Static Enforcement of Web Application Integrity Through Strong Typing

William Robertson
wkr@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Giovanni Vigna
vigna@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Abstract

Security vulnerabilities continue to plague web applications, allowing attackers to access sensitive data and co-opt legitimate web sites as a hosting ground for malware. Accordingly, researchers have focused on various approaches to detecting and preventing common classes of security vulnerabilities in web applications, including anomaly-based detection mechanisms, static and dynamic analyses of server-side web application code, and client-side security policy enforcement.

This paper presents a different approach to web application security. In this work, we present a web application framework that leverages existing work on strong type systems to statically enforce a separation between the *structure* and *content* of both web documents and database queries generated by a web application, and show how this approach can automatically prevent the introduction of both server-side cross-site scripting and SQL injection vulnerabilities. We present an evaluation of the framework, and demonstrate both the coverage and correctness of our sanitization functions. Finally, experimental results suggest that web applications developed using this framework perform competitively with applications developed using traditional frameworks.

Keywords: Web applications, strongly typed languages, functional languages, cross-site scripting, SQL injection.

1 Introduction

In the last decade, web applications have become an extremely popular means of providing services to large numbers of users. Web applications are relatively easy to develop, the potential audience of a web application is a significant proportion of the planet's population [36, 10], and development frameworks have evolved to the point that web applications are approaching traditional thick-client applications in functionality and usability.

Unfortunately, web applications have also been found to contain many security vulnerabilities [40]. Web applications are also widely accessible and often serve as an interface to large amounts of sensitive data stored in back-end databases. Due to these factors, web applications have attracted much attention from cyber-criminals. Attackers commonly exploit web application vulnerabilities to steal confidential information [41] or to host malware in order to build botnets, both of which can be sold to the highest bidder in the underground economy [46].

By far, the most prevalent security vulnerabilities present in web applications are cross-site scripting (XSS) and SQL injection vulnerabilities [42]. Cross-site scripting vulnerabilities are introduced when an attacker is able to inject malicious scripts into web content to be served to other clients. These scripts then execute with the privileges of the web application delivering the content, and can be used to steal authentication credentials or to install malware, among other nefarious objectives. SQL injections occur when malicious input to a web application is allowed to modify the structure of queries issued to a back-end database. If successful, an attacker can typically bypass authentication procedures, elevate privileges, or steal confidential information.

Accordingly, much research has focused on detecting and preventing security vulnerabilities in web applications. One approach is to deploy web application firewalls (WAFs), usually incorporating some combination of misuse and anomaly detection techniques, in order to protect web applications from attack [6, 14, 8, 29, 45]. Anomaly detection approaches are attractive due to their black-box approach; they typically require no *a priori* knowledge of the structure or implementation of a web application in order to provide effective detection.

Another significant focus of research has been on applying various static and dynamic analyses to the source code of web applications in order to identify and mitigate security vulnerabilities [21, 33, 25, 2, 7, 50]. These approaches have the advantage that developers can continue to create web applications using traditional languages and frameworks, and periodically apply a vulnerability analysis tool to provide a level of assurance that no security-relevant flaws are present. Analyzing web applications is a complex task, however, as is the interpretation of the results of such security tools. Additionally, several approaches require developers to specify security policies to be enforced in a specialized language.

A more recent line of research has focused on providing client-side protection by enforcing security policies within the web browser [43, 22, 13]. These approaches show promise in detecting and preventing client-side attacks against newer web applications that aggregate content from multiple third parties, but the specification of policies to enforce is generally left to the developer.

In this paper, we propose a different approach to web application security. We observe that cross-site scripting and SQL injection vulnerabilities can be viewed as a failure on the part of the web application to enforce a separation of the *structure* and the *content* of documents and database queries, respectively, and that this is a result of treating documents and queries as untyped sequences of bytes. Therefore, instead of protecting or analyzing existing web applications, we describe a framework that strongly types both documents and database queries. The framework is then responsible for automatically enforcing a separation between structure and content, as opposed to the *ad hoc* sanitization checks that developers currently must implement. Consequently, the integrity of documents and queries generated by web applications developed using our framework are automatically protected, and thus, *by construction*, such web applications are not vulnerable to server-side cross-site scripting and SQL injection attacks.

To illustrate the problem at hand, consider that HTML or XHTML documents to be presented to a client are typically constructed by concatenating strings. Without additional type information, a web application framework has no means of determining that the following operations could lead to the introduction of a cross-site scripting vulnerability:

```
String result = "<div>" + userInput + "</div>";
```

The key intuition behind our work is that because both documents and database queries are strongly typed in our framework, the framework can distinguish between the structure (`<div>` and `</div>`) and the content (`userInput`) of these critical objects, and enforce their integrity automatically.

In this work, we leverage the advanced type system of Haskell, since it offers a natural means of expressing the typing rules we wish to impose. In principle, however, a similar framework could be implemented in any language with a strong type system that allows for some form of multiple inheritance (e.g., Java or C#).

In summary, the main contributions of this paper are the following:

- We identify the lack of typing of web documents and database queries as the underlying cause of cross-site scripting and SQL injection vulnerabilities.
- We present the design of a web application development framework that automatically prevents the introduction of cross-site scripting and SQL injection vulnerabilities by strongly typing both web documents and database queries.
- We evaluate our prototype web application framework, demonstrate the coverage and correctness of its sanitization functions, and show that applications under our framework perform competitively with those using existing frameworks.

The remainder of this paper is structured as follows. Section 2 presents the design of a strongly typed web application framework. The specification of documents under the framework and how their integrity is enforced is discussed in Section 3, and similarly for SQL queries in Section 4. Section 5 evaluates the design of the framework, and demonstrates that web applications developed under this framework are free from certain classes of vulnerabilities. Related work is discussed in Section 6. Finally, Section 7 concludes and presents avenues for further research.

2 Framework design

At a high level, the web application framework is composed of several familiar components. A web server component processes HTTP requests from web clients and forwards these requests in an intermediate form to the application server based on one of several configuration parameters (e.g., URL path prefix). These requests are directed to one of the web applications hosted by the application server. The web application examines any parameters to the request, performs some processing during which queries to a back-end database may be executed, and generates a document. Note that in the following, the terms “document” or “web document” shall generically refer to any text formatted according to the HTML or XHTML standards. This document is then returned down the component stack to the web server, which sends the document as part of an HTTP response to the web client that originated the request. A graphical depiction of this architecture is given in Figure 1.

