
26     if (i >= 1 && i <= 10) {
27       if (tries > 0 && i == secret) {
28         declassify ({*→*} to {*→client}) {
29           tries = 0;
30           finishApp("You win!");
31         }
32       } else {
33         declassify ({*→*} to {*→client}) {
34           tries--;
35           if (tries > 0)
36             message.setText("Try again");
37           else finishApp("Game over");
38         }
39       }
40     } else {
41       message.setText("Out of range:" + i);
42     }
43   }
44 }
45 class GuessListener
46   implements ButtonListener[{*→client},
47                             {*→client}] {
48   ...
49   public void onClick{*→client} (
50     Button[{*→client},
51            {*→client}]{*→client} b)
52           :{*→client} {
53     guessApp.makeGuess(guessbox.getNumber());
54   }
55 }
```

Figure 2: Guess-a-Number web application

pattern of validating untrusted inputs both explicit and more convenient.

Because Jif forces the programmer to use `endorse`, they are forced to recognize the potential security vulnerability. This requirement to protect information integrity is also useful against common vulnerabilities such as SQL injection and cross-site scripting. 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 27 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. Jif constrains 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 20. In general Jif enforces a security property of *robust declassification* [2] in which declassification cannot be performed without sufficient integrity.

3.2 Swift user interface framework

Swift programs interact with the user via a user interface framework. The UI framework provides abstraction from the details of the HTML and JavaScript implementation, allowing Swift programmers to use a straightforward event-driven model when implementing web applications. The ability to use a rich, interactive UI framework while controlling information flows is a novel feature of Swift.

Figure 3 presents part of the signatures of several Swift UI framework classes. All classes in the framework are annotated with security policies that track information flow that may occur within the framework. For example, the framework should ensure that all information displayed by the UI is information the user is permitted to view, preventing confidential information from being sent to the user. Also, any information received from the user interface should be annotated as having been tainted by the user.

The class `Widget` is the ancestor of all UI widgets, such as `TextInput` (which allows a user to enter text), `Button` (which represents a clickable button), and `Panel` (which composes widgets). All widget classes are parameterized on two security labels, `L` and `E`. The parameter `L` is an upper bound on the security labels of information that is contained in the widget, or its children. Thus, for labels ℓ and ℓ', the text displayed on a `Button`$[\ell,\ell']$ 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 11–12. Similarly, given a `Panel`$[\ell,\ell']$ object to which we are adding a child `Widget`$[\ell_w,\ell'_w]$ `w`,

```
1  class Widget[label L, label E] { ... }
2  class Panel[label L, label E]
3   extends Widget[L,E] {
4    void addChild{L}(label wL,
5                     label wE,
6                     Widget[wL,wE]{L} w)
7      where {*wL} <= L, {E;w} <= {*wE};
8  }
9  class Button[label L, label E]
10  extends Widget[L,E] {
11   String{L} getText() { ... }
12   void setText{L}(String{L} text) {
13       ...
14   }
15   void addListener{E}
16        (ButtonListener[L,E]{E} li) {
17       ...
18   }
19 }
20 interface ButtonListener[label L,
21                          label E] {
22   void onClick{E}
23        (Button[L, E]{E} b):{E};
24 }
```

Figure 3: UI framework signatures

we require that the security label of the child's contents, ℓ_w, is no more restrictive than the upper bound of the panel's content, ℓ. This requirement is expressed in the annotation "`where {*wL} <= L`" on the `addChild` method (line 7).

The parameter `E` 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`$[\ell,\ell']$ is added as a listener to a `Button`$[\ell,\ell']$ 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 `{E}` on the `addListener` and `onClick` method signatures on lines 15 and 22.

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 adds button `bt` to the UI if the value of a secret boolean `v` is `true`, and button `bf` if the value is `false`; a listener to `bt` can infer the value of `v` upon the invocation of the `onClick` method. Thus, the `E` parameter for `bt` must be at least as restrictive as the security label for the boolean `v`. More generally, if a `Widget`$[\ell_w,\ell'_w]$ `w` is added to a `Panel`$[\ell,\ell']$ `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 {E;w} <= {*wE}`" on line 7.

