

Secure Compilation of a Multi-Tier Web Language

Ioannis G. Baltopoulos

University of Cambridge Computer Laboratory

Andrew D. Gordon

Microsoft Research

Abstract

Storing state in the client tier (in forms or cookies, for example) improves the efficiency of a web application, but it also renders the secrecy and integrity of stored data vulnerable to untrustworthy clients. We study this general problem in the context of the LINKS multi-tier web-programming language. Like other systems, LINKS stores unencrypted application data, including web continuations, on the client tier; hence, LINKS is open to attacks that expose secrets, and modify control flow and application data. We characterise these attacks as failures of the general principle that security properties of multi-tier applications should follow simply from review of the source code (as opposed to the detailed study of the files compiled for each tier, for example). We propose a secure compilation strategy, which uses authenticated encryption to eliminate these threats, and we implement it as a simple extension to the LINKS system. We model this compilation strategy as a translation from a core fragment of the language to a concurrent λ -calculus equipped with a formal representation of cryptography. To formalize source-level reasoning about LINKS programs, we define a type and effect system for our core language; our implementation can machine-check various integrity properties of the source code. By appeal to a recent system of refinement types for secure implementations, we show that our compilation strategy guarantees all the properties provable by our type and effect system.

1. Introduction

1.1 The Background Trend: Toward Multi-Tier Languages

Nobody could question the success of the web, but they do question the need for so many different web programming languages.

The multiplicity of languages and technologies becomes a hindrance to web application development because one must be well versed in a number of languages to write even trivial web applications (El-Ansary et al. 2004). Their logic usually relies on a range of languages, such as JSP (Sun Microsystems 2006), PHP (The PHP Group 2006), ASP (Microsoft 2006), database queries are written in SQL or XQuery, and then the front-end in languages such as HTML (W3C 1999), CSS (W3C 2007) and JavaScript (ECMA International 1999).

Another problem with so many languages is *impedance mismatch* (Meijer and Schulte 2003): data exchanged between the different tiers of the same application often comes in incompatible

formats and shapes, further complicating the construction of a single application. A third problem is that the abundance of languages and technologies hinders the review of security properties of web applications.

A new class of multi-tier languages enables developers to mix client, server and database source code by shifting the burden of code and data partitioning to the compiler. LINKS is a strongly typed, multi-tier, functional programming language for the web (Cooper et al. 2006). From a single source file, the compiler generates code for the different tiers of the web application: the back-end database, the web server and the client front-end, ensuring that all data is stored either on the client side or in the database. Several other languages adopt similar approaches: HOP (Serrano et al. 2006) is a Scheme-based language for creating interactive applications across the web; Hilda (Yang et al. 2006) is a declarative language for developing data-driven web applications; ML5 (Murphy VII et al. 2008) is a distributed language, with a prototype for web programming that enables code to run on different tiers; the Google Web Toolkit (Google 2006) simplifies the construction of client-side code by compiling Java to JavaScript; LINQ (Microsoft 2005) is a set of extensions to the .NET framework that provides language-level syntax and libraries to express database queries.

These languages provide expressive abstractions for writing web applications; however, it is important that their compilers respect the source level semantics during the code splitting (Neubauer and Thiemann 2005), and, more crucially, that they do so in a secure fashion (Chong et al. 2007). Our objective in this paper is to allow security reasoning about multi-tier programs at the source level and to formalize security properties in the context of LINKS.

1.2 LINKS, a Multi-Tier Functional Language

LINKS is a prototypical example of a multi-tier web language that stores data only on the client or in the database tier. Like other languages, the LINKS system uses continuations for implementing the illusion of cross-tier programming. Continuations are well understood and have been studied extensively in the past 25 years, along with the associated technique of structuring programs in continuation-passing style (CPS) (Plotkin 1975; Appel 1992). It is more recently that their use was suggested in the context of web programming languages (Hughes 2000; Queinnee 2000) and implemented in the PLT Scheme Web Server (Graunke et al. 2001).

While web continuations are an elegant solution to managing the control flow of web applications, several issues need to be considered when implementing them in a language. LINKS represents continuations as closures (expression identifier plus values of free variables) in hidden fields within HTML pages or as URL parameters. This creates a security risk as a malicious client may modify these continuation strings and force unexpected computations on the server.

1.3 Source-Based Security for Multi-Tier Web Applications

The presence of multiple different languages makes it difficult to argue about web application security. However, in a multi-tier language, program analyses can be used to reason statically about the run-time behaviour of entire programs. The benefits of language-based security in comparison to traditional security enforcement mechanisms are argued in the survey paper by Schneider et al. (2000). Language-based policy enforcement solutions can be easily extended or changed to meet new, application-specific demands, and enable fine-grained control of the security policies. Yet, in the context of multi-tier web languages, security depends not only on the language design but also on the correctness of the implementation.

Abadi (1999) uses the concept of full abstraction to relate and bring attention to the connections between an implementation and high-level programs. He postulates that using fully abstract program translations one can ensure that the resulting implementation is incapable of exhibiting behavior disallowed by some security policy at hand. By doing source level analysis, we ensure that only those programs that do not violate the policy are ever given an opportunity to be translated and that the translation is equivalent to the original program. We can therefore characterise possible attacks to web applications as violations of the following principle that captures the essence of our approach.

Principle of Source-Based Reasoning. Security properties of implementations should follow from review of the source code and its source-level semantics.

Our objective is to simplify the way security reviews are conducted by allowing security reasoning about multi-tier programs at the source level and guaranteeing that properties provable at the source level are preserved by the implementation.

1.4 Some Violations of Source-Based Security and a Solution

To devise appropriate counter-measures, we need to specify the threat model and subsequently the kind of protection we provide against possible attacks. The focus of this paper is the threat model where the client (as opposed to some third party) is the attacker. Our main concern is securing code or data that must legitimately be stored in a tier controlled by the attacker. We want to protect our application against rogue clients and guarantee that the implementation corresponds to our intuitions about the source level semantics.

In what follows, we assume an untrusted client that can intercept all messages and decode them. We also consider only LINKS that keep no mutable state in a database or on the server. We assume that the LINKS system's source code may be public (and hence any encoding formats are known to the attacker), but the attacker does not have access to the source code of the program being executed. We also assume that an attacker can guess certain hashes that correspond to legitimate program expressions and that all functions reside on the server. We are using SSL/TLS to protect against a third party and such attacks are not considered.

Given this threat model, the following actual attacks on the LINKS system show failures of the *Source-Based Reasoning* principle.

1. The client may learn secret data that is held in a closure embedded in a webpage; for example, they may learn server data such as a password.
2. The client may break the integrity of server data by modifying a closure embedded in a webpage so as to change future behaviour of the application; for example, the client may change the price of an item in a shopping cart.

3. The client may change the control flow of the program by discovering an unreachable function held in one closure, and then modifying a function value held in another closure.

The problem of untrustworthy clients is not new (for example, Krishnamurthi et al. (2007) discuss it in the context of the PLT Scheme Web Server), and indeed the solution in our setting, without mutable state, is quite simple; still, we believe we are the first to formalise and implement the solution. We propose to apply authenticated encryption (that is, a combination of secrecy and integrity protection) to closures to fix these problems. We need randomized encryption to prevent failures of secrecy such as point (1). We need the encryption to be authenticated (in our case, by including a hash) to prevent failures of integrity such as points (2) and (3). We have coded our proposal as a small modification of the LINKS system.

1.5 TINYLINKS and Formalizing a Secure Implementation

To enable a precise description and a security proof, we introduce TINYLINKS, a core fragment of the LINKS language.

Our development makes use of a concurrent λ -calculus (RCF), and its implementation in the practical typechecker F7 (Bengtson et al. 2008). RCF and F7 rely on refinement types (Xi and Pfenning 1999; Flanagan 2006), to check a range of security properties by typing. F7 checks programs in the ML dialect F# (Syme et al. 2007) against interface files enhanced with refinement types.

We develop a typed, formal model of a server implementing TINYLINKS as a function from HTTP requests to XML responses. Our model is within F7 (that is, within F# but with typing enhanced with refinement types). Given this model, we define the *standard implementation* (as used by the LINKS system) as a translation from TINYLINKS expressions into F7. Moreover, by appeal to existing formal models of cryptography within F7, we describe our *secure implementation* strategy as a simple modification of the standard implementation.

As an example of source-based specifications, we allow event-based assertions as annotations within TINYLINKS. Moreover, we enhance the usual LINKS type system with effects as a technique for proving assertion safety. TINYLINKS is a syntactic subset of LINKS and we have implemented our type system as a type checker for LINKS source code. Hence, we can state and prove expected safety properties, including properties violated by the previously mentioned attacks, within TINYLINKS. The main theorem of the paper, Theorem 2, is that the class of properties provable with our type-and-effect system is preserved by the secure implementation.

Assuming our F7 model of the secure implementation corresponds to our secure modification of the LINKS system, this result amounts to a formal instance of our principle of source-based reasoning: any property proved by running our typechecker as a review of a LINKS source program, actually applies to the implementation of the program as a web application. Hence, for example, we may rule out attack (2) on data integrity and attack (3) on control integrity.

1.6 Contributions

The practical motivation behind our research is to protect the real LINKS system from demonstrable attacks.

We summarise the paper's main contributions as follows.

- We propose a solution based on cryptography and implement our approach as an extension of LINKS. We experimentally evaluate our approach on a collection of examples and modify the real LINKS system to use our secure compilation strategy.
- We define a formal semantics for a subset of LINKS and develop a type-and-effect system that allows source level reasoning about data and control integrity.