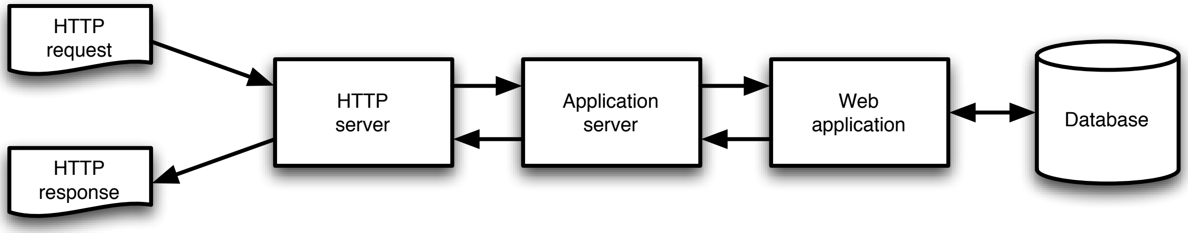


Figure 1: Architectural overview of the web application framework.

Web applications developed for our framework are structured as a set of functions with access to a combination of configuration data and application state. More precisely, web applications execute inside the `App` monad. Monads are a category theoretic construction that have found wide application in Haskell to sequence actions or isolate code that can produce side effects.[1] For the purposes of our framework, we use the `App` monad to thread implicit state through the functions comprising a web application, and to provide a controlled interface to potentially dangerous functions. In particular, the `App` monad itself is structured as a stack of monad transformers that provide a functional interface to a read-only configuration type `AppConfig`, a read-write application state type `AppState`, and filtered access to the `IO` monad. The definitions for `AppConfig` and `AppState` are given in Figures 2 and 3.

```
data AppConfig = AppConfig {
     appCfgPort :: Int,
     appCfgPrefix :: String,
     appCfgRoutes :: RouteMap,
     appCfgFileRoot :: FilePath,
     appCfgDBConn :: Connection,
     appCfgDBStmts :: StmtMap
}
```

Figure 2: Definition for the `AppConfig` type.

The `AppConfig` type holds static information relating to the configuration of the application, including the port on which to listen for HTTP requests and the root directory of static files to serve from the filesystem. Of particular interest, however, are the `RouteMap` and `StmtMap` fields. The `RouteMap` type describes how URL paths are mapped to values of type `DocumentGen`, which are simply functions that generate documents within the `App` monad. In addition, the `RouteMap` type contains a default `DocumentGen` type that specifies an error page. Given an incoming HTTP request destined for a particular web application, the application server uses that application's `RouteMap` type to determine the proper function to call in order to generate the document to be returned to the client.[2] Finally, the `StmtMap` type associates unique database query identifiers to prepared statements that can be executed by a document generator.

```
data AppState = AppState {
     appStClient :: Maybe SockAddr,
     appStUrl :: Maybe Url
}
```

Figure 3: Definition for the `AppState` type.

The `AppState` type contains mutable state that is specific to each request for a document. In particular, one field records information indicating the source of the request. Additionally, another field records the URL that was requested, including any parameters that were specified by the client. More complex state types that hold additional information (e.g., cached database queries or documents) are possible, however.

3 Document structure

In this section, we introduce the means by which documents are specified under the framework. Then, we discuss how these specifications allow the framework to automatically contain the potentially harmful effects of dynamic data.

3.1 Document specification

Once an appropriate route from the `RouteMap` structure has been selected by the application server, the associated document generator function is executed within the context of the `App` monad (i.e., with access to the

[1] For further information on monads, please refer to [37, 48].

[2] This construction is similar to the "routes" packages present in popular web development frameworks such as Rails [18] and Pylons [4].

```
data Document Document {
     docType :: DocumentType,
     docHead :: DocumentHead,
     docBody :: DocumentBody
}

data DocumentType = DOC_TYPE_HTML_4_01_STRICT
                  | DOC_TYPE_HTML_4_01_TRANS
                  | ...
                  | DOC_TYPE_XHTML_1_1

data DocumentHead = DocumentHead {
     docTitle :: String,
     docLinks :: [Node],
     docScripts :: [Node],
     docBaseUrl :: Maybe Url,
     docBaseTarget :: Maybe Target,
     docProfile :: [Url]
}

data DocumentBody = DocumentBody {
     docBodyNode :: Node
}
```

Figure 4: Definition for the Document type.

```
data Node = TextNode {
     nodeText :: String
}         | AnchorNode {
     anchorAttrs :: NodeAttrs,
     anchorHref :: Maybe Url,
     anchorRel :: Maybe Relationship,
     anchorRev :: Maybe Relationship,
     anchorTarget :: Maybe Target,
     anchorType :: Maybe MimeType,
     anchorCharset :: Maybe CharSet,
     anchorLang :: Maybe Language,
     anchorName :: Maybe AttrValue,
     anchorShape :: Maybe Shape,
     anchorCoords :: Maybe Coordinates,
     anchorNodes :: [Node]
}         | DivNode {
     divAttrs :: NodeAttrs,
     divNodes :: [Node]
} ...
```

Figure 5: Sample Node definitions.

configuration and current state of the application). The document generator function processes the request from the application server and returns a variable of type Document. The definition of the Document type and its constituent types are shown in Figure 4.

As is evident, documents in our framework are not represented as an unstructured stream of bytes. Rather, the structure of the Document type closely mirrors that of parsed HTML or XHTML documents. The DocumentType field indicates the document's type, such as "HTML 4.01 Transitional" or "XHTML 1.1". The DocumentHead type contains information such as the title and client-side code to execute. The DocumentBody type contains a single field that represents the root of a tree of nodes that represent the body of the document.

Each node in this tree is an instantiation of the Node type. Each Node instantiation maps to a distinct (X)HTML element, and records the set of possible properties of that element. For instance, the TextNode data constructor creates a Node that holds a text string to be displayed as part of a document. The AnchorNode data constructor, on the other hand, creates a Node that holds information such as the href attribute, rel attribute, and a list of child nodes corresponding to the text or other elements that comprise the "body" of the link. A partial definition of the Node type is presented in Figure 5.