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

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

Placement 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 annotations

4 WebIL

After the Swift compiler has confirmed that information flow in a Jif program is in accordance with the specified security policies, the program is translated to the intermediate language WebIL.

The language WebIL extends Java with placement annotations for both code and data. Placement annotations provide 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 onto two machines, the server and the client. Thus, when translating to WebIL, the compiler projects from the rich space of DLM security policies down to the much smaller space of placement constraints.

Using the placement annotations, a partitioning of the WebIL code is chosen. A partitioning is an assignment of every statement and field to a machine or machines on which the statement will execute, or the field be replicated. The compiler chooses a partitioning to optimize performance, using a polynomial-time approximation algorithm based on linear programming. 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.

In this section we explain the placement annotations, how the translation from Jif to WebIL uses these annotations, and describe how the compiler chooses a partitioning. Details of the translation of WebIL code to Java are given in Section 5. As a running example, we use the result of translating the `GuessTheNumber.makeGuess` method to WebIL, shown in Figure 4.

With a few modifications (such as providing suitable defaults for missing annotations), 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 it lacks the security benefits of Swift.

4.1 Placement annotations

Each statement and field declaration in WebIL has a placement annotation that appears immediately before it. There are 9 possible placement annotations: `C`, `S`, `CS`, `CS?`, `C?S`, `C?S?`, `Sh`, `C?Sh`, and `CSh`. Each placement annotation defines the possible placements for the field or statement. There are three possible placements: *client*, *server*, and *both*. Table 1 shows the possible placements corresponding to each placement annotation.

The placement of a field declaration indicates on which machine or machines data stored in the field will be replicated. For example, if the field `f` of a class `C` is given the placement *server*, then that field will be stored only on the server; if the field is given the placement *both*, then the field will be replicated on both client and server.

The placement of a statement indicates on which machine or machines the computation of the statement will be replicated. For `if` and loop statements, which are compound statements, the placement indicates the machines for evaluating the test expression. Thus, for example, on line 10 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 placement one should expect: the secret number cannot be sent to the client, so the comparison must occur on the server. On line 4, the annotation C?S? indicates that the test that the input is non-null is unconstrained, and may occur on either the client, the server, or 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.

Examining lines 6–12 of Figure 4, we see they are marked as high integrity, since 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 6, 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 need to be protected from a malicious client.

The Guess-a-Number Jif program also has an endorsement for the entire makeGuess method, indicated by the "endorse({*←*})" annotation on line 20 of Figure 2. This indicates that the start of the method body has integrity *←*. The endorsement is reflected in the WebIL code by a high-integrity empty statement at the start of the method body (Figure 4, line 2). This statement is needed by the runtime system to control the execution of later high-integrity statements.

4.2 Translation from Jif to WebIL

When the compiler translates from Jif to WebIL code, it replaces DLM security policies with corresponding placement 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 will be 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. Declassifications and endorsements are removed, as they have no effect on the runtime behavior of the program. Endorsements and declassifications do however affect the security policies of code and expressions, and thus affect the placement annotations generated. Uses of the primitive Jif type label are translated to uses of a class jif.lang.Label.

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

4.3 Goals and constraints

The compiler determines the partitioning of WebIL code by choosing a placement for every field and statement of the WebIL program. The placements are chosen to satisfy both the placement annotation constraints, and also certain consistency requirements. Once the 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 is to place fields and statements securely in a way that optimizes overall performance. Since network latency is typically the most significant contributor to web application run time, fields and statements are placed in order to minimize the number of messages sent between client and server. For example, it is desirable to give consecutive statements the same placement.

Replicating computation can reduce the number of messages. Consider lines 6–8 of the Guess-a-Number application in Figure 4, which check that the user's input i is between 1 and 10 inclusive. These statements must execute on the server to check that the client provides valid input. If these lines execute only on the server, and the value entered by the user is not in the valid range, then a message must be sent from the server to the client to execute line 23, to inform the user of the error. However, if lines 6–8 execute on both the client and server, then no message from server to client is needed, producing a more responsive user interface.

The placement of a field and 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 machines that the field is replicated on. These consistency requirements simplify the treatment of field accesses in the runtime system, ensuring that every replicated copy of a field is updated correctly, and that every read to a field occurs on a machine on which the field is present. The requirements do not restrict the expressiveness of WebIL, since a simple program transformation can rewrite every field access as an assignment to or from a temporary local variable.