- We formalise two type-directed compilation strategies of LINKS programs to F7 expressions: the first one faithfully simulates the LINKS system and fails to preserve source level reasoning, demonstrating the possibility of attacks on the actual implementation; the second compilation strategy is an enhancement of the standard one based on authenticated encryption.
- We prove a correctness result for the secure translation, Theorem 2: if a LINKS program is well typed using the type-and-effect system, then the properties proved are respected by the secure compilation strategy in the F7 level.

2. Introducing the LINKS Multi-Tier Language

A LINKS program is a top-level expression, E_{url} (which typically consists of a series of **var**-bound functions, ending with an XML expression for the top-level page) located at an implicit URL. A client browser initiates the execution of the program by requesting a URL and receives an (XML-encoded) HTML value, the result of evaluating the top-level expression in the program associated with the URL. From the perspective of the user, the computation proceeds by interacting with the page: clicking on links, filling forms and submitting them. From the perspective of the LINKS system, the computation consists of receiving and responding to a series of requests following the HTTP protocol.

2.1 The HTTP Protocol (Review)

The HyperText Transfer Protocol (HTTP) is a stateless, request-response protocol that uses a client-server model. An HTTP client opens a connection and sends a request message to an HTTP server. The requested resource (a static or dynamic page) is identified by a URI. The server returns a response message, usually containing the resource that was requested, and closes the server connection. For creating dynamic pages, a web server can use the Common Gateway Interface (CGI) to run any server-side program to produce responses on its behalf. The protocol defines several possible request messages (methods) that might produce side-effects on the server. In this paper we consider only the GET and POST methods because LINKS is built on top of just these two primitives.

The GET method instructs the browser to retrieve the resource associated with the URI. The resource may be statically or dynamically generated but its production should cause no side-effects. In the case of dynamic resources, the URI supplied by the GET method might contain an additional query string that contains data to be passed to a web application. A question mark in the URI acts as a separator between the host and directory, and the query string. This query string is composed of a series of field-value pairs separated by ampersand characters. The POST method is used to send data to the server, potentially updating server state. The request URI indicates which program is to process the data, with the body of the message containing the encoded data of the form.

2.2 LINKS and its Implementation

The LINKS system is called by the web server as a CGI program, to process an HTTP request and produce an XML response.

Once the source code is uploaded to the server, a user executes the program by entering its URL; this corresponds to a GET request with no query string. The LINKS interpreter receives the incoming request and translates the program into XML. During the translation of client-tier code with embedded LINKS expressions to plain HTML and JavaScript, the suspended expressions embedded inside links and forms need to be transformed, along with their environment bindings, into a continuation string. These continuation strings replace the embedded LINKS expressions in the translated code. To achieve this, the system generates a unique label (a pointer) for each expression that occurs inside the pro-

gram and thereafter maintains an association list between every sub-expression and its label. When constructing a response, the LINKS applies a base64 encoding to the hashed expression label and the environment bindings, representing them as a list of pairs.

An important design decision in implementing web continuations for stateful interactions over the stateless HTTP protocol, is where to store the continuation object (server or client) and for how long. LINKS takes the approach of storing the serialised continuation in a field named $_k$, either as a hidden field inside a form or in the URL itself. During a subsequent GET or POST request, the system checks if such a field exists, and continues the execution from the sub-expression corresponding to the encoded label.

The serialised continuation includes the expression pointer along with the required environment bindings. A malicious user, using simple tools and some knowledge of the encoding used by the LINKS system, can modify these serialised pointers that are passed between the browser and the server. The system upon receiving a subsequent request is unable to determine if a modification has taken place, leaving it exposed to rogue clients.

2.3 TINYLINKS, a Fragment of LINKS

The compilation process followed by LINKS is formalised in an extended fragment of the language which we call TINYLINKS. TINYLINKS is an extension of the simply-typed, call-by-value λ -calculus with XML values for representing web pages.

The syntax of TINYLINKS goes beyond LINKS for two reasons. First, so that we may represent the browsing behaviour of users as code within TINYLINKS, we include **get** and **post** expressions that use HTTP to fetch links and fill in forms. Second, so that we may specify expected trace-based properties within code, we include **event** and **assert** expressions.

We have purposefully left out several LINKS features like database queries, concurrency, formlets and tier splitting from our formalisation. Our simplification is justified since the security anomalies we are studying can be observed even without all these features, which would simply distract and complicate our theory. For similar reasons, our formalisation does not support recursive functions but we can easily generalise it to include them.

The original (LINKS) program goes through a sequence of transformations (recreating links and forms, λ -lifting, naming intermediate results) to generate an expression in TINYLINKS. The syntax is given in the display below; it is in a minimal style, reminiscent of A-normal form (Sabry and Felleisen 1993), where the **var** expression is the principal way of sequencing computations. Phrases of syntax are identified up to α -conversion, that is, up to consistent renaming of bound variables. Thus, we write $E = E'$ to mean that E and E' are α -equivalent.

Values and Expressions of TINYLINKS:

f, y, x	Variables
p	Predicate symbol
W	Annotation, see Section 4
$c ::=$	Data type constructor
Unit	unit constructor
Zero Succ	integer constructors
String	string constructor
Nil Cons	list constructors
Tuple	tuples
Elem Text	HTML constructors
$g ::=$	Primitive functions
$+ \mid - \mid * \mid / \mid \text{random}$	arithmetic functions
$\text{intToXml} \mid \text{stringToInt}$	type conversions
$L ::= p(V_1, \dots, V_n)$	Event: tag p with a list of values
$V, U ::=$	Value
x	variable

$c(V_1, \dots, V_n)$	constructor
$\lambda x_1, \dots, x_n. E$	abstraction
href (E)	link
form ($[\ell_1, \dots, \ell_n], E$)	form
$E ::=$	Expression
V	value
$(E:W)$	type and effect annotation
var $x = E_1; E_2$	variable binding
$g(U_1, \dots, U_n)$	primitive application
$V(U_1, \dots, U_n)$	function application
switch (V) {	
case $c(x_1, \dots, x_n) \rightarrow E_1$	pattern matching
case $_ \rightarrow E_2$	
}	
get (V)	get request
post ($(\ell_i = V_i)^{i \in 1..n}, U$)	post request
event L	mark an event
assert L	assertion of a prior event

The syntactic category of values includes variables, constructor applications, λ -abstractions and two HTML constructs for creating forms and links. Booleans, integers and strings are not taken as primitive but are recovered through standard encodings using data type constructors and constructor applications. We have **Zero** and **Succ** to represent integers, and similarly, we represent lists of values using **Nil** and **Cons**. Strings become lists of characters, and characters are mapped to integers through a one-to-one correspondence (ASCII).

We express HTML pages are values created by applying data constructors (**Elem**, **Text**) to value arguments that may contain suspended expressions as follows:

- An **href** value represents a link which, when clicked, evaluates the suspended expression E . The evaluation request for the expression is implemented using a GET message.
- A **form** value represents an HTML form with a suspended computation that requires additional user input to proceed. The labels represent the available input fields a client can provide or modify, both visible and hidden; we treat ℓ_i as bound variables which are pairwise distinct with scope E . The evaluation request for a form is implemented using a POST message.

Next, we informally describe the semantics of TINYLINKS.

- A value V is fully evaluated.
- An expression with a type and effect annotation behaves as the expression on its own. We illustrate the usage of these types in Section 4.
- To evaluate a variable binding, we first evaluate the expression E_1 until it yields a value V and we subsequently substitute V for the variable x in the expression E_2 .
- To evaluate a function application $V(U_1, \dots, U_n)$, we ensure that V is a λ -abstraction of the form $\lambda x_1, \dots, x_n. E$. The application evaluates to E with the U_i substituting for the free occurrences of the variable x_i in the expression.
- To evaluate a pattern match, we unify the value V with the pattern $c(x_1, \dots, x_n)$. If V is a matching constructor applied to the same number of values, $c(V_1, \dots, V_n)$, the whole switch expression evaluates to the expression E_1 with the corresponding values V_i substituting the free variables x_i . If the unification fails the whole expression evaluates to E_2 .
- To evaluate a **get** expression, we check that the value provided is an **href**(E) and we evaluate the expression E .

- To evaluate a **post** expression, we check that the value provided is a form, **form**($[\ell_1, \dots, \ell_n], E$). We then proceed to evaluate the embedded expression E with the values V_i substituting the free labels ℓ_i .
- The annotations **assert** L and **event** L return **Unit** at once.

The annotations **assert** L and **event** L have no computational significance, and are included in TINYLINKS simply to express certain safety properties. We say an expression is *safe* to mean that whenever an assertion **assert** L occurs in an execution, there is a previous occurrence within the execution of an event **event** L . Such properties are known as (non-injective) correspondences (Woo and Lam 1993), for example. They are widely used for specifying integrity properties of security mechanisms (Gollmann 2003). Here, we use correspondences in Section 4 to express integrity properties of web applications.

In subsequent sections, we treat the language syntax liberally by relying on special derived forms. Unit, lists and tuples are written using conventional shorthand notation but it is understood that the underlying implementation makes use of the corresponding constructors. We lift expressions occurring inside constructor applications and we bind them to freshly generated variables. Tuple projection uses pattern matching and sequencing of expressions is defined by binding to a freshly generated anonymous variable. Multi-case switch expressions are derived by nested two-case switch ones; pattern matches with a default case are empty.