With this construction, the entire document produced by a web application in our framework is strongly typed. Instead of generating a document as a byte stream, document structure is explicitly encoded as a tree of nodes. Furthermore, each element and element attribute has an associated type that constrains, to one degree or another, the range of possible values that can be represented. For instance, the MimeType, CharSet, and Language types are examples of enumerations that strictly limit the set of possible values the attribute can take to legal values. Standard (X)HTML element attributes (e.g., id, class, style) are represented with the NodeAttr type. Optional attributes are represented using either the Maybe type,[3] or as an empty list if multiple elements are allowed.

Note that it is possible for a Document to represent an (X)HTML document that is not necessarily consistent with the respective W3C grammars that specify the set of well-formed documents. One example is that any Node instantiation may appear as the child of any other Node that can hold children, which violates the official grammars in several instances. Strict conformance with the W3C standards is not, however, our goal.[4]

Instead, the typing scheme presented here allows our framework to specify a separation between the *structure* and the *content* of the documents a web application generates. More precisely, the dynamic data that enters a web application as part of an HTTP request (e.g., as a GET or POST parameter) can indirectly influence the structure of a document. For instance, a search request to a web application may result in a variable number of

[3] Maybe allows for the absence of a value, as Haskell does not possess nullable types. For example, the type Maybe a can be either Just "..." or Nothing.

[4] Indeed, standards-conforming documents have been shown to be difficult to represent in a functional language [12].

```
class Render a where
      render :: a -> String

instance Render AttrValue where
         render = quoteAttr

quoteAttr :: AttrValue -> String
quoteAttr a = foldl’ step [] (attrValue a)

step acc c | c == ’<’ = acc ++ "&lt;"
           | c == ’>’ = acc ++ "&gt;"
           | c == ’&’ = acc ++ "&amp;"
           | c == ’"’ = acc ++ "&quot;"
           | otherwise = acc ++ [c]
```

Figure 6: `Render` typeclass definition and (simplified) instance example. Here, `quoteAttr` performs a left fold over attribute values using `foldl’`, which applies the `step` function to each character of the string and accumulates the result. The definition of `step` specifies a number of *guards*, where `| c == ’<’` is a condition that must be satisfied for the statement `acc ++ "&lt;"` to execute. This statement simply appends the string `"&lt;"` to `acc`, the accumulator, in order to build a new, sanitized string. If no guard condition is satisfied, the character is appended without conversion.

table rows in the generated document depending on the number of results returned from a database query. Due to our framework, however, client-supplied data cannot *directly* modify the structure of the document in such a way that a code injection can occur.

3.2 Enforcing document integrity

Once a `Document` has been constructed by the web application in response to a client request, it is returned to the application server. The application server is responsible for converting this data structure into a format the client can understand – that is, it must *render* the document into a stream of bytes representing an (X)HTML document. Consequently, the set of types that can comprise a `Document` are instances of the `Render` typeclass, shown in Figure 6.[5]

The `Render` type class specifies that any instance of the class must implement the `render` function. From the type signature, the semantics of the function are clear: `render` converts an instance type into a string representation suitable for presentation to a client. Crucially, for our purposes, the `Render` type class is also responsible for enforcing the integrity of a document’s structure.

As an example, Figure 6 presents a simplified `render` definition for the `AttrValue` type that is used to indicate element attribute strings that may assume (almost) arbitrary values. In order to preserve the integrity of the document, an attribute value must not contain certain characters that would allow an attacker to inject malicious code into the document. Consider, for instance, the following element:

```
<input type="hidden" name="h1" value="..."/>
```

Now, suppose an attacker submitted the following string as part of a request such that it was reflected to another client as the value of the hidden input field:

```
"/>
<script src="http://example.com/malware.js">
</script>
<span id="
```

The result would be the following:

```
<input type="hidden" name="h1" value=""/>
<script src="http://example.com/malware.js">
</script>
<span id=""/>
```

To prevent such an injection from occurring, the `render` function for the `AttrValue` class applies a sanitization function on the string wrapped by `AttrValue`. Any occurrence of an unsafe character is replaced by an equivalent HTML entity encoding that can safely appear as part of an attribute value.[6] Similar `render` functions are defined for the set of types that can comprise a `Document`.

Therefore, to prepare a `Document` as part of an HTTP response to a client, the application server applies the `render` function to the document, which recursively converts the data structure into an (X)HTML document. As part of this process, the content of the document is sanitized by type-specific `render` functions, ensuring that client-supplied input to the web application cannot modify the document structure in such a way as to result in a client-side code injection.

4 SQL query structure

Similarly to the case of documents, SQL queries are given structure in our framework through the application of strong typing rules that control how the structure of the query can be combined with dynamic data. In this section, we examine the structure of SQL queries and discuss two mechanisms by which SQL query integrity is enforced under the framework.

[5] Haskell typeclasses are roughly similar to Java interfaces, in that they specify a function interface that all instances (in Java, implementors) must provide.

[6] In the real implementation, the sanitization function is somewhat more complex, as there are multiple encodings by which an unsafe character can be injected. The example function given here is simplified for the purposes of presentation.

```
INSERT INTO users(login, passwd)
VALUES(?, ?)

SELECT * FROM users
WHERE login='admin' AND passwd='test'

UPDATE users SET passwd='$passwd'
WHERE login='$login'
```

Figure 7: Examples of SQL queries.

4.1 Query specification

SQL queries, as shown in Figure 7, are composed of clauses, predicates, and expressions. For instance, a clause might be `SELECT *` or `UPDATE users`. An example of a predicate is `login='admin'`, where `'admin'` is an expression. Clauses, predicates, and expressions are themselves composed of static tokens, such as keywords (`SELECT`) and operators (=), and dynamic tokens, such as table identifiers (`users`) or data values (`'admin'`).

Typically, the structure of a SQL query is fixed.[7] Specifically, a query will have a static keyword denoting the operation to perform, will reference a static set of tables and fields, and specify a fixed set of predicates. Generally, the only components of a query that change from one execution to the next are data values, and, even then, their number and placement remain fixed.

