Automated Code Injection Prevention for Web Applications

Zhengqin Luo, Tamara Rezk, and Manuel Serrano

INRIA Sophia Antipolis, France
{zluo,trezk,mserrano}@inria.fr

Abstract. We propose a new technique based on multitier compilation for preventing code injection in web applications. It consists in adding an extra stage to the client code generator which compares the dynamically generated code with the specification obtained from the syntax of the source program. No intervention from the programmer is needed. No plugin or modification of the web browser is required. The soundness and validity of the approach are proved formally by showing that the client compiler can be fully abstract. The practical interest of the approach is proved by showing the actual implementation in the Hop environment.

1 Introduction

The impact of the Web 2.0 on sensitive aspects of daily life (home banking, e-commerce, social websites such as Facebook or Twitter, e-voting, etc.) has triggered an unprecedented demand of means for writing highly secured web applications. Unfortunately, their multitier architecture makes the security difficult to enforce. Usually web applications are written in a main language (*e.g.,* PHP, JSP) that executes in the main-tier and dynamically generates programs in a target language (*e.g.,* SQL, Html, JavaScript) that executes in the target-tier. Furthermore, to enhance the interaction experience between users and web applications, main-tier programs may accept (untrusted) user input. The input may be stored in a database or a persistent variable, and later on be used to generate other target-tier programs. This *dynamic* generation of target-tier programs using untrusted inputs may represent a serious application vulnerability when input is "confused" with the original code of the application to be executed. This kind of attack is known as code injection and more generally, it can be seen as an integrity violation [20]. In spite of several efforts from the security community to avoid code injection attacks, recent statistics [6,31] show that these kind of attacks (*e.g.,* cross-site scripting, SQL injection) are still the most common security vulnerabilities for web applications.

Multitier languages [29,8,10,7] have recently emerged as a response to the need of simplifying the development of web applications. A multitier language provides a unified syntax for server code and client code. Its runtime environment compiles the source into the various formats supported by the tiers. For instance, the runtime system of the Hop programming language contains three

S. Mödersheim and C. Palamidessi (Eds.): TOSCA 2011, LNCS 6993, pp. 186–204, 2011.

dynamic compilers. The first generates server-side byte code. The second generates JavaScript code. The third generates HTML. Multitier languages provide natural tools to solve code injection problems, as they allow global reasoning to be applied to the web applications.

We propose a methodology for preventing code injection in multitier languages which consists in modifying the client code compilers at the point of dynamic generation for comparing the generated code with the specification extracted from the syntax of the the source program. The methodology follows the technique for SQL injection by Su and Wasserman [30], and it also applies to other target-tier languages such as XQuery or LDAP. The methodology complements that of [30] by the following added value:

- The programmer is freed from making any intervention in order to achieve security guarantees. Indeed the expected syntax structure is not provided by the programmer, since it is already given by the syntax of the multitier program.
- Proofs are given by means of standard language-based techniques and programming language semantics. We use the multitier programming language formal semantics in order to prove that the HOP compiler is certified regarding code injection prevention. We formally prove the validity of our approach by showing that the client compiler is fully abstract.

Related Work. The WebSSARI [13], and Pixy [15,16] tools propose tainted-flow static analysis for PHP to identify where untrusted input should be validated. Xie and Aiken [34] develop a finer approximation of tainted flow at the intra-block, intra-procedural, and inter-procedural level. However, tainted flow does not guarantee proper validation of untrusted data, since any pre-defined validation procedure or user-defined filter will be considered as correct. Based on tainted flow analysis, other works propose sophisticated string analysis for assessing the correctness of the validation procedure [23,3,33,32] for SQL or web applications. Those analysis are over-approximations of the possible set of output strings that may be injection attacks. Those approaches above, whether sound or not, require explicit intervention of programmers to deal with untrusted inputs by proper validation. Our approach, on the other hand, does not require any intervention of programmer to be sound with respect to injection attacks.

Other approaches dynamically detect and prevent injection attacks. Perl's *tainted mode* [25] is one of the earliest dynamic mechanisms for disallowing untrusted input to be used in a security-sensitive context. Xu and his colleagues [35] present a policy-based solution for dynamic detection of insecure tainted information flow, where a source-to-source translation instruments programs to track tainted data. Nguyen-Tuong and his colleagues propose similar dynamic flow monitoring with a modified PHP interpreter. Run-time instruction-set randomization [17] prevents SQL injection by randomly masking SQL query keywords, making it difficult to change the intended syntax structure of query by untrusted input. Most of the solutions to prevent code injection (*e.g.,* tainted flow analysis, static string analysis, syntax embedding, etc) either do not free