2.4 An Example Web Application and a Client

We introduce programming in LINKS through an example; even this simple example is susceptible to attacks. The following program represents, in abstract, the process of selling an item at an agreed price. A user requests the web page and is presented with a single button. We assume the price is pre-agreed and thus is hard-coded in the source code. Once the user clicks the buy button the subsequent page is fetched displaying the price to be paid.

The program example declares two functions running on the server. The **sellAt** function represents the sale offer page; it takes an integer value as its input and produces a web page, containing a form with a button. The **onsubmit** attribute of the form tag is annotated with the `l` namespace, to instruct the compiler that the text contained in the curly braces is not a literal string but rather a LINKS expression. Furthermore, this expression is not to be evaluated immediately, but only after the form is submitted. The **buy** function represents the confirmation page that a user sees after clicking the buy button. It receives an integer price value and a database password argument, does a simple conversion to XML and returns the XML value as its output.

A Simple Web Application: Sale

```

fun buy(value, dbpass) server {
  intToXml(value) # omitting actual call to the database
}
fun sellAt(price) server {
  var dbpass = "secret";
  <form l:onsubmit="{buy(price, dbpass)}" method="POST">
    <button type="submit">Buy</button>
  </form>
}
sellAt(42)

```

After the initial request for the program which is implemented as a GET request, the server returns a response with the required web page containing an empty form with a single button. Clicking the button generates a POST request that includes a hidden variable `_k` and the user is finally presented with the result, 42. It is clear, even by inspecting the source code, that if the program terminates, the

only possible output of any interaction with it would be the number 42 being displayed in a web page.

We have included **get** and **post** expressions within TINYLINKS so we may formally express the browsing behaviour of users as TINYLINKS expressions. Let a *client* be any expression context E_{client} within TINYLINKS containing a *hole* of the form **href**(-), that represents the web application that it interacts with. The idea is that the value **href**(E_{url}) represents a link to the main page of the web application E_{url} . The expression $E_{client}[E_{url}]$ obtained by filling the hole in E_{client} with E_{url} is a formal representation of the client E_{client} browsing the web application E_{url} .

For example, here is a client for our example that requests the main page, which consists of a single form, and submits it.

An Example of a Client within TINYLINKS:

```
var app = href(-)
var p = get(app)
var p' = post(p)
p'
```

The reason we model web clients as expression contexts is to reduce source-based reasoning about the security properties of a web application E_{url} to a formal question: for all client contexts E_{client} , does $E_{client}[E_{url}]$ enjoy the intended property.

As a very simple example, we expect that for all client contexts E_{client} , whenever **intToXml**(value) occurs in any execution of $E_{client}[E_{url}]$, value is 42. Given the simple functional semantics of LINKS this sort of property may easily be established by inspection or by some suitable static analysis (such as in Section 4). Unfortunately, as we discuss next, such source-based reasoning is not preserved by LINKS.

3. Some Attacks on the LINKS System

With sufficient knowledge about how the system works, a rogue client working at the HTTP level can examine and modify the continuation strings embedded within web pages to mount at least three kinds of attack: the client (1) may obtain hidden information stored by LINKS in pages on the client, (2) may modify such information, and (3) may construct function calls that did not exist in the source program. We refer to these as failures of secrecy, data integrity, and control integrity.

3.1 An Attack on Secrecy

We demonstrate an attack on secrecy using the sale example from Section 2.4. A client makes an initial request for the example using the GET method. The server responds with the intermediate page that contains a single form with the buy button. The following figure contains the generated HTML.

Translated form from Sale

```
<form method="post" action="#">
  <input type="hidden" name="_k"
    value="base64((hash(buy(price,dbpass))), [price=42,
      dbpass="secret"])" />
  <button type="submit">Buy</button>
</form>
```

The translation is straightforward; forms are directly translated to their corresponding HTML elements and the hidden variable **_k** holds the continuation string. For clarity we represent the continuation string symbolically, it is a base64 encoding of the expression hash of the **onsubmit** attribute, along with an environment that binds **price** to 42 and **dbpass** to "secret". The implementation stores the secret value of **dbpass** in plain text inside the serialised environment as it happened to be in scope.

3.2 An Attack on Data Integrity

A more sophisticated attack on the previous program might involve modifying parts of continuation string. We can rewrite the continuation replacing it with the following one, in which the binding of **price** is 0 in the environment (e.g. **[price=0,dbpass="secret"]**). By clicking on the buy button, we submit a POST request with the updated continuation and force the server to evaluate the original expression under the counterfeit environment.

3.3 An Attack on Control Integrity

The unreachable example demonstrates how an attacker can make calls that weren't in the source level program. The program presents the user with a link to click, and subsequently displays a page a number. The implementation declares a function **page** with two local functions **f** and **g** corresponding to a reachable page and an unreachable one. It also contains a link that upon clicking will display the reachable page. Note that the variable **g** corresponding to the unreachable page is bound to **w** inside the link's expression but never used subsequently in the body of the **var**-expression.

Unreachable page example

```
fun page(x) server {
  var f = fun () { <html> A page with {intToXml(x)} </html> };
  var g = fun () {
    <html> An unreachable page with {intToXml(x)} </html> };
  <a href="{var w=g; f()}"> Click for reachable page </a>
}
page(42)
```

Writing the translation of the intermediate page in symbolic form we discover that the closure for the **var**-expression contained in the link has an environment binding for the variable **g** which is never used in the body of the expression. By replacing the contents of the **href** with **base64(hash(g()), [g=fun])** one can now see the unreachable page. This creates a way for an attacker to call the function **g**; yet, such a call doesn't exist in the source code.

Translation of unreachable page

```
<a href="base64((hash(var w=g; f())), [f=fun,g=fun])">
  Click for reachable page </a>
```

4. A Type-and-Effect system for LINKS

In subsequent sections, Sections 5 and 6, we formalize the HTTP-based implementation described and attacked above, and also a simple cryptographic mechanism intended to protect against these attacks. We aim not just to show that a few specific attacks fail, but to show that the cryptography protects whole classes of properties provable at the source-level.

The purpose of this section is to define an exemplary source-based analysis that is preserved by the secure translation. We describe a dependent type-and-effect system for proving the assertion-based safety properties described in Section 2, that whenever an assertion **assert** L occurs in an execution, there is a previous occurrence of an event **event** L . Data integrity and control integrity properties, such as those attacked in the previous section, are provable within our system.

We have implemented a typechecker for our system and checked a range of LINKS programs.

The design of our type-and-effect system is inspired by a simple system for typing correspondences in a process calculus (Gordon and Jeffrey 2003); we expect our results could be extended to more sophisticated program analyses or logics. For the sake of a simple presentation, our system lacks parametric polymorphism, a feature of LINKS itself, but it could easily be added.

The syntax of types and effects is as follows. The intention is that a type describes a value, and a type-and-effect describes an expression.

Syntax of types, effects and environments

$F ::= L_1, \dots, L_m$	Effect: a set of events
$W ::= \langle x:T \rangle \{F\}$	(monadic) Type-and-Effect
$P ::= \langle x_1:T_1 \dots x_n:T_n \rangle \{F\}$	polyadic Type-and-Effect
$B ::= \text{unit} \mid \text{int} \mid \text{string} \mid \text{xml}$	Base Types
$S, T, H ::=$	Types
B	base type
$[T]$	list
$T_1 \times \dots \times T_n$	tuple
$P \rightarrow W$	polyadic function
$\Gamma ::= x_1:T_1, \dots, x_n:T_n$	Environment
$\text{dom}(x_1:T_1, \dots, x_n:T_n) = \{x_1, \dots, x_n\}$	

Types consist of base types, lists, tuples and function types. A type-and-effect expression is formed by a sequence of monomorphic types along with an effect. Intuitively, it captures the evaluation properties of expression E which may have the observable effect in W . In a (monadic or polyadic) type-and-effect expression $\langle x_1:T_1 \dots x_n:T_n \rangle \{F\}$ a variable x_i is bound with scope T_{i+1}, \dots, T_n and F . In function types $\langle x_1:T_1 \dots x_n:T_n \rangle \{F_1\} \rightarrow \langle x:T \rangle \{F_2\}$ the variables x_1, \dots, x_n are bound with scope F_1 and F_2 , whereas the variable x is bound with scope F_2 . An environment is a list of variables along with their associated types. We write $\text{dom}(\Gamma)$ for the domain of an environment and \emptyset for the empty one.

The type-and-effect system consists of a set of inductively defined algorithmic judgements, relative to a typing environment, Γ , and an effect, F . Assigning a type-and-effect $W = \langle x:T \rangle \{F'\}$ to an expression means that, assuming that the set of events in T have occurred, evaluation of the expression is safe, and if the expression yields a value x , it has type T , and afterwards we may assume the events in F' have occurred. Hence, the effect F is a precondition, a set of events assumed to have occurred before execution, and the effect F' is a postcondition, a set of events safe to assume after execution. The rules are in bidirectional style (Pierce and Turner 1998) and correspond directly to our implementation.

Judgements

$\Gamma \vdash \diamond$	Γ is well-formed
$\Gamma; F \vdash V \overset{\vee}{\rightarrow} T$	value V synthesises output type T
$\Gamma; F \vdash V \overset{\checkmark}{\leftarrow} T$	value V type-checks against input T
$\Gamma; F \vdash E \overset{\rightarrow}{\rightarrow} W$	expression E synthesises output W
$\Gamma; F \vdash E \overset{\leftarrow}{\leftarrow} W$	expression E type-checks against input W

The superscripts on the synthesis and checking arrows serve no other purpose than to disambiguate between the functions applied to values and those applied to expressions.