SQL injection attacks rely upon the ability of the attacker to modify the structure of a query in order to perform a malicious action. When SQL queries are constructed using string operations without sufficient sanitization applied to user input, such attacks become trivial. For instance, consider the `UPDATE` query shown in Figure 7. If an attacker were to supply the value “`quux' OR login='admin`” for the `$login` variable, the following query would result:

```
UPDATE users SET passwd='foo'
WHERE login='quux' OR login='admin'
```

Because the attacker was able to inject single quotes, which serve as delimiters for data values, the structure of the query was changed, resulting in a privilege escalation attack.

4.2 Integrity enforcement with static query structure

```
SELECT * FROM users WHERE login=? AND passwd=?
   UPDATE users SET passwd=? WHERE login=?
```

Figure 8: Examples of prepared statements, where “?” characters serve as placeholders for data substitution.

In contrast to the case of document integrity enforcement, a well-known solution exists for specifying SQL query structure: prepared statements. Prepared statements are a form of database query consisting of a parameterized query template containing placeholders where dynamic data should be substituted. An example is shown in Figure 8, where the placeholders are signified by the “?” character.

A prepared statement is typically parsed and constructed prior to execution, and stored until needed. When an actual query is to be issued, variables that may contain client-supplied data are *bound* to the statement. Since the query has already been parsed and the placeholders specified, the structure of the query cannot be modified by the traditional means of providing malicious input designed be interpreted as part of the query. In the case of the injection attack described previously, the result would be as the following (note that the injected single quotes have been escaped):

```
UPDATE users SET passwd='foo'
WHERE login='quux'' OR login=''admin'
```

From our perspective, the query has been typed as a composition of static and dynamic elements; it is exactly this distinction between structure and content that we wish to enforce. Haskell’s database library (HDBC), as do most other languages, supports the use of prepared statements. Therefore, the framework exports functions that allow a web application to associate prepared statements with a unique identifier in the `AppConfig` type. During request processing, a document generator can then retrieve a prepared statement using the identifier, bind values to it, and execute queries that are not vulnerable to injection attacks.

One detail remains, however. The HDBC library also provides functions allowing traditional *ad hoc* queries that are assembled as concatenations of strings to be executed. Without any other modification to the framework, a web application developer would be free to directly call these functions and bypass the protections afforded by the framework. Therefore, an additional component is required to encapsulate the HDBC interface and prevent execution of these unsafe functions. This component takes the form of a monad transformer `AppIO`, which simply wraps the `IO` monad and exposes only those functions that are considered safe to execute. The structure of this stack is shown in Figure 9. In this environment, within which all web applications using the framework operate, unsafe database execution functions are inaccessible, since they will fail to type-check. Thus, assuming

[7] This is not always the case, but the case of dynamic query structure will be considered later in this section.

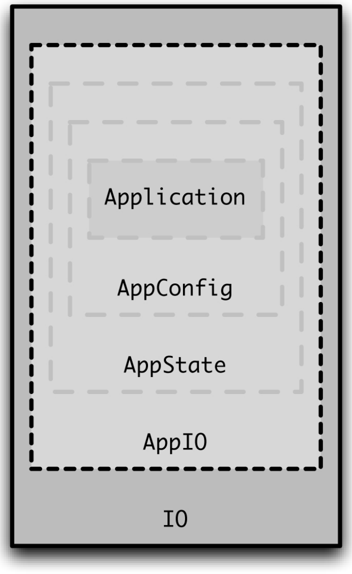


Figure 9: Graphical representation of the monad stack within which framework applications execute. The `AppIO` monad encapsulates applications, preventing them from calling unsafe functions within the `IO` monad.

the correctness of the HDBC prepared statement interface, web applications developed using the framework are not vulnerable to SQL injection.

4.3 Integrity enforcement with dynamic query structure

Though most SQL queries possess a fixed structure, there does exist a small class of SQL queries that exhibit dynamic structure. For instance, many SQL database implementations provide a set membership operator, where queries of the form

```
SELECT * FROM users WHERE
login IN ('admin', 'developer', 'tester')
```

can be expressed. In this case, the size of the set of data values can often change at runtime. Another example is the case where the structure of queries is determined by the user, for instance through a custom search form where many different combinations of predicates can be dynamically expressed. Unfortunately, since these queries cannot be represented using prepared statements, they cannot be protected using the monadic encapsulation technique described previously.

Therefore, a second database interface is exposed by the framework to the application developers. Instead of relying upon prepared statements, this interface allows developers to dynamically construct queries as a tree of algebraic data types as in the case of web documents. Figure 10 shows an example of the type representing a `SELECT` query.

```
data Select = Select {
     sFields :: [Expr],
     sTables :: [Expr],
     sCons :: Maybe Expr,
     sGrpFields :: [Expr],
     sGrpCons :: Maybe Expr,
     sOrdFields :: [Expr],
     sLimit :: Maybe Int,
     sOffset :: Maybe Int,
     sDistinct :: Bool
}

data Expr = EXPR_TABLE Table
          | EXPR_FIELD Field
          | EXPR_DATA String
          | EXPR_NOT Expr
          | EXPR_OR Expr Expr
          | EXPR_AND Expr Expr
          | ...

data Table = Table {
     tName :: String,
     tAlias :: Maybe String
}

data Field = Field {
     fName :: String,
     fAlias :: Maybe String
}
```

Figure 10: Definition for the `Select` type.

To populate instances of these types, the interface provides a set of combinators, or higher-order functions, that can be chained together. These combinators, which assume names similar to SQL keywords, implement an embedded domain-specific language (DSL) that allows application developers to naturally specify dynamic queries within the framework. For instance, a query could be constructed using the following sequence of function invocations:

```
qSelect [qField "*"] >>=
qFrom [qTable "users"] >>=
qWhere (((qField "login") == (qData "admin")) &&
((qField "passwd") == (qData "test")))
```

Similar to the case of the `Document` type, queries constructed in this manner are transformed into raw SQL statements solely by the framework.[8] Therefore, the

[8] Note that, as in the case of web documents, we do not attempt to enforce the generation of correct SQL, but rather focus on preventing attacks by preserving query structures specified by the developer.