the programmer from doing sanitization by themselves or require browser modifications. The work by Su and Wassermann [30] is the closest one to ours. They prevent SQL injection by comparing constructed SQL queries with given policies at run-time when the query is submitted to a database back-end. Besides the fact that the syntactic structure of SQL code does not change dynamically as is the case of generated JavaScript programs, the difference with respect to our work is that they require a separate grammar specification, whereas our approach takes source semantics directly as specification for programs to be executed in the target-tier. The work by Robertson and Vigna [27] propose a framework that uses typing to identify where untrusted inputs are used in an output HTML page, therefore to use a sanitization function to prevent the structure of the output page from being modified. However the sanitization function in their proposal is not proved to be sound. There are other alternatives to tackle the code injection problem from the client side [14,21,36,26,19]. Swift [9] and SELinks [11] use information flow security analysis to detect code injection attacks, among other techniques such as partitioning of code (in the case of Swift) and typing or filtering (in the case of SELinks). Our technique is significantly different from theirs since it does not require any integration of program static analysis. Moreover, its implementation in multitier compilers is simple and no effort is required to adapt to any target-tier language (it is not dependent of or limited to Html+JavaScript) provided that the multitier language includes appropriate target-tier language constructors at the source level and that a parser for the target-tier language is available at compilation time.

Blueprint [22] and xJS [2] are two tools that focus on keeping the intended syntax structure of server generated document when untrusted inputs are present. In both systems, untrusted content is parsed at the server-side (eliminating dynamic content), and then encoded as a model in a safe alphabet which does not trigger script evaluation. On the client-side, a safe library function is invoked to recover the untrusted content. As a result, no script evaluation will be triggered by untrusted content. In contrast to our approach, their approach works for existing developing framework of web applications, but requires the programmer to identify untrusted inputs and the context where untrusted input is output. Furthermore, there are performance penalties both on the server-side and client-side. Our solution does not require any programmer's intervention for identifying untrusted sources or context and it has no performance penalty on the client-side.

Contents. In Section 2 we give an overview of the proposed compilation technique. In Section 3 we present the Hop language and its semantics for globally describing web applications. This semantics can be thought as a high-level description of the server and (specially) the client code. In Section 4 we define a compiler from Hop programs to client code (Html+JavaScript). In Section 4.2 we introduce the compilation extension that allows us to prove the result. We discuss the Hop implementation and some practical JavaScript issues in Section 5. We conclude in Section 6.

2 Overview

In this section we give an overview of multitier web programs and informally present our approach to prevent code injection by compilation.

Web 2.0 applications are commonly composed of three tiers: the server tier or web server contains and executes server code, the client tier or browser executes client code, and the database tier contains a database and a database management system (DBMS) to execute queries.

One of the characteristics of modern web applications is the dynamic generation of code. The web server may generate a particular HTML page based on input from the client for example. Typically, without using a multitier language, dynamic generation of client code is obtained by manipulating strings. Generally the server represents HTML as a text template with holes that are to be filled with dynamic content, as shown in the following example of server code written in Java:

```
1 public class Greeting extends HttpServlet {
2   public void doGet(HttpServletRequest req,
3                     HttpServletResponse res){
4       res.setContentType("text/html");
5       PrintWriter out = res.getWriter();
6       String name = req.getParameter("name");
7       out.println("<HTML>\n<BODY>\n");
8       out.println("Greeting from " + name + "\n");
9       out.println("</BODY>\n</HTML>\n");}}
```

The web service corresponding to the code above, after receiving a request from a client with a parameter "`name`", will respond with a HTML page to be displayed in the browser, containing a greeting. Multitier programming languages, such as HOP, follow a different path mainly to harmonize the programming of the client and the server. They support a coherent unique syntax and semantics for both ends of the applications. This in general, involves supporting a Document Object Model (henceforth *DOM*) for HTML on the client as well as on the server as shown in Figure 1. This approach has proved to have some advantages over HTML textual representations. It eases the creation and manipulation of HTML documents that are represented as a regular data structures of the language. It also allows to separate the creation of a document from its actual external representation which can vary from one client to another. For instance, a HTML5 capable client might receive a document expressed in that particular HTML version while another one, less skilled, might receive it in XHTML.

In a multitier language, a semantically equivalent program, can be written using constructs from the language, as shown in the following HOP example:

```
1 (define-service (greeting name)
2   (<HTML>
3     (<BODY>
4       "Greeting from" name)))
```

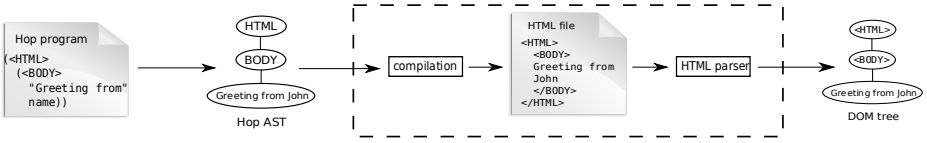

Fig. 1. AST and DOM generated trees after client request

As before, the code above will respond to a client request by a HTML page containing a greeting, but the difference is that `<HTML>` and `<BODY>` are regular library functions of the language.