The following tables contain the typing rules for the type-and-effect system.

Well-formed Environment: $\Gamma \vdash \diamond$

(Env \emptyset)	(Env Ext)
$\Gamma \vdash \diamond$	$x \notin \text{dom}(\Gamma) \quad \text{fv}(T) \subseteq \text{dom}(\Gamma)$
$\emptyset \vdash \diamond$	$\Gamma, x:T \vdash \diamond$

According to these rules, in a well-formed environment the variables in the domain of the environment are distinct and every variable occurring free in a type is mentioned in the domain.

The constructors in our language have a fixed arity and type. We formalise this by providing a relation with the intended types.

Constructor instances $c : (T_1, \dots, T_n) \rightarrow T$

Nil : $() \rightarrow [T]$
Cons : $(T, [T]) \rightarrow [T]$
Zero : $() \rightarrow \text{int}$
Succ : $(\text{int}) \rightarrow \text{int}$
String : $([\text{int}]) \rightarrow \text{string}$
Tuple : $(T_1, \dots, T_n) \rightarrow (T_1 \times \dots \times T_n)$
Unit : $() \rightarrow \text{unit}$
Elem : $(\text{string}, [\text{xml}]) \rightarrow \text{xml}$
Text : $(\text{string}) \rightarrow \text{xml}$

Next, we present the synthesis and checking rules for values.

Algorithmic typing rules for values (synthesis): $\Gamma; F \vdash V \overset{\vee}{\rightarrow} T$

(T-Var)	$\Gamma \vdash \diamond \quad \text{fv}(F) \subseteq \text{dom}(\Gamma) \quad \Gamma = \Gamma', x:T, \Gamma''$	$\Gamma; F \vdash x \overset{\vee}{\rightarrow} T$
(T-Const)	$c : (T_1, \dots, T_n) \rightarrow T$ $c \in \{\text{Unit}, \text{Zero}, \text{Succ}, \text{String}, \text{Elem}, \text{Text}\}$	$\Gamma; F \vdash V_i \overset{\checkmark}{\leftarrow} T_i \quad \forall i \in 1..n$
(T-Href)	$\Gamma; F \vdash E \overset{\leftarrow}{\leftarrow} \langle _ : \text{xml} \rangle \{ \}$	$\Gamma; F \vdash \text{href}(E) \overset{\vee}{\rightarrow} \text{xml}$
(T-Form)	$\Gamma, \ell_1:\text{string}, \dots, \ell_n:\text{string}; F \vdash E \overset{\leftarrow}{\leftarrow} \langle _ : \text{xml} \rangle \{ \}$ $\ell_1 \dots \ell_n \notin \text{fv}(F)$	$\Gamma; F \vdash (\text{form}([\ell_1, \dots, \ell_n], E)) \overset{\vee}{\rightarrow} \text{xml}$

By (T-Var), we synthesize a type for a variable by looking up its type in the typing environment. By (T-Const), to synthesize a type for a constructor application, we match the constructor against a specific instance with a monomorphic argument type. We then proceed to type check each value supplied to the constructor against the types read from the constructor instance. By (T-Href), an **href** value has type **xml**, provided that the embedded expression is also of type **xml**. By (T-Form), a **form** value has type **xml**, provided that the embedded expression is also of type **xml**, in an extended environment with all parameters having type **string**.

Algorithmic typing rules for values (checking): $\Gamma; F \vdash V \overset{\checkmark}{\leftarrow} T$

(T-Abs)	$\Gamma, x_1:T_1 \dots x_n:T_n; F, F_1 \vdash E \overset{\leftarrow}{\leftarrow} W$ $T = \langle x_1:T_1 \dots x_n:T_n \rangle \{F_1\} \rightarrow W \quad x_1, \dots, x_n \notin \text{fv}(F), T \text{ closed}$	$\Gamma; F \vdash (\lambda x_1, \dots, x_n. E) \overset{\checkmark}{\leftarrow} T$
(T-Const-S)	$c : (T_1, \dots, T_n) \rightarrow T$ $c \in \{\text{Nil}, \text{Cons}, \text{Tuple}\}$	(T-Swap) $\Gamma; F \vdash V \overset{\vee}{\rightarrow} T$ $\Gamma; F \vdash V_i \overset{\checkmark}{\leftarrow} T_i \quad \forall i \in 1..n$ $\Gamma; F \vdash V \overset{\checkmark}{\leftarrow} T$

By (T-Abs), to type check a λ -abstractions against an arrow type, we ensure that the body of the abstraction typechecks against the result type of the arrow, in an extended environment with function arguments given the respective types from the arrow argument and the effect being extended with the corresponding effect from the type. By (T-Const-S), during type checking of a type constructor that has a polymorphic return type (**Nil, Cons, Tuple**), we can infer the source types and subsequently type check the applied values against this inferred type. By (T-Swap), the last rule type checking for values, turns the type checking problem into a type synthe-

sis one. This saves duplication of inter-derivable rules in both type checking and the type synthesis definitions.

We present the algorithmic typing rules to associate expressions with types-and-effects.

Algorithmic rules for expressions (synthesis): $\Gamma; F \vdash E \xrightarrow{E} W$

(T-Source) $\frac{\Gamma; F \vdash E \xrightarrow{E} W}{\Gamma; F \vdash (E; W) \xrightarrow{E} W}$ (T-Val) $\frac{\Gamma; F \vdash V \xrightarrow{V} T}{\Gamma; F \vdash V \xrightarrow{E} \langle _ : T \rangle \{ F \}}$

(T-Switch) $\frac{c : (T_1, \dots, T_n) \rightarrow T \quad \Gamma; F \vdash V \xrightarrow{V} T \quad \Gamma, x_1:T_1, \dots, x_n:T_n; F \vdash E_1 \xrightarrow{E} W \quad \Gamma; F \vdash E_2 \xrightarrow{E} W}{\Gamma; F \vdash \left(\begin{array}{l} \text{switch}(V) \{ \\ \text{case } c(x_1, \dots, x_n) \rightarrow E_1 \\ \text{case } _ \rightarrow E_2 \} \end{array} \right) \xrightarrow{E} W}$

(T-Bind) $\frac{\Gamma; F \vdash E_1 \xrightarrow{E} \langle x:T_1 \rangle \{ F_1 \} \quad \Gamma, x:T_1; F, F_1 \vdash E_2 \xrightarrow{E} W \quad x \notin \text{fv}(W)}{\Gamma; F \vdash (\text{var } x = E_1; E_2) \xrightarrow{E} W}$

(T-App) $\frac{\Gamma; F \vdash U \xrightarrow{V} T \quad T = \langle x_1:T_1 \dots x_n:T_n \rangle \{ F_1 \} \rightarrow W \quad T \text{ closed} \quad \Gamma; F \vdash V_i \xrightarrow{V} T_i \quad \forall i \in 1..n \quad F_1[V_1/x_1] \dots [V_n/x_n] \subseteq F}{\Gamma; F \vdash U(V_1, \dots, V_n) \xrightarrow{E} W[V_1/x_1] \dots [V_n/x_n]}$

(T-Get) $\frac{\Gamma; F \vdash V \xrightarrow{V} \text{xml}}{\Gamma; F \vdash \text{get}(V) \xrightarrow{E} \langle _ : \text{xml} \rangle \{ \}}$

(T-Post) $\frac{\Gamma; F \vdash V_i \xrightarrow{V} \text{string} \quad \Gamma; F \vdash U \xrightarrow{V} \text{xml}}{\Gamma; F \vdash (\text{post}((\ell_i = V_i)^{i \in 1..n}, U)) \xrightarrow{E} \langle _ : \text{xml} \rangle \{ \}}$

(T-Assert) $\frac{\Gamma \vdash \diamond \quad \text{fv}(F, L) \subseteq \text{dom}(\Gamma) \quad L \in F \quad L = p(V_1, \dots, V_n) \quad \Gamma; F \vdash V_i \xrightarrow{V} T_i \quad \forall i \in 1..n}{\Gamma; F \vdash \text{assert } L \xrightarrow{E} \langle _ : \text{unit} \rangle \{ L \}}$

(T-Event) $\frac{\Gamma \vdash \diamond \quad \text{fv}(F, L) \subseteq \text{dom}(\Gamma) \quad L = p(V_1, \dots, V_n) \quad \Gamma; F \vdash V_i \xrightarrow{V} T_i \quad \forall i \in 1..n}{\Gamma; F \vdash \text{event } L \xrightarrow{E} \langle _ : \text{unit} \rangle \{ L \}}$

The rule (T-Source) synthesises a type-and-effect W for an annotated expression by checking that the expression type checks against the source level annotation. Rule (T-Val) synthesises a type-and-effect by combining the synthesised type during value synthesis with the effect used as an assumption for the judgement. To synthesise a type for a switch expression we type check the value V against the result type of the constructor c . We then ensure that the expressions of both branches synthesise the same type-and-effect, which is the synthesised type of the entire expression. In the case of variable binding (T-Bind), we synthesise a type-and-effect for E_1 . We then synthesise one for E_2 while extending the typing environment with a type for the bound variable and the overall effect with the effect of E_1 . The rule (T-App) synthesises a type-and-effect from a polyadic function application. By (T-Get), the type for the `get` operation is always of type `xml` with an empty effect, provided that the embedded value is of type `xml`. By (T-Post), the `post` operation has always got type `xml` with an empty effect, provided that the

values associated with the submission labels are of type `string` and the embedded value of type `xml`. The rules (T-Event) for `event L` and (T-Assert) for `assert L` are the same, except that (T-Assert) requires $L \in F$, where F is the precondition on the judgment.