Context	Semantics
Document nodes	Conversion to static string
Document node attributes	Encoding of HTML entities
Document text	Encoding of HTML entities
URL components	Encoding of HTML entities, percent encoding
SQL static value	Removal of spaces, comments, quotes
SQL data value	Escaping of quotes

Table 1: Example contexts for which specific sanitization functions are applied, and the semantics of those sanitization functions under various encodings.

types that represent queries also implement the Render typeclass. Consequently, sanitization functions must be applied to each of the fields comprising the query types, such that the intended structure of the query cannot be modified. This can be accomplished by enforcing the conditions that no data value may contain an unescaped single quote, and that all remaining query components may not contain spaces, single quotes, or character signifying the beginning of a comment. Assuming that these sanitization functions are correct, this construction renders applications developed under the framework invulnerable to SQL injection attacks while allowing for more powerful query specifications.

5 Evaluation

To demonstrate that web applications developed using our framework are secure by construction from server-side XSS and SQL injection vulnerabilities, we conducted an evaluation of the system. First, we demonstrate that all dynamic content contained in a Document must be sanitized by an application of the render function, and that a similar condition holds for dynamically-generated SQL queries. Then, we provide evidence that the sanitization functions themselves are correct – that is, they successfully strip or encode unsafe characters. We also verify that the prepared statement library prevents injections, as expected. Finally, to demonstrate the viability of the framework, an experiment to evaluate the performance of a web application developed using the framework is conducted.

5.1 Sanitization function coverage

The goal of the first experiment was to justify the claim that all dynamic content contained in a Document or query type must be sanitized prior to presentation to the client that originated the request. To accomplish this, a static control flow analysis of the framework was performed. Figure 11 presents a control flow graph of the application server in a simplified form, where function calls are sequenced from left to right. Of particular interest is the renderDoc function, which retrieves the appropriate document generator given a URL path, executes it in the call to route, sanitizes it by applying render, and creates an HTTP response by calling make200. The sanitized document is then returned to procRequest, which writes it to the client. Therefore, the entire process of converting the document to a byte stream for presentation to the client is solely due to the recursive render application. Similarly, because the only interface exposed to applications to execute SQL queries are execStmt and execPrepStmt from within the App monad, queries issued by applications under the framework must be sanitized either by the framework or the HDBC prepared statement functions.

Figure 12 displays a subset of the full control flow graph depicting an instance of the render function for the AnchorNode Node instantiation. For clarity of presentation, multiple calls to render and maybeRenderAttr have been collapsed into single nodes. Recall from Figure 5 that the definition of AnchorNode does not contain any bare strings; instead, each field of the type is either itself a composite type, or an enumeration for which a custom render function is defined. Since no other string conversion function is applied in this subgraph, we conclude that all data contained in an AnchorNode variable must be filtered through a sanitization function.

The analysis of this single case generalizes to the set of all types that can comprise a Document or query type. In total, 163 distinct sanitization function definitions were checked to sanitize the contexts shown in Table 1. For each function, our analysis found that no irreducible type was concatenated to the document byte stream without first being sanitized.

5.2 Sanitization function correctness

The goal of this experiment is to determine whether the sanitization functions employed by the framework are correct (i.e., whether all known sequences of dangerous characters are stripped or encoded). To establish

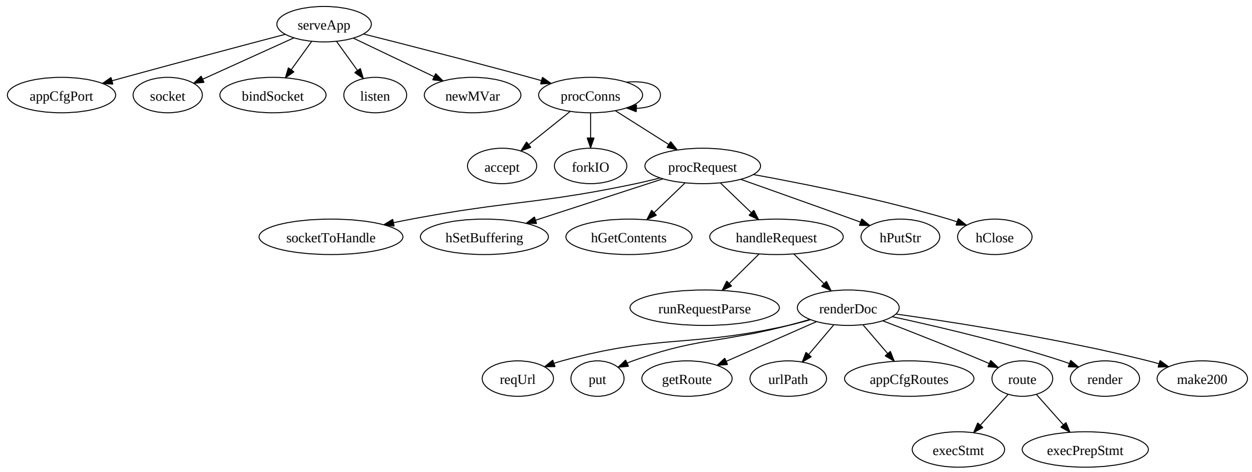


Figure 11: Simplified control flow graph for application server.

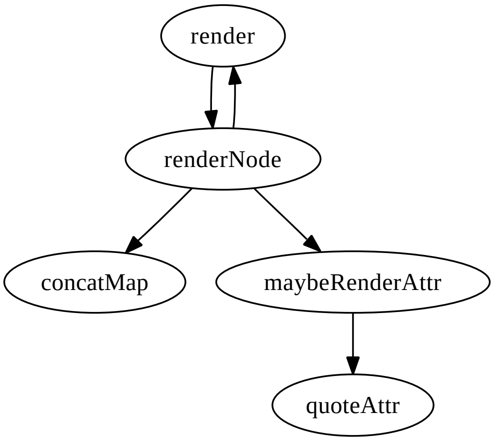


Figure 12: Example control flow graph for `Render Node` instance.

this, we applied a dynamic test-driven approach using the QuickCheck property testing library [9]. QuickCheck allows a developer to specify invariants in an embedded language that should hold for a given set of functions. The library then automatically generates a set of random test cases, and checks that the invariants hold for each test. In our case, we selected invariants based upon known examples of XSS [44] and SQL injection attacks [15]. In addition, we introduced modifications of the invariants that account for different popular document encodings, since these encodings directly affect how browser parsers interpret the sequences of bytes that comprise a document.