Figure 5 shows the GuessTheNumber.makeGuess method after partitioning. A placement has been chosen for each statement. The field tries has been replicated on both

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

Figure 5: Guess-a-Number after partitioning

client and server, requiring all assignments to it to occur on both machines (lines 12 and 15). 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 the correctness of the code, but placing it on the client allows the user to be informed of an error if needed (line 23) without waiting for a response from the server.

4.4 Partitioning algorithm

The compiler chooses placements for statements and fields in two stages. First, it constructs a weighted directed graph that approximately represents the control flow of the whole program. Each node in the graph is a statement, and weights on the graph edges are static approximations of the relative frequency of program execution transitioning on 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 solved using an approximation algorithm. The solution for the integer programming problem directly yields the placements for fields and statements.

Control flow graph. A weighted directed graph is constructed to approximate program control flow between statements. Weights are assigned to graph edges in a simple tractable manner. First, for each method in the program, we construct a control flow graph (CFG), and, assuming that the method is executed n times, assign (non-negative real) weights to edges in the method's CFG; the edge weights are multipliers of n, representing how often that edge is transitioned. We assume that each branch of an `if` statement is taken the same number of times, and that each loop executes 10 times before the loop exits. We ignore exceptions, `break` and `continue` statements, and method calls.

We then perform an interprocedural analysis, constructing a call graph of the whole program. For dynamically dispatched methods, we conservatively find all possible method bodies that may be invoked. We ignore recursive methods, so the resulting call graph is acyclic. We assume that the application's `main` method and each UI event handler is called exactly once, and use each method's CFG with edge weights to propagate the weights forward through the graph. At method calls, we assume that every possible target is invoked an equal number of times. We thus construct a control flow graph of the entire program with weights on the edges that approximate how often the edge is transitioned.

Integer programming problem. Using the weighted directed graph and the placement constraint annotations of 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 and only if the statement u will be replicated on the server, and c_u is 1 if and only if u will be replicated on the client. Similarly, for each field f there are variables s_f and c_f. For each pair of variables we constrain their values based on their placement annotations. Also, linear constraints are used to ensure consistency in the annotations between fields and the statements that access them.

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 and only if a message is sent from the client to the server when program execution transitions from statement u to statement v. This occurs if v executes on the server, but u does not, and so we require that $x_e \geq s_v - s_u$. Similarly, y_e is 1 if and only if a message is sent from the server to the client when program execution transitions on edge e; we require $y_e \geq c_v - c_u$.

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 $\sum_e w_e(x_e + y_e)$.

We have shown that this problem is NP-complete by a reduction from MAXCUT. Therefore, it is infeasible to solve it optimally in general. However, we have developed a polynomial-time 2-approximation algorithm, based on linear programming (LP) relaxation and the randomized rounding technique [24]. If a solution exists for the IP problem, the approximation algorithm will find a solution whose cost is at most twice that of the optimal solution.

The derandomized version of the algorithm works as follows. First, the IP problem is relaxed to obtain an LP problem by replacing the constraint $s_v, c_v, x_e, y_e \in \{0, 1\}$ with

$s_v, c_v, x_e, y_e \geq 0$. Using an LP solver [8], an optimal solution $s_v^*, c_v^*, x_e^*, y_e^*$ is generated. Define the set of thresholds $T = \{s_v^* \mid 0 < s_v^* < \frac{1}{2}\} \cup \{c_v^* \mid 0 < c_v^* < \frac{1}{2}\}$. For each $t \in T$, generate a candidate solution:

$$\bar{s}_v = \begin{cases} 1 & s_v^* \geq t \\ 0 & \text{otherwise} \end{cases} \quad \bar{c}_v = \begin{cases} 1 & c_v^* \geq t \\ 0 & \text{otherwise} \end{cases}$$

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

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

The threshold t that minimizes the cost $\sum w_e(\bar{x}_e + \bar{y}_e)$ is picked; the corresponding $\bar{s}_v$ and $\bar{c}_v$ are the final solution.