Algorithmic rules for expressions (checking): $\Gamma; F \vdash E \xleftarrow{E} W$

(T-Subs) $\frac{\Gamma; F \vdash E \xrightarrow{E} \langle \tilde{x}:\tilde{T} \rangle \{ F_2 \} \quad F_1 \subseteq F_2}{\Gamma; F \vdash E \xleftarrow{E} \langle \tilde{x}:\tilde{T} \rangle \{ F_1 \}}$

(T-Switch) $\frac{c : (T_1, \dots, T_n) \rightarrow T \quad \Gamma; F \vdash V \xrightarrow{V} T \quad \Gamma, x_1:T_1, \dots, x_n:T_n; F \vdash E_1 \xleftarrow{E} W \quad \Gamma; F \vdash E_2 \xleftarrow{E} W}{\Gamma; F \vdash \left(\begin{array}{l} \text{switch}(V) \{ \\ \text{case } c(x_1, \dots, x_n) \rightarrow E_1 \\ \text{case } _ \rightarrow E_2 \} \end{array} \right) \xleftarrow{E} W}$

(T-Bind) $\frac{\Gamma; F \vdash E_1 \xrightarrow{E} \langle x:T_1 \rangle \{ F_1 \} \quad \Gamma, x:T_1; F, F_1 \vdash E_2 \xleftarrow{E} W \quad x \notin \text{fv}(F, W)}{\Gamma; F \vdash (\text{var } x = E_1; E_2) \xleftarrow{E} W}$

By (T-Subs), we may drop events from the postcondition F_2 to yield the weaker postcondition F_1 . The rules (T-Switch) and (T-Bind) are counterparts to the synthesis rules (T-Switch) and (T-Bind).

4.1 Expressing Data and Control Integrity with Assertions

We now demonstrate how to use the type and effect system to specify the required properties about data integrity and control flow integrity.

Our working example, `sale`, consists of two pages; one is an offer to sell if the button is clicked and the other is confirmation of the sale at the pre-agreed price. The required security property is that, if the confirmation page is ever reached with a price, the same price was offered in the previous page. To express that, we add an `event` expression with the `PriceIs` event to the page making the offer for the item, and an `assert` expression with the same event in the confirmation page. What we ask for is that, for all possible execution paths of this program, whenever an `assert PriceIs(value)` occurs, there is a previous occurrence of the event `event PriceIs(price)` with the arguments `value` and `price` being equal. The following figure shows concretely the annotated program that we use as input to our typechecker.

Data Integrity with Assertions: Sale

```
sig buy : <value:int, dbpass:string> {PriceIs(value)} → <r:xml> { }
fun buy(value,dbpass) server {
  assert PriceIs(value);
  intToXml(value) # omitting actual call to the database
}
sig sellAt: <price:int> { } → <r:xml> { }
fun sellAt(price) server {
  var dbpass = "secret";
  event PriceIs(price);
  <form l:onsubmit="{buy(price,dbpass)}" method="POST">
  <button type="submit">Buy</button>
  </form>
}
sellAt(42)
```

Similarly, the unreachable function example consists of two pages. The first page presents a link and the second one presents a message that the page was reachable. The required security property is that the additional function `g` declared in the source code is never called, despite being in scope inside the `href` expression. To

express this property we assert an impossible event `Unreachable()` inside the body of the unreachable function. Since there is no corresponding event in the program and the function `g` is never called, the program typechecks correctly.

Control Integrity with Assertions: Unreachable

```
fun page(x) server {
  var f = fun () { <html> A page with {intToXml(x)} </html> };
  var g: <x:unit>{Unreachable()} → <r:xml>{} = fun () {
    assert(Unreachable());
    <html> An unreachable page with {intToXml(x)} </html>
  };
  <a href="{var w=g; f ()}"> Click for reachable page </a>
}
page(42)
```

Note that if you change `f()` to `g()` inside the body of the `href`, then the program is no longer typable (because the precondition on `g` is not satisfied) and indeed no longer safe (because `assert Unreachable()` is reachable with no prior `Unreachable()` event)

5. A Semantics for the Standard Implementation

We give a semantics for the standard implementation of LINKS by translating a well-typed TINYLINKS program E_{url} to an F7 expression $\llbracket E_{url} \rrbracket$.

5.1 Interlude: Refinement Types and Safety in F7

Saying $\llbracket E_{url} \rrbracket$ is an F7 expression means that it is a directly executable F# program, while also being an expression in the concurrent λ -calculus that underpins F7 (Bengtson et al. 2008).

F7 allows any type to be refined with a formula of first-order logic; for example, the *refinement type* $(x : T)\{C\}$ consists of the values x of type T such that formula C holds (C may contain x).

The purpose of refinement types in F7 is to type check safety properties induced by `assume` C and `assert` C expressions, where C is a formula. An F7 expression is *safe* if and only if, whenever `assert` C occurs in an execution, the formula C follows from a set C_1, \dots, C_n of formulas where, for each i , `assume` C_i has previously occurred in the execution. The *safety-by-typing theorem* for F7 is that whenever a closed expression is well-typed, it is safe. The F7 typechecker makes use of an external theorem prover to ensure statically that each asserted formula will be provable. The F7 type system is rather more flexible than the one for TINYLINKS, and hence we can directly interpret the `event` and `assert` expressions of TINYLINKS using `assume` and `assert`.

5.2 Well-Typed Web Applications

We share as much functionality as possible between the standard semantics $\llbracket E_{url} \rrbracket$, of this section, and the secure semantics $\llbracket E_{url} \rrbracket_s$, explained in Section 6. The two translation algorithms are similar with the differences being mentioned in this section and explained in the next.

Throughout the two translations, we consider some fixed well-typed TINYLINKS expression E_{url} , and a structure $\mathcal{W} = (E_{url}, \mathcal{J}, \mathcal{H})$. The expression E_{url} represents the main entry point to the web application; E_{url} is a closed expression of type `xml`. The set \mathcal{J} identifies all functions, links, and forms occurring in E_{url} , together with their types and effects. The set \mathcal{H} records a distinct type constructor H_J for each member $J \in \mathcal{J}$; each H_J corresponds in the implementation to the hashed expression label for the expression part of J (recall Section 2.2).

Web-Typed Web Application: $\mathcal{W} = (E_{url}, \mathcal{J}, \mathcal{H})$

There is a typing derivation $\emptyset; \emptyset \vdash E_{url} \overset{E}{\rightsquigarrow} \langle _ : \text{xml} \rangle \{ \}$ for E_{url} . Set \mathcal{J} records all instances of (T-Abs), (T-HRef), and (T-Form)

in the typing derivation; we have $J \in \mathcal{J}$ if and only if $\Gamma; F \vdash E \overset{E}{\rightsquigarrow} W$ occurs in the typing derivation for E_{url} and $J = (\Gamma, F, E, W)$. Set $\mathcal{H} = \{H_J \mid J \in \mathcal{W}\}$ is a set of distinct constructor names.

As a shorthand for referring to the global typing derivation \mathcal{J} , we sometimes write a subexpression of E_{url} with a type subscript to indicate the type assigned by \mathcal{J} ; for example, we may write $U_T(V_1, \dots, V_n)$ for an application where function U has type T .

5.3 Modelling a Web Server hosting LINKS

We begin by formalizing the client's view of a LINKS-based web application as a function type $(\text{'g}, \text{'p})\text{webapp}$, which sends HTTP requests to XML-encoded HTML. The type parameters `'g` and `'p` stand for the types of continuations attached to GET and POST methods, as discussed in Section 2.2. In the standard implementation, the continuation types are algebraic types `linkclos` and `formclos` (given subsequently) representing possible closures of subexpressions of E_{url} . (In the next section, continuations are encrypted closures.)

Types for HTTP, XHTML, and Web Applications:

```
type ('g, 'p) req =
  | Get of 'g option
  | Post of 'p * string list
```

```
type ('g, 'p) xml =
  | Elem of string * ('g, 'p) xml list
  | Text of string
  | Href of 'g
  | FormElem of 'p * string list
```

```
type ('g, 'p) webapp = ('g, 'p) req → ('g, 'p) xml
```

Values of type $(\text{'g}, \text{'p})\text{xml}$ are abstract representations of XML (in fact, XHTML) web pages. A value `Elem`(s, xs) represents an XML element named s , with children xs . A value `Text`(s) represents actual text s . A value `Href`(g) represents an HTML link to a TINYLINKS expression encoded by the continuation g . A value `FormElem`(p, ls) represents an HTML form, with parameters ls , and that when submitted produces a page from a TINYLINKS expression encoded by the continuation p . (For full LINKS we would need a more accurate XML model, but this suffices for TINYLINKS.)

The algebraic type $(\text{'g}, \text{'p})\text{req}$ models HTTP requests. A message `Get`(`None`) is a GET for the main page, that is, the one obtained by evaluating E_{url} . A message `Get`(`Some`(g)), is a GET for a page referenced by a link `Href`(g), that is itself contained within a previously obtained page, such as the main page. A message `Post`(p, ss) is a POST on a form `FormElem`(p, ls), filling in the values ss for the parameters ls . We can omit any explicit URL from these messages, as all the pages described by a web application E_{url} are at the same base URL on the same server.