Since the coverage of the sanitization functions has been established by the control flow analysis, we focused our invariant testing on the low-level functions responsible for processing string data. In particular, we specified invariants for 7 functions that are responsible for sanitizing (X)HTML content, element attributes, and various URL components.[9] An example invariant specification is shown in Figure 13.

```
propAttrValueSafe :: AttrValue -> Bool
propAttrValueSafe input =
    (not $ elem ’<’ output) &&
    (not $ elem ’>’ output) &&
    (not $ elem ’&’ $ stripEntities output) &&
    (not $ elem ’"’ output) where
    output = render input
```

Figure 13: Simplified example sanitization function invariant specification. Here, `propAttrValueSafe` is a conjunction of predicates, where `not $ elem c output` specifies that the character `c` should not be an element of the output of `render` in this context. Since “`&`” is used to indicate the beginning of an HTML entity (e.g., &amp;), the `stripEntities` function ensures that ampersands may only appear in this form.

For each of the sanitization functions, we first tested the correctness of the invariants by checking that they were violated over a set of 100 strings corresponding to real-world cross-site scripting, command injection, and other code injection attacks. Then, for each sanitization function, we generated 1,000,000 test cases of random strings using the QuickCheck library. In all cases, the invariants were satisfied.

In addition to performing invariant testing on the set of document sanitization functions, we also applied a similar testing process to the sanitization of query types de-

[9] The 163 functions noted above eventually apply one of these 7 context-specific sanitization functions for web documents.

scribed in Section 4.3. Finally, we applied manual invariant testing on the HDBC prepared statement interface. In all cases, the invariants on the integrity of the queries and the database itself held.

5.3 Framework performance

In this experiment, we compared the performance of a web application developed using our framework to similar applications implemented using other frameworks. In particular, we developed a small e-commerce site with a product display page, cart display page, and checkout page under our framework, using the Pylons framework 0.9.7 [4], and as a Java servlet using Tomcat 6.0.18.[10] Each application was backed by a SQLite database containing product information. The application servers were hosted on a server running Ubuntu Server 8.10 with dual Intel Core 2 Duo CPUs, 2 GB of RAM, and 1000BaseT Ethernet network interfaces. The `httperf` [20] web server benchmarking tool was deployed on a similar server to generate load for each application.

Figure 14 presents averaged latency and throughput plots for 8 benchmarking runs for each framework tested. In each run, the number of concurrent clients issuing requests was varied, and the average response latency in milliseconds and the aggregate throughput in kilobytes was recorded. In this experiment, our framework performed competitively compared to Pylons and Tomcat, performing somewhat better than Pylons in both latency and throughput scaling, and vice versa for Tomcat. In particular, the latency plot shows that our framework scales significantly better with the number of clients than the Pylons framework. Unfortunately, our framework exhibited approximately a factor of two increase in latency compared to the Tomcat application. Cost-center profiling revealed that this is mainly due to the overhead of list-based `String` operations in Haskell,[11] though this could be ameliorated by rewriting the framework to prefer the lower-overhead `ByteString` type. Therefore, it is not unreasonable to assume that web applications developed using our framework would exhibit acceptable performance behavior in the real world.

5.4 Discussion

The security properties enforced by this framework are effective at guaranteeing that applications are not vulnerable to server-side XSS and SQL injection. There are limitations to this protection that need to be highlighted, however, and we discuss these here.

First, web applications can, in some cases, be vulnerable to *client-side* XSS injections, or DOM-based XSS, where the web application can potentially not receive any portion of such an attack [28]. This can occur when a client-side script dynamically updates the DOM after the document has been rendered by the browser with data controlled by an attacker. In general, XSS attacks stemming from the misbehavior of client-side code within the browser are not addressed by the framework in its current form.

Recently, a new type of XSS attack against the content-sniffing algorithms employed by web browsers has been demonstrated [5]. In this attack, malicious non-HTML files that nevertheless contain HTML fragments and client-side code are uploaded to a vulnerable web application. When such a file is downloaded by a victim, the content-sniffing algorithm employed by the victim's browser can potentially interpret the file as HTML, executing the client-side code contained therein, resulting in an XSS attack. Consequently, our framework implements the set of file upload filters recommended by the authors of [5] to prevent content-sniffing XSS. Since, however, the documents are supplied by users and not generated by the framework itself, the framework cannot guarantee that it is immune to such attacks.

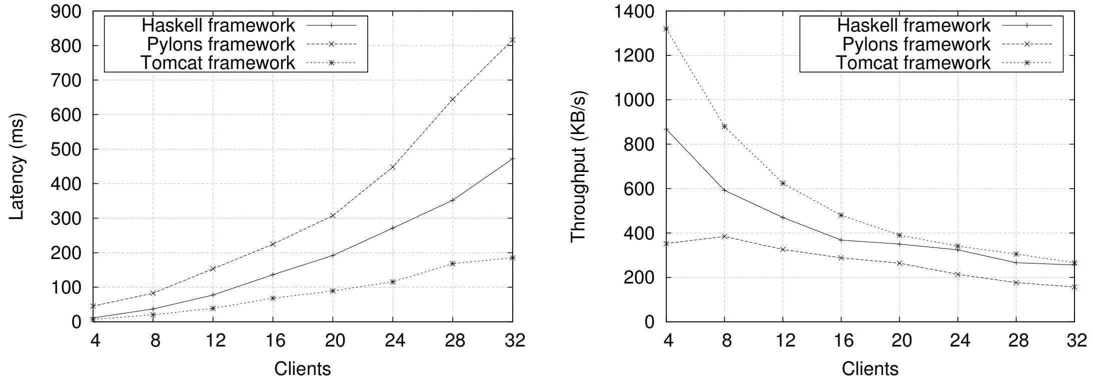
Finally, CSS stylesheets and JSON documents can also serve as vectors for XSS attacks. In principle, these documents could be specified within the framework using the same techniques applied to (X)HTML documents, along with context-specific sanitization functions. In the case of CSS stylesheets that are uploaded to a web application by users, additional sanitization functions could be applied to strip client-side code fragments. However, the framework in its current form does not address these vectors.