Of course, the accuracy of this approach is limited by how closely the weighted directed graph approximates actual runtime behavior. More sophisticated static analysis techniques or profiling data may yield more precise weighted directed graphs. However, the placements produced by this approach appear to work well in practice.

5 The Swift runtime

From a partitioning of a WebIL program, the Swift compiler produces two Java programs. One executes on the client, and the other on the server. Each statement and field declaration of the WebIL program is represented in one or both of these programs, according to its placement. Concurrent execution of these two Java programs simulates execution of the original Jif program. The two programs rely on Swift's runtime support, which manages communication and synchronization. The client and the server each have a separate runtime system, which are similar but not identical, since the trust model is asymmetric; the client's runtime 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 programs that simulate the execution of the original program while enforcing its security requirements. It also explains how GWT is used to compile client-side application and library code into JavaScript code that executes on the web browser.

5.1 Execution blocks and closures

WebIL code is divided into units called *execution blocks*, which are placed on one or both machines. If simulating the program requires execution of a given execution block s, then all machines on which s is placed will run s. That is, if s is placed on both client and server, then (assuming a well-behaved client) it will never be the case that one machine executes s and the other does not.

Each method of WebIL code is divided into one or more execution blocks. For example, Figure 6 depicts some of the execution blocks of the Guess-a-Number `makeGuess` method. Each execution block has a single entry point, and every statement in an execution block has the same placement. The placement of an execution block is the same as the placement of the statements it contains. In general, an execution block is a small structured program, not just a basic block; a simple dataflow analysis finds execution blocks of maximal size.

```
1  // void makeGuess(Integer num)
2  block0: (CSh) goto block1;
3  block1: (CS)
4    int i = 0;
5    if (num != null) i = num.intValue();
6    goto block2;
7  block2: (CSh)
8    boolean b1 = (i >= 1);
9    boolean b2 = (i <= 10);
10   if (b1&&b2) goto block3; else goto block10;
11 block3: (Sh)
12   boolean c1 = (tries > 0);
13   boolean c2 = (i == secret);
14   if (c1&&c2) goto block4; else goto block6;
15 ...
16 block10: (C)
17   call message.setText("Out of range: "+i);
```

Figure 6: Guess-a-Number execution blocks

The two WebIL runtime systems explicitly represent the stack frames for method invocations. Stack frames contain the values of local variables and arguments. Each stack frame is given a unique identifier, which is shared by the two machines. A *closure* is a pair of an execution block and a stack frame identifier; the Swift runtime systems execute closures. As a closure runs, it may read and update local variables stored in its stack frame. The runtime systems synchronize the values stored in stack frames by forwarding updated local variable values to each other as needed, piggybacked on other messages sent between machines. Forwarding local values does not violate the security policies of the original program. If a local variable stores information that the client should not see, then that local variable will never be needed on the client, and thus the server will never forward its value; if a local variable stores information that the client should not be able to affect, then the server will not accept an update to it from the client.

There is one additional kind of closure: a *return closure*, which represents the point to which control should return after a method call or after exiting a `try` block. Return closures handle dispatching to the appropriate execution block depending on whether the method or `try` block returned normally or with an exception. The runtime systems keep an explicit stack of return closures, called the *closure stack*, so that when an exception is thrown, or a `return` statement executed, the correct next closure can be found. Return closures are pushed onto the stack when a method is called or a `try-catch` block is entered, and popped when execution leaves the method body or the `try` block.

The client and server synchronize the closure stack by sending stack update messages to each other, piggybacked on other messages. Care is taken in the server runtime sys-

tem to ensure that stack update messages from the client cannot violate the security of the original program. This is discussed further in Section 5.4.

5.2 Closure results

When a closure s runs, it produces a *result* controlling what closure t to run next. If s and t are placed on different machines, the runtime system will send a message to the other machine. For example, if s is placed on the server, and t is placed on the client, the server's runtime system sends a message to the client instructing it to begin execution of t. On the other hand, if s is placed on both the client and server, then the server does not need to send a message to the client, since the client's execution of s will ensure that t is invoked.

A closure may have one of four kinds of results: a *standard result*, an *exception result*, a *method call result*, and a *method return result*. A standard result identifies a closure t within the same method as the closure s that returned it (and thus s and t have the same stack frame id). When given a standard result, the runtime system simply invokes the closure, sending a message to the other machine if needed. Standard results are returned when the invocation of a closure does not throw an exception, and does not return from a method call, but simply identifies the next closure to invoke according to the control flow graph of the original method.