5.4 The Standard Semantics

The translation algorithm proceeds as follows:

- The first step is to perform type-directed closure conversion on all the λ -abstractions, forms and links occurring in the source and generate suitable datatypes for representing them in F7.
- Generate mutually recursive function listeners (fH_j); each corresponding to the closures that were generated previously.
- Finally, translate the top level web server listener.

We begin with a translation of TINYLINKS types and the generation of the following F7 types for closures: `funclos` $_{P \rightarrow W}$ has a constructor H_J for each λ -abstraction of type $P \rightarrow W$ occurring in E_{url} ;

formclos has a constructor H_J for each **form**-expression occurring in E_{url} ; and **linkclos** has a constructor H_J for each **form**-expression occurring in E_{url} ;

Translation of Types, Effects, and Environments:

$$\begin{aligned} \llbracket \text{int} \rrbracket &= \text{int} \\ \llbracket \text{string} \rrbracket &= \text{string} \\ \llbracket \text{unit} \rrbracket &= \text{unit} \\ \llbracket \text{xml} \rrbracket &= (\text{linkclos}, \text{formclos}) \text{xml} \\ \llbracket [T] \rrbracket &= [T] \text{ist} \\ \llbracket T_1 \times \dots \times T_n \rrbracket &= [T_1] \times \dots \times [T_n] \\ \llbracket P \rightarrow W \rrbracket &= \text{funclos}_{P \rightarrow W} \\ \llbracket \langle x_1:T_1 \dots x_n:T_n \rangle \{F\} \rrbracket &= (x_1:[T_1] * \dots * x_n:[T_n]) \{F\} \\ \llbracket \Gamma \rrbracket &= x_1:[T_1], \dots, x_n:[T_n] \quad \text{if } \Gamma = x_1:T_1, \dots, x_n:T_n \\ \llbracket \Gamma; F \rrbracket &= (x_1:[T_1] * \dots * x_n:[T_n]) \{F\} \quad \text{if } \Gamma = x_1:T_1, \dots, x_n:T_n \end{aligned}$$

Generated datatypes:

$$\begin{aligned} \text{type } \text{funclos}_{P \rightarrow W} &= \\ &\sum \left\{ H_J \text{ of } \llbracket \Gamma; F \rrbracket \mid J = (\Gamma, F, (\lambda x_1, \dots, x_n.E), P \rightarrow W) \wedge J \in \mathcal{J} \right\} \\ \text{and } \text{formclos} &= \\ &\sum \left\{ H_J \text{ of } \llbracket \Gamma; F \rrbracket \mid J = (\Gamma, F, \text{form}(\llbracket \ell_1, \dots, \ell_m \rrbracket, E), \text{xml}) \wedge J \in \mathcal{J} \right\} \\ \text{and } \text{linkclos} &= \sum \left\{ (H_J \text{ of } \llbracket \Gamma; F \rrbracket) \mid J = (\Gamma, F, \text{href}(E), \text{xml}) \wedge J \in \mathcal{J} \right\} \end{aligned}$$

We rely on F7 refinement types to record effects. We regard a TINYLINKS effect F as an F7 formula: the conjunction of the formulas obtained by interpreting each event $L = p(V_1, \dots, V_n)$ in F as a predicate p with parameters V_1, \dots, V_n . The F7 translation $(x_1:[T_1] * \dots * x_n:[T_n]) \{F\}$ of a type-and-effect is an n -ary tuple refined with F treated as a formula. The F7 translation $\llbracket \Gamma; F \rrbracket$ of an environment with a pre-condition is similar.

Next, we define the translation of values and expressions. When a value V is a λ -abstraction, link or form, we emit the data type constructor H_J where J is the typing judgment for V ; the constructor H_J corresponds to the hashed expression label for V in the actual implementation. The function $\mathcal{C}[V]$ defines how each constructor takes a record argument representing the environment. (The LINKS system and our implementation actually optimize what is included in each closure, by omitting top-level values and library functions.)

Translation of Values: $\llbracket V \rrbracket$

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket \text{Unit} \rrbracket &= () \\ \llbracket \text{Zero} \rrbracket &= 0 \\ \llbracket \text{Succ}(V) \rrbracket &= [V] + 1 \\ \llbracket \text{Nil} \rrbracket &= \text{nil} \\ \llbracket \text{Cons}(V, V') \rrbracket &= V :: [V'] \\ \llbracket \text{Tuple}([V_1; \dots; V_n]) \rrbracket &= ([V_1], \dots, [V_n]) \\ \llbracket c(V_1, \dots, V_n) \rrbracket &= c([V_1], \dots, [V_n]) \\ \llbracket (\lambda x_1, \dots, x_n.E) \rrbracket &= \mathcal{C}[\lambda x_1, \dots, x_n.E] \\ \llbracket \text{href}(E) \rrbracket &= \text{Href}(\mathcal{C}[\text{href}(E)]) \\ \llbracket \text{form}(\llbracket \ell_1, \dots, \ell_n \rrbracket, E) \rrbracket &= \\ &\quad \text{FormElem}(\mathcal{C}[\text{form}(\llbracket \ell_1, \dots, \ell_n \rrbracket, E)], \llbracket \ell_1, \dots, \ell_n \rrbracket) \\ \mathcal{C}[V] &= H_J(\langle x_1, \dots, x_n \rangle) \quad \text{for } J = ((x_i:U_i)^{i \in 1..n}, F, V, T) \wedge J \in \mathcal{J} \end{aligned}$$

Translation of Expressions: $\mathcal{E}[E]$

$$\begin{aligned} \mathcal{E}[V] &= [V] \\ \mathcal{E}[\text{var } x = E_1; E_2] &= \text{let } x = \mathcal{E}[E_1] \text{ in } \mathcal{E}[E_2] \end{aligned}$$

$$\begin{aligned} \mathcal{E}[g(U_1, \dots, U_n)] &= g [U_1] \dots [U_n] \\ \mathcal{E}[\llbracket x_T(U_1, \dots, U_n) \rrbracket] &= \text{match } x \text{ with} \\ &\quad \sum \left\{ (H_J(g) \rightarrow f_{H_J} g \llbracket (U_1, \dots, U_n) \rrbracket) \mid J = (\Gamma, F, V, T) \wedge J \in \mathcal{J} \right\} \\ \mathcal{E}[\llbracket (\lambda x_1, \dots, x_n.E)(U_1, \dots, U_n) \rrbracket] &= E \{U_1/x_1\} \dots \{U_n/x_n\} \\ \mathcal{E}[E] &= \left(\text{match } [V] \text{ with} \right. \\ &\quad \left. c(x_1, \dots, x_n) \rightarrow \mathcal{E}[E_1] \text{ else } \mathcal{E}[E_2] \right) \\ &\quad \text{where } E = \left(\text{switch}(V) \left\{ \begin{array}{l} \text{case } c(x_1, \dots, x_n) \rightarrow E_1 \\ \text{case } _ \rightarrow E_2 \end{array} \right\} \right) \\ \mathcal{E}[\text{get}(V)] &= \left(\text{match } [V] \text{ with } \text{Href}(l) \rightarrow \right. \\ &\quad \left. \text{webserver}(\text{Get}(\text{Some}(l))) \right) \\ \mathcal{E}[\text{post}(\llbracket (l_i = V_i)^{i \in 1..n}, U \rrbracket)] &= \\ &\quad \left(\text{match } [U] \text{ with } \text{FormElem}(f, l_s) \text{ when } l_s = [l_1, \dots, l_n] \rightarrow \right. \\ &\quad \left. \text{webserver}(\text{Post}(f, \llbracket (l_i, [V_i])^{i \in 1..n} \rrbracket)) \right) \\ \mathcal{E}[\text{event } L] &= \text{assume } L \\ \mathcal{E}[\text{assert } L] &= \text{assert } L \end{aligned}$$

The translation of expressions assumes the following functions f_{H_J} , for unpacking and evaluating closures, and also a top-level function **webserver** defined below.

Generated Functions:

$$\begin{aligned} f_{H_J} : \llbracket (x_i:U_i)^{i \in 1..n}; F \rrbracket \rightarrow [P] \rightarrow [W] \\ \text{let rec } f_{H_J} g y = \\ &\quad \text{match } g \text{ with } (x_1, \dots, x_n) \rightarrow \\ &\quad \text{match } y \text{ with } (y_1, \dots, y_n) \rightarrow \mathcal{E}[E] \\ &\quad \text{where } J = ((x_i:U_i)^{i \in 1..n}, F, \lambda x_1, \dots, x_n.E, P \rightarrow W) \text{ and } J \in \mathcal{J} \\ f_{H_J} : \llbracket (x_i:U_i)^{i \in 1..n}; F \rrbracket \rightarrow \text{xml} \\ \text{and } f_{H_J} g = \text{match } g \text{ with } (x_1, \dots, x_n) \rightarrow \mathcal{E}[E] \\ &\quad \text{where } J = ((x_i:U_i)^{i \in 1..n}, F, \text{href}(E), \text{xml}) \text{ and } J \in \mathcal{J} \\ f_{H_J} : \llbracket (x_i:U_i)^{i \in 1..n}; F \rrbracket \rightarrow \text{string list} \rightarrow \text{xml} \\ \text{and } f_{H_J} g l_s = \\ &\quad \text{match } g \text{ with } (x_1, \dots, x_n) \rightarrow \\ &\quad \text{match } l_s \text{ with } [\ell_1; \dots; \ell_n] \rightarrow \mathcal{E}[E] \\ &\quad \text{where } J = ((x_i:U_i)^{i \in 1..n}, F, \text{form}(\llbracket \ell_1, \dots, \ell_n \rrbracket, E), \text{xml}) \text{ and } J \in \mathcal{J} \end{aligned}$$

The top-level **webserver** listener accepts the possible GET and POST requests and redirects them appropriately to their corresponding function listeners. A GET request with no given link closure means we are dealing with the initial request for the application, so we should return the translation of the top level expression. If the request was a GET with a particular link closure l we determine which closure constructor H_J corresponds to this closure and we call the corresponding function with the closing environment g .