6 Related work

An extensive literature exists on the detection of web application vulnerabilities. One of the first tools to analyze server-side code for vulnerabilities was WebSSARI [21], which performs a taint propagation analysis of PHP in order to identify potential vulnerabilities, for which runtime guards are inserted. Nguyen-Tuong *et al.* proposed a precise taint-based approach to automatically hardening PHP scripts against security vulnerabilities in [39]. Livshits and Lam [33] applied a points-to static analysis to Java-based web applications to identify a number of security vulnerabilities in both open-source programs and the Java library itself. Jovanovic *et al.* presented Pixy, a tool that performs flow-sensitive, interprocedural, and context-sensitive data flow analysis to detect security

[10] Pylons is a Python-based framework that is similar in design to Ruby on Rails, and is used to implement a variety of well-known web applications (e.g., Reddit (`http://reddit.com/`)).

[11] Strings are represented as lists of characters in Haskell – that is, `type String = [Char]`.



(a)

(b)

Figure 14: Latency and throughput performance for the Haskell, Pylons, and Tomcat-based web applications.

vulnerabilities in PHP-based web applications [25]; Pixy was later enhanced with precise alias analysis to improve the accuracy of the technique [26]. A precise, sound, and fully automated technique for detecting modifications to the structure of SQL queries was described by Wassermann and Su in [49]. Balzarotti *et al.* observed that more complex vulnerabilities in web applications can manifest themselves as interactions between distinct modules comprising the application, and proposed MiMoSA to perform multi-module vulnerability analysis of PHP applications [2]. In [7], Chong *et al.* presented SIF, a framework for developing Java servlets that enforce legal information flows specified by a policy language. A syntactic technique of string masking is proposed by Johns *et al.* in [23] in order to prevent code injection attacks in web applications. Lam *et al.* described another information flow enforcement system using PQL, and additionally propose the use of a model checker to generate test cases for identified vulnerabilities [30]. In [1], Balzarotti *et al.* applied a combination of static and dynamic analysis to check the correctness of web application sanitization functions. Wassermann and Su applied a combination of taint-based information flow and string analysis to enforce effective sanitization policies against cross-site scripting in [50]. Nadji *et al.* propose a similar notion of document structure integrity in [38], using a combination of web application code randomization and runtime tracking of untrusted data on both the server and the browser. Finally, Google's ctemplate [17], a templating language for C++, and Django [11], a Python-based web application framework, include an Auto-Escape feature that allows for context-specific sanitization of web documents, while Microsoft's LINQ [35] is an approach for performing language-integrated data set queries in the .NET framework.

The approach described in this work differs from the above server-side techniques in several respects. First, an advantage of several of the above techniques is that they provide greater generality in their enforcement of security policies; in particular, SIF allows for the enforcement of complex information flows and uses some of the techniques presented in this work. Our framework, on the other hand, requires neither information flow policy specifications or additional static or dynamic analyses to protect against cross-site scripting or SQL injection vulnerabilities. String masking embodies a similar notion of a separation of code and data for web applications, but is implemented as a preprocessor for existing web applications and allows the possibility for both false positives and false negatives. Django and ctemplate are similar in spirit to this work in that they apply a similar context-sensitive sanitization of documents generated from template specifications. In both cases, however, this sanitization is optional and relies upon a separate document parser, whereas documents in our framework are specified in the language itself. ctemplate in particular has an advantage in that it supports limited sanitization of CSS and JSON documents, though this analysis is not currently based upon a robust parser. Finally, LINQ provides a language-based mechanism for dynamically constructing parameterized queries on arbitrary data sets, including SQL databases, and is therefore similar to the system proposed in this framework. Use of this interface is, however, optional and can be bypassed.

In addition to server-side vulnerability analyses, much work has focused on client-side protection against malicious code injection. The first system to implement client-side protection was due to Kirda *et al.* In [27], the authors presented Noxes, a client-side proxy that uses manual and automatically-generated rules to prevent cross-side scripting attacks. Vogt *et al.* proposed a combination of dynamic data tainting and static analysis to prevent cross-site scripting attacks from successfully executing within a web browser [47]. BrowserShield, due to Reis *et al.*, is a system to download signatures for known cross-site scripting exploits; JavaScript wrappers that implement signature detection for these attacks are then installed into the browser [43]. Livshits and Erlingsson described an approach to cross-site scripting and RSS attacks by modifying JavaScript frameworks such as Dojo, Prototype, and AJAX.NET in [32]. BEEP, presented by Jim *et al.* in [22], implements a coarse-grained approach to client-side policy enforcement by specifying both black- and white-lists of scripts. Erlingsson *et al.* proposed Mutation-Event Transforms, a technique for enforcing finer-grained client-side security policies by intercepting JavaScript calls that would result in potentially malicious modifications to the DOM [13].

In contrast to the client-side approaches discussed here, our framework does not require a separate analysis to determine whether cross-site scripting vulnerabilities exist in a web application. In the case of web applications that include client-side scripts from untrusted third parties (e.g., mashups), a client-side system such as BEEP or Mutation-Event Transforms can be considered a complementary layer of protection to that provided by our framework.

Several works have studied how the safety of functional languages can be improved. Xu proposed the use of pre/post-annotations to implement extended static checking for Haskell in [51]; this work has been extended in the form of contracts in [52]. Li and Zdancewic demonstrated how general information flow policies could be integrated as an embedded security sublanguage in Haskell in [31]. A technique for performing data flow analysis of lazy higher-order functional programs using regular sets of trees to approximate program state is proposed by Jones and Andersen in [24]. Madhavapeddy *et al.* presented a domain-specific language for securely specifying various Internet packet protocols in [34]. In [16], Finifter *et al.* describe Joe-E, a capability-based subset of Java that allows programmers to write pure Java functions that, due to their referential transparency, admit strong analyses of desirable security properties.

The work presented in this paper differs from those above in that our framework is designed to mitigate specific vulnerabilities that are widely prevalent on the Internet. The generality of our approach could be enhanced, however, by integrating general information flow policies, at the cost of an additional burden on developers.

Several application servers for Haskell have already been developed, most notably HAppS [19]. To the best of our knowledge, however, none of these frameworks implement specific protection against cross-site scripting or SQL injection vulnerabilities. Finally, Elsman and Larsen studied how XHTML documents can be typed in ML [12]. Their focus, however, is on generating standards-conforming documents; they do not directly address security concerns.