In response to a client request, a HOP program generates a response which can either be *static*, for example a simple static file or string, or *dynamic*. In that case the server executes user code to dynamically generate the response content as in the example above. Only this kind of response is liable to code injection which occurs when the input of the server side computation is inserted in the generated response as a client-side executable code. A typical attack consists in stealing the client's cookies and redirecting then to an adversary's website (see for instance Figure 2). In this example, the input `name` binds to a malicious string `"<script>win...</script>"`. Therefore the HOP AST (Abstract Syntax Tree) obtained from server-side run-time environment is different from the DOM tree obtained by parsing the generated HTML document in the client's browser. Prevention of code injection only requires to add additional treatment to dynamic responses, no special treatment being required for other responses.

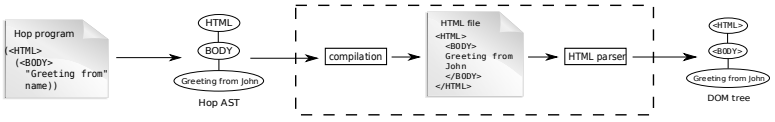

Fig. 2. Mismatch between AST tree and DOM tree

Splitting apart dynamic HTTP requests and being provided with a server-side HOP AST makes code injection detection easier: it is only needed to reproduce the DOM tree that is generated on the client-side and compare it with the HOP AST. This tree can be easily obtained by parsing the HTML document generated from the HOP AST *on the server-side* using a standard HTML parser. Then it is sufficient to compare the two trees to detect code injection attacks. If the two trees have the same shape, the program is safe and the response is sent to the client. If the two trees differ, code has been injected and an exception is raised.

In terms of compiler re-writing, code injection detection only requires to extend the code of the compiler that is in charge of delivering dynamic content.

In order to prove that this eliminates unexpected behaviors from the dynamically generated pages, we use the HOP program semantics, that abstracts away from compilation processes, as the specification of what expected behavior is.

$$
\begin{aligned}
str &\in \mathcal{S}tring \\
p,\ q,\ r\ldots &\in \mathcal{P}ointer \\
s &::= x \mid w \mid (s_0\ s_1) \mid {\sim}t \mid (\langle tag \rangle) \mid (\textsf{dom-appchild!}\ s_0\ s_1) \\
t &::= x \mid (\textsf{lambda}\ (x)\ t) \mid () \mid str \mid p \mid (t_0\ t_1) \mid \$x \\
w &::= (\textsf{lambda}\ (x)\ s) \mid {\sim}c \mid () \mid str \mid p \\
c &::= x \mid v \mid (c_0\ c_1) \\
v &::= (\textsf{lambda}\ (x)\ c) \mid () \mid str \\
tag &::= \textsf{HTML} \mid \textsf{DIV} \mid \cdots
\end{aligned}
$$

Fig. 3. Simplified Hop Syntax

In the rest of the paper, we describe the formalization and code injection soundness proofs (correspondence of the behavior of generated client code with respect to the client Hop semantics) for a subset of the Hop language. However, the implementation of the technique, that is made publicly available from the Hop web page[1], is applied to the full-fledged language, including a non-trivial set of Html attributes and constructs, as well as CSS files and JavaScript functions.

3 A Multitier Language

We present a core of the Hop language limited to a minimal set of web programming abstractions that are enough to present our compilation technique and develop its formal correctness (a larger formal description of the language that includes constructors for defining services on the server-side, calling services from the client, events handler such as "onclick" in web pages, and event loops as a form of cooperative multithreading [1], can be found in [5]).

3.1 Syntax

The syntax is given in Figure 3, where x denotes any variable. We assume given a set $\mathcal{S}tring$ of strings; a set $\mathcal{P}ointer$ of pointers. Those sets are mutually disjoint, and are also disjoint from the set of variables. Pointers are run-time values for denoting Html tree nodes. The simplified Hop syntax is stratified into *server code* s and *tilde code* t. The former is basically Scheme [18] code enriched with a construct $\sim t$ to ship (tilde) code t to the client, and constructs to dynamically build Html trees. The latter may include references $\$x$ to server values, and will be translated into *client code* c, before being shipped to the client.

A server expression usually contains sub-expressions of the form $\sim t$. As we said, t represents code that will be executed on the clients. This code may use values provided by the server, by means of sub-expressions $\$x$. When the latter are absent (that is, when they have been replaced by a value bound to x), a t

[1] `http://hop.inria.fr`

expression reduces to a client expression c. Notice that for the server an expression $\sim c$ is a value, meaning that the evaluation of code c is delayed until installed on a client site. Server code syntax is enriched with some basic HTML constructs, written in Scheme