In the case of a POST request we perform a similar operation matching the form closure against the constructor H_J with the additional provision of also breaking up the list of additional arguments that were part of the form.

Top level web server listener:

$$\begin{aligned} \text{let } \text{webserver req} : (\text{linkclos}, \text{formclos}) \text{webapp} = \\ \text{match req with} \\ \mid \text{Get}(\text{None}) \rightarrow \mathcal{E}[E_{url}] \\ \mid \text{Get}(\text{Some}(l)) \rightarrow \\ &\quad \text{match } l \text{ with} \\ &\quad \prod_{J \in \mathcal{J} \wedge J = (\Gamma, F, \text{href}(E), T)} ((H_J(g) \rightarrow f_{H_J} g)) \\ \mid \text{Post}(\text{clos}, l_s) \rightarrow \\ &\quad \text{match clos, } l_s \text{ with} \\ &\quad \prod_{J \in \mathcal{J} \wedge J = (\Gamma, F, \text{form}(\llbracket \ell_1, \dots, \ell_n \rrbracket, E), T)} \left(\mid H_J(g), [\ell_1; \dots; \ell_n] \rightarrow \right. \\ &\quad \left. f_{H_J} g [\ell_1; \dots; \ell_n] \right) \end{aligned}$$

The following summarizes the formal translation (and how our TINYLINKS typechecker also generates an actual F7 program):

Translation from E_{url} in TINYLINKS to $\llbracket E_{url} \rrbracket$ in F7:

Let $\llbracket E_{url} \rrbracket$ be the F7 expression obtained from E_{url} consisting of the: (1) fixed datatypes; (2) generated datatypes; (3) generated functions; (4) toplevel `webserver` function.

Lemma 1. *Suppose that $\emptyset; \emptyset \vdash E_{url} \overset{E}{\sim} \langle _ : \text{xml} \rangle \{ \}$. Then $\llbracket E_{url} \rrbracket$ is a closed expression of F7 of type: $\llbracket E_{url} \rrbracket : (\text{linkclos}, \text{formclos})\text{webapp}$.*

Theorem 1. *If E_{url} is provably safe at the source level, then the (standard) `webserver` $\llbracket E_{url} \rrbracket$ is provably safe.*

This theorem is a corollary, given the safety-by-typing theorem of F7 (Section 5.1), of Lemma 1.

Suppose we provide $\llbracket E_{url} \rrbracket$ to a well-typed F7 context, representing a well-behaved HTTP-level user of the web application. The force of the theorem is that the context may invoke $\llbracket E_{url} \rrbracket$ with arguments of type $(\text{linkclos}, \text{formclos})\text{req}$ yielding $(\text{linkclos}, \text{formclos})\text{xml}$ results, and no matter what happens, no `assert` in $\llbracket E_{url} \rrbracket$ can fail. Hence, Theorem 1 shows soundness of our type-and-effect system for TINYLINKS.

6. Semantics of a Secure Implementation

Of course, it is naive to imagine that your opponent across the network is so polite as to obey typing rules. A better model of a potentially untrustworthy HTTP-level user in F7 is as an arbitrary (not necessarily well-typed) expression context (as in the spi calculus and subsequent work). Although our semantics $\llbracket E_{url} \rrbracket$ abstracts some low level details, we can still express each of the attacks from Section 3 as untyped contexts.

This section formalizes our use of cryptography to protect against these attacks, and establishes our main result, that the secured model $\llbracket E_{url} \rrbracket_s$ is safe even when placed in an untyped context.

6.1 Interlude: Public Types and Robust Safety in F7

Let an *opponent* be an arbitrary F7 expression context. We say an F7 expression is *robustly safe* if it is safe whenever it is placed within any opponent context. A significant result concerning F7 is the *robust-safety-by-typing theorem*: that a closed well-typed expression is robustly safe, provided its type satisfies conditions for being *public*. A function type is public, roughly, when there are no non-trivial preconditions (refinements) on its arguments. We cannot expect an untyped context to respect such preconditions. In particular, the function type $(\text{linkclos}, \text{formclos})\text{webapp}$ is not public, because of the refinements on types for the constructors H_J of `linkclos` and `formclos`. The attacks (2) and (3) can be seen as an attacker failing to respect the integrity constraints expressed by these refinements.

6.2 Protecting Continuations with Authenticated Encryption

As outlined earlier, our solution is to encrypt all continuations returned by the server to the client, to protect confidentiality, and moreover to include a keyed hash within the encryption to protect against modifications.

The `make` and `check` functions, to construct and deconstruct the authenticated encryption of a continuation, are programmed as follows in F7. We use different keys for hashing and encrypting; in total there are four keys used for hashing links and forms, and for encrypting links and forms respectively. These keys are maintained on the server, and need never be sent to the client. (Since TINYLINKS is stateless, there is no need for replay protection.)

Authenticated encryption:

```
let mkKey: unit → 'a symkey = Crypto.mkEncKey // encryption key
let mkHKey: unit → 'a hkey = Crypto.mkHKey // hashing key

let make (ek: (hmac * 'a) symkey) (hk: 'a hkey) (s: 'a) : cipher =
  aesEncrypt ek (pickle (hmacsha1 hk (pickle s), s))

let check (ek: (hmac * 'a) symkey) (hk: 'a hkey) (m: cipher): 'a =
  let h,s = unpickle (aesDecrypt ek m) in
  let sv = hmacsha1 Verify hk (pickle s) h in s

let lkHKey: linkclos hkey = mkHKey()
let fhKey: formclos hkey = mkHKey()
let lSKey: (hmac * linkclos) symkey = mkKey()
let fSKey: (hmac * formclos) symkey = mkKey()
```

6.3 The Secure Semantics

The following table describes the changes to the definitions of the previous section to obtain our formal model $\llbracket E_{url} \rrbracket_s$ of the secure translation. The modifications are to the translation of the `xml` type, the translation of `href` and `form` values, and the top-level web server listener.

Modifications for the Secure Translation: $\llbracket E_{url} \rrbracket_s$

```
 $\llbracket \text{xml} \rrbracket_s = (\text{cipher}, \text{cipher})\text{xml}$ 

 $\llbracket \text{href}(E) \rrbracket_s =$ 
  let ciph = make lSKey lkHKey (C  $\llbracket \text{href}(E) \rrbracket_s$ ) in Href(ciph)
 $\llbracket \text{form}([\ell_1, \dots, \ell_n], E) \rrbracket_s =$ 
  let ciph = make fSKey fhKey (C  $\llbracket \text{form}([\ell_1, \dots, \ell_n], E) \rrbracket_s$ ) in
  FormElem(ciph,  $[\ell_1, \dots, \ell_n]$ )

let webserver (req : (cipher, cipher)req) → (cipher, cipher)xml =
  match req with
  | Get(None) →  $\mathcal{E} \llbracket E_{url} \rrbracket_s$ 
  | Get(Some(ciph)) →
    match (check lSKey HKey ciph) with
     $\prod_{J \in \mathcal{J} \wedge J = (\Gamma, F, \text{href}(E), T)}$  (|  $H_J(g) \rightarrow fH_J g$ )
  | Post(ciph, ls) →
    match (check fSKey fhKey ciph), ls with
     $\prod_{J \in \mathcal{J} \wedge J = (\Gamma, F, \text{form}([\ell_1, \dots, \ell_n], E), T)}$  (|  $H_J(g), [\ell_1; \dots; \ell_n] \rightarrow$ 
       $fH_J g [\ell_1; \dots; \ell_n]$ )
```

The encryption occurs during the translation of `href` and `form`, the values which construct continuations that may flow to the potential opponent. The code to dispatch on incoming requests in `webserver` is the same as before, except that the encrypted continuations need to be decrypted.

The function type $(\text{cipher}, \text{cipher})\text{webapp}$ is public, because there are no refinement types in its argument type. (None of the continuations in an argument $(\text{cipher}, \text{cipher})\text{req}$ are acted on until they have been dynamically validated by the cryptographic operations used by the `check` function. In F7, the type `cipher` of byte arrays acts somewhat like a dynamic type.) Hence, our main theorem is a corollary, given the robust-safety-by-typing theorem of F7, of Lemma 2.

Lemma 2. *Suppose that $\emptyset; \emptyset \vdash E_{url} \overset{E}{\sim} \langle _ : \text{xml} \rangle \{ \}$. Then $\llbracket E_{url} \rrbracket_s$ is a closed expression of F7 of type: $\llbracket E_{url} \rrbracket_s : (\text{cipher}, \text{cipher})\text{webapp}$.*

Theorem 2. *If E_{url} is provably safe at the source level, then the corresponding `webserver` $\llbracket E_{url} \rrbracket_s$ is provably robustly safe.*

7. Implementations

Our implementations consist of a modification to the Edinburgh LINKS system to support our secure compilation strategy, a type checker corresponding to the system presented in Section 4, and a compiler implementing the straightforward and secure translation rules from Section 5 and Section 6.