7 Conclusions

In this paper, we have presented a framework for developing web applications that, by construction, are invulnerable to server-side cross-site scripting and SQL injection attacks. The framework accomplishes this by strongly typing both documents and database queries that are generated by a web application, thereby automatically enforcing a separation between structure and content that preserves the integrity of these objects.

We conducted an evaluation of the framework, and demonstrated that all dynamic data that is contained in a document generated by a web application must be subjected to sanitization. Similarly, we show that all SQL queries must be executed in a safe manner. We also demonstrate the correctness of the sanitization functions themselves. Finally, we give performance numbers for representative web applications developed using this framework that compare favorably to those developed in other popular environments.

In future work, we plan to investigate how the framework can be modified to allow developers to specify "safe" transformations of document structure to occur in a controlled manner. Also, we plan to investigate static techniques for verifying the correctness of the sanitization functions in terms of their agreement with invariants extracted from web browser document parsers and database query parsers, for instance using a combination of static and dynamic analyses [3, 5]. Finally, future work will consider how language-based techniques for ensuring document integrity could be applied on the client.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments. We would also like to thank Adam Barth for providing feedback on an earlier version of this paper. This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Pro-*

ceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, CA, USA, May 2008.

[2] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, USA, October 2007.

[3] D. Balzarotti, W. Robertson, C. Kruegel, and G. Vigna. Improving Signature Testing Through Dynamic Data Flow Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, USA, December 2007.

[4] B. Bangert and J. Gardner. PylonsHQ. `http://pylonshq.com/`, February 2009.

[5] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009. IEEE Computer Society.

[6] Breach Security, Inc. Breach WebDefend. `http://www.breach.com/products/webdefend.html`, January 2009.

[7] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 2007 USENIX Security Symposium*, Boston, MA, USA, 2007. USENIX Association.

[8] Citrix Systems, Inc. Citrix Application Firewall. `http://www.citrix.com/English/PS2/products/product.asp?contentID=25636`, January 2009.

[9] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. *ACM SIGPLAN Notices*, 37(12):47–59, 2002.

[10] M. de Kunder. The Size of the World Wide Web. http://www.worldwidewebsize.com/, May 2008.

[11] Django Software Foundation. Django Web Application Framework. `http://www.djangoproject.com/`, June 2009.

[12] M. Elsman and K. F. Larsen. Typing XHTML Web Applications in ML. In *Proceedings of the* 6*th* *International Symposium on Practical Aspects of Declarative Languages*, pages 224–238. Springer-Verlag, 2004.

[13] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the* 11*th* *USENIX Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007. USENIX Association.

[14] F5 Networks, Inc. BIG-IP Application Security Manager. `http://www.f5.com/products/big-ip/product-modules/application-security-manager.html`, January 2009.

[15] Ferruh Mavituna. SQL Injection Cheat Sheet. `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/`, June 2009.

[16] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable Functional Purity in Java. In *Proceedings of the* 15*th* *ACM Conference on Computer and Communications Security*, pages 161–174, Alexandria, VA, USA, October 2008. ACM.

[17] Google, Inc. ctemplate. `http://code.google.com/p/google-ctemplate/`, June 2009.

[18] D. H. Hansson. Ruby on Rails. `http://rubyonrails.org/`, February 2009.

[19] HAppS LLC. HAppS – The Haskell Application Server. `http://happs.org/`, February 2009.

[20] Hewlett Packard Development Company, L.P. httperf. `http://www.hpl.hp.com/research/linux/httperf/`, February 2009.

[21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the* 13*th* *International Conference on the World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[22] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Emdeded Policies. In *Proceedings of the* 16*th* *International Conference on the World Wide Web*, Banff, Alberta, Canada, May 2007. ACM.

[23] M. Johns and C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of ACM Symposium on Applied Computing*, Seoul, Korea, March 2007. ACM.

[24] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1–3):120–136, 2007.

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2006)*, pages 258–263, Oakland, CA, USA, May 2006. IEEE Computer Society.

[26] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 27–36, Ottawa, Ontario, Canada, 2006. ACM.

[27] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006. ACM.

[28] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005.

[29] C. Kruegel, W. Robertson, and G. Vigna. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5):717–738, July 2005.

[30] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, San Francisco, CA, USA, 2008. ACM.

[31] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *Proceedings of the* 19^{th} *IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2006.

[32] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 95–104, San Diego, CA, USA, 2007. ACM.

[33] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the* 14^{th} *USENIX Security Symposium (USENIX Security 2005)*, pages 271–286. USENIX Association, August 2005.

[34] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "Functional Internet". In *Proceedings of the* 2^{nd} *ACM European Conference on Computer Systems*, pages 101–114, Lisbon, Portugal, April 2007. ACM.

[35] Microsoft, Inc. LINQ. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`, June 2009.

[36] Miniwatts Marketing Group. World Internet Usage Statistics. http://www.internetworldstats.com/stats.htm, May 2008.

[37] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[38] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[39] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shifley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 2005 International Information Security Conference*, pages 372–382, 2005.

[40] Ofer Shezaf and Jeremiah Grossman and Robert Auger. Web Hacking Incidents Database. `http://www.xiom.com/whid-about`, January 2009.

[41] Open Security Foundation. DLDOS: Data Loss Database – Open Source. `http://datalossdb.org/`, January 2009.

[42] Open Web Application Security Project (OWASP). OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, February 2009.

[43] C. Reis, J. Dunagan, H. J. Wang, and O. Dubrovsky. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3):11, 2007.

[44] Robert Hansen (RSnake). XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`, June 2009.

[45] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, USA, February 2006.

[46] Symantec, Inc. Symantec Report on the Underground Economy – July 07 – June 08. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf`, November 2008.

[47] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2007)*, February 2007.

[48] P. Wadler. The Essence of Functional Programming. In *Proceedings of the* 19*th* *Annual Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, USA, 1992. ACM.

[49] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. *ACM SIGPLAN Notices*, 42(6):32–41, April 2007.

[50] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE 2008)*, pages 171–180, Leipzig, Germany, 2008. ACM.

[51] D. N. Xu. Extended Static Checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 48–59, Portland, OR, USA, 2006. ACM.

[52] D. N. Xu, S. P. Jones, and K. Claessen. Static Contract Checking for Haskell. In *Proceedings of the* 36*th* *Annual ACM Symposium on the Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, 2009. ACM.