A closure returns a method call result when it wishes to invoke a method. The method call result contains a reference to the receiver object, the method identifier, and a *return closure*. The runtime system pushes the return closure onto the closure stack, creates a new stack frame, and dispatches to the execution block that implements the method for the actual class of the receiver object.

When a method return result or an exception result is produced by a closure, the topmost return closure on the stack is invoked. In both cases, the result contains a value (the value returned or thrown), and elements are popped off the closure stack.

5.3 Classes and objects

A Jif class C is translated into two classes: C_s for use by the server Java program, and C_c for the client Java program. 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 object identifier. An object o of class C is represented by a pair of objects o_s and o_c, where o_s of class C_s is on the server, o_c of class C_c is on the client, and o_s and o_c share the same object identifier.

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 runtime class of the object; the receiving machine's runtime system will create 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. The fields of an object will never be read before they are initialized.

5.4 Integrity of control flow

Server-only execution blocks are marked as high-integrity (`Sh`) if statements within the block are marked as high-integrity. The execution of high integrity execution blocks may influence data that the client should not be able to affect, so in general, the WebIL server will not invoke a high integrity closure if requested to by the client.

However, in some situations, the client should be allowed to 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 high-integrity execution blocks, but line 2 must execute on the client. Thus, correct control flow of the program requires the client to invoke the high-integrity closure for line 3.

To allow the client to correctly invoke high-integrity closures, high-integrity closures are pushed onto the closure stack. 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 correctly invoke line 3. A client cannot pop a high-integrity closure without executing it. The server checks that closure invocations and closure stack updates from the client obey these rules. As a result, the client has no way to control the execution of high-integrity closures.

Clearly, only the server should be allowed to push high integrity closures onto the closure stack. 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 precede 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.

5.5 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 the client to execute. Static checking of the Jif program prevents these *implicit flows* [6] (covert storage channels arising from program control structure). Similarly, heap updates and local variable forwarding do not leak information covertly.

Care must also be taken in the runtime system to ensure that no new information channels are introduced in translated code. In particular, the unique identifiers used for stack frames and objects form a potential information channel. If the identifiers of objects and stack frames follow a predictable sequence, and confidential information may affect the number of objects or stack frames created on the server, then the client may be able to infer confidential information based on the object and stack frame identifiers it sees.

To ensure that object and stack frame identifiers do not reveal confidential information, a cryptographic hash function is used to generate unpredictable identifiers for computation in server-only closures. Thus, sending the identifier of an object or stack frame to the client does not reveal any confidential details of the server's execution history.

5.6 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 (Asynchronous JavaScript and XML) and JavaScript user interfaces. This implementation choice facilitates the development of the Swift runtime system and compiler, but is not fundamental to the design of Swift.

Ajax permits an elegant implementation of our runtime protocol. Communication between client and server occurs mostly invisibly to the user. The Swift server runtime system implements 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 has an inherent asymmetry: 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

To evaluate our system, we implemented four web applications with varying characteristics. None of these applications is large, but because they test the functionality of Swift in different ways, it suggests 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 is the running example used in the paper, and is a good demonstration of how Swift uses replication to avoid round-trip communication between client and server. Figure 5, lines 6–8, 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 23. Potential insecurities are also avoided by automatically placing the `tries` field on the server so a malicious client cannot corrupt it, and by placing the `secret` field on the server where it is not leaked or corrupted.

Online Poll. This application is a 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 the user who requests 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 the server to read it. An explicit declassification is used to make the secret readable to the client only after the client provides the correct user name and password. This example shows that Swift can automatically generate password-based authentication and authorization protocols from the high-level information flow policies in source programs.

Treasure Hunt. The fourth application is a treasure hunt game with a relatively rich user interface. The game has a random secret grid in which some cells contain bombs and some cells contain treasure. The user explores the grid by digging up cells, exposing their contents. The user interface is dynamically and incrementally updated as the user discovers what lies beneath each cell in the grid. Because the grid is secret, it is placed on the server and accessed via Ajax calls.

6.2 Code size results

Table 2 shows the code size of the example applications and the generated target code, including both the client-side Java code and the JavaScript code generated by GWT. The length of generated code is reported in non-comment tokens rather than in lines because line counts are not meaningful. However, as a point of comparison, the Jif source programs use 9–11 tokens per line. Note that the generated code length does not include the runtime systems.

Example	Jif	Java target code (server)	Java target code (client)	JavaScript
Null program	4 lines	1.7k tokens	2.1k tokens	67 kB
Guess-a-Number	129 lines	13k tokens	15k tokens	114 kB
Poll	103 lines	11k tokens	11k tokens	105 kB
Secret Keeper	162 lines	17k tokens	17k tokens	116 kB
Treasure Hunt	81 lines	7.9k tokens	7.5k tokens	108 kB

Table 2: Code size of example applications

Example	Actual		Optimal	
	Round-trip	Client→Server	Round-trip	Client→Server
Guess-a-Number	1	2	1	1
Poll	0	2	0	1
Secret Keeper	1	2	1	1
Treasure Hunt	1	1	1	1

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

These results show that the code expansion is reasonable, with about 315 bytes (130 tokens) of JavaScript generated per line of 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 considerably.

6.3 Performance results

We studied the performance of the example applications from the user's perspective. We expect network delays 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. In every case, the total number of round trips in the Swift application is optimal.

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 sent from client to server is not important for responsiveness.

6.4 Automatic repartitioning

One advantage of Swift is that the compiler can repartition the application when security policies change. We experimented 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 9–11 of Figure 5 all become replicated on both server and client, and the message for the transition from line 11 to 12 is no longer needed. The only change in the source code is to replace the label `{*→*; *←*}` with `{*→client; *←*}` (Fig. 2, line 2). Everything else follows automatically.

7 Related work

In recent years there have been a number of attempts to improve the security of web applications. 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 control and taint tracking for 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, 26, 12], and some use dynamic taint tracking [10, 17, 27], 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 (except [3]) only controls information flows arising from a single client request, and not information flow arising across multiple client actions or across sessions.

Yu et al. [30] propose instrumenting JavaScript with dynamic security checks, 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 [20] 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 code to run as SQL queries on a database, 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 [29, 28] is a high-level declarative language for developing data-driven web applications. The most recent version [28] 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. Although both Swift and Hilda use LP-based approximation algorithms, Hilda's bicriteria approximation algorithm solves soft constraints that may be violated by a constant factor; it is therefore not suitable for satisfying the hard constraints needed for security.

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 [18] and Ruby on Rails [5] 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 explicitly used). Programming is still awkward, and reasoning about security is challenging.

The Google Web Toolkit [9] makes construction of rich client-side code easier by compiling Java to JavaScript. GWT also provides a clean interface for Ajax requests. However, GWT does not unify programming across the client–server boundary, nor does it address security.

7.3 Security by construction

An important aspect of Swift is that it provides security by construction: the programmer explicitly 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 [31, 32] 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. Existing Jif/split applications do not appear to handle 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, by the Fairplay [13] system. Its compiler translates a two-party secure function specified in a high-level language into a Boolean circuit. While Fairplay provides strong and precise security guarantees, the computations it can handle are very limited, and it does not scale to general programs.

8 Conclusions

We have shown that it is possible to build web applications that enforce security by construction. Not only is there greater assurance that the resulting application is secure, but web applications are easier to build. The awkward task of partitioning application functionality across the client–server boundary is taken care of automatically and securely.

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. Overall, the annotation burden is clearly less than the current burden of managing client–server communication explicitly, even ignoring the effort that should currently be expended on manually reasoning about security.

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.

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, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. Submitted for publication, 2007.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. `http://groups.inf.ed.ac.uk/links/`, 2006.

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

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

[8] GNU Linear Programming Kit. Available at `http://www.gnu.org/software/glpk/`.

[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 *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[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 *Proceedings of the 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, at 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] PHP: hypertext processor. `http://www.php.net`.

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

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

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

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

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

[24] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

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

[26] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Conference*, July 2006. To appear.

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

[28] 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.

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

[30] 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.

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

[32] 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.