7.1 Modifications to the Edinburgh LINKS System

The Edinburgh LINKS system, is written in Objective Caml (OCaml) and consists of an interpreter that is called from a web server as a CGI program to process GET or POST requests. For our modifications we used the latests public release of the system which is version 0.4. To implement our solution based on cryptography, we used the Cryptokit library for OCaml which provides a variety of cryptographic primitives to implement security-sensitive applications.

The LINKS system defines clear interfaces between the interpreting loop and the marshaling and unmarshaling of requests and results, which are all collected in the `Result` module. Our modification amounts to extending the functions responsible for marshaling and unmarshaling continuations, expressions and environments, by adding a call to our custom authenticated encryption and decryption functions between the pickling, and the base64 encoding/decoding stages. Since our implementation requires only local modifications in a single module, we anticipate these changes to be applicable to later versions of the system. The same modification is applicable to the SE LINKS system. We validated our approach by performing our original attacks to the modified system and observed that they are no longer realisable.

7.2 A Certified (But Symbolic) Implementation of LINKS

There are always discrepancies of some sort between formal calculi like TINYLINKS and actual programming languages; we have implemented our results so as to validate them experimentally as well as theoretically. Our source level analysis is implemented in a certifying compiler, that, given a program with a security policy expressed by source level type annotations, checks that the program satisfies the particular policy, and not only produces executable F# code using the secure translation, but also produces machine checkable evidence that the translated code respects the policy (F7 interface).

The TINYLINKS implementation consists of 4500 lines of F# code divided between the parser, the type checker and the compiler. While we could have used the Edinburgh LINKS code as a basis, we chose not to do so for two reasons: (1) we want to understand better the security related details and abstract away from the complexity of the language implementation, (2) we intend to study the effect of language features to security in isolation, and progressively combine them with other novel language features,

Our compiler accepts unmodified LINKS syntax, so that a program (with ordinary type annotations for top-level functions) that is accepted by LINKS is also accepted by our compiler. The certified compiler works in two phases; the compilation phase and the verification phase. The compilation phase takes a TINYLINKS program that contains type-and-effect annotations, `event`, and `assert`, expressing the required security property, type checks it and produces an F# program along with an F7 interface as output. For the verification stage, we employ a type checker with refinement types (F7), that is part of the trusted computing base, to automatically check the certificate. Having executable formal semantics for LINKS gives us the opportunity for a fully interoperable but certified web server using the resulting F# program as a concrete implementation in a web server. Currently our translation cannot communicate with a browser but this is left as future work.

8. Related Work

λ -calculus with locations. HOP (Serrano et al. 2006) is a language similar to LINKS that uses a stratified approach in writing interactive applications across the web. It defines two strata: the computational one that carries out the computations defined by the program's logic and the graphical one that executes the computations of the graphical user interface. Like in LINKS, the compiler transforms the expressions that appear inside forms or links into continuation passing style and closure converts them. These continuations can float freely between the client and the server.

Neubauer and Thiemann (2005) propose a programming framework consisting of a calculus, a static analysis that performs an assignment of code to locations, and a range of program transformations that automatically insert communication primitives, perform resource pooling and finally split the program into separate processes to run at separate locations.

The implementation of the PLT Scheme Web Server (Krishnamurthi et al. 2007), provides web-specific core primitives for writing applications that are able to interact with a database. The main contribution is the advanced control flow mechanisms that are exposed through the core primitives. Actual continuations are stored on the server, and only continuation identifiers are shipped to the client, with their corresponding threads being suspended, waiting for input on a channel.

ML5 (Murphy VII et al. 2008) extends the two tier approach of HOP to a fully distributed multi-tier language focusing more on mobility and develops a type system that ensures that mobile resources are used in a type safe manner. The implementation shares several of the ideas (CPS, closure conversion) that appear in both LINKS and HOP, but there is no explicit mention of protection against a rogue client.

Security for multi-tier languages. The closest work to ours is FABLE (Swamy et al. 2008), a language for defining and enforcing user-defined security policies. SELinks expands on original goal of LINKS and extends it by allowing the web server and the database to collaboratively enforce application specific security policies. While the type system ensures that a specific policy is respected by a program using the source level semantics, no claim is made about whether these semantics are securely implemented by the SE Links system; we have evidence that it is indeed susceptible to the low level attacks that our work directly addresses. (The SELinks extensions address questions of database security, and do not address the problem of low level attacks by untrustworthy HTTP clients; the attacks in Section 3 apply to both the original Edinburgh version and to SE Links.)

The work on Swift at Cornell (Chong et al. 2007) uses Java and GWT to write web applications expressing integrity and confidentiality properties as type annotations. The compiler statically analyses these annotations and decides where data and code will be placed to guarantee the security constraints. Location annotations are at the statement level and subexpressions of a compound expression are allowed to execute at different hosts. To ensure control flow integrity, the client and the server runtime systems maintain a stack of closures that communicate with each other with control transfer messages. A client is not allowed to execute a high integrity closure, unless it was pushed from the server and it happens to be at the top of the closure stack. The server checks that these conditions hold and can detect violations.

9. Conclusion

We have obtained practical and theoretical results demonstrating that it is possible to perform source-based security analysis in a multi-tier web programming language. In the context of TINYLINKS, we have formalised an example type-and-effect sys-

tem that can guarantee data and control flow integrity properties of programs. Our results show that through a secure, type-directed translation we can ensure that properties proved at the source level are preserved by the actual implementation. To further validate our approach, we have implemented both a type-and-effect checker and our secure translation producing executable semantics in F#, that can form part of a certified web server.

We have identified additional attacks that are not expressible in our limited calculus with no mutable state and no support for client-side functions; we aim to address these limitations in future work. Additionally, we intend to extend our compiler and the theoretical framework developed in this paper to handle secrecy properties.

Acknowledgements Several discussions about LINKS and security with Ezra Cooper, Nikhil Swamy, and Phil Wadler were stimulating and fun. We also enjoyed discussing web application security with Cédric Fournet, Shriram Krishnamurthi, and Benjamin Livshits. Karthik Bhargavan was a great help with mastering the arts of F7 programming. David Greaves commented on a draft.

References

- Martín Abadi. Protection in Programming-Language Translations. In *Secure Internet Programming*, pages 19–34, 1999.
- Andrew W. Appel. *Compiling with Continuations*. CUP, 1992.
- J. Bengtson, K. Bhargavan, C. Fourmet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR-TR-2008-118, Microsoft Research, 2008. A preliminary, abridged version appears in the proceedings of CSF’08.
- Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP ’07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.
- Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS. Springer-Verlag, 2006.
- ECMA International. Standard ECMA-262, ECMAScript Language Specification, 3rd edition. WWW electronic publication, December 1999. <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- Sameh El-Ansary, Donatien Grolaux, Peter Van Roy, and Mahmoud Rafea. Overcoming the multiplicity of languages and technologies for web-based development using a multi-paradigm approach. In Peter Van Roy, editor, *MOZ*, volume 3389 of *LNCS*, pages 113–124. Springer, 2004.
- C. Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL’06)*, pages 245–256, 2006.
- D. Gollmann. Authentication by correspondence. *IEEE Journal on Selected Areas in Communication*, 21(1):88–95, 2003.
- Google. Google Web Toolkit. WWW electronic publication, 2006. <http://code.google.com/webtoolkit/>.
- A. D. Gordon and A. S. A. Jeffrey. Typing correspondence assertions for communication protocols. *TCS*, 300:379–409, 2003.
- Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with High-Level Programming Languages. In *ESOP ’01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, 2001.
- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.
- Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT scheme Web server. *HOSC*, 20(4), 2007.
- Erik Meijer and Wolfram Schulte. Programming with Rectangles, Triangles, and Circles. In *XML Conference*, 2003.
- Microsoft. ASP.Net. WWW electronic publication, November 2006. <http://www.asp.net/>.
- Microsoft. Language Integrated Query (LINQ). WWW electronic publication, 2005. <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>.
- Tom Murphy VII, Karl Crary, and Robert Harper. Type-Safe Distributed Programming with ML5. In *Trustworthy Global Computing*, LNCS, pages 108–123, 2008.
- Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005.
- B. C. Pierce and D. N. Turner. Local type inference. In *ACM Symposium on Principles of Programming Languages (POPL’98)*, pages 252–265. ACM, 1998.
- Gordon D. Plotkin. Call-by-name, call-by-value, and the λ -calculus. *TCS*, 1:125–159, 1975.
- Christian Queinnee. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP ’00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000.
- A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.
- Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101. Springer, 2000.
- Manuel Serrano, Erick Gallezio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA ’06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.
- Sun Microsystems. JSR 245: JavaServerTM Pages 2.1. WWW electronic publication, November 2006. <http://java.sun.com/jsp/>.
- Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *IEEE Symposium on Security and Privacy*, pages 369–383, 2008.
- D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
- The PHP Group. PHP: Hypertext Preprocessor. WWW electronic publication, November 2006. <http://www.php.net/>.
- W3C. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. WWW electronic publication, July 2007. <http://www.w3.org/TR/2007/CR-CSS21-20070719/>.
- W3C. HTML 4.01 Specification. WWW electronic publication, December 1999. <http://www.w3.org/TR/html401/>.
- T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.
- H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL’99)*, pages 214–227. ACM, 1999.
- 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)*, 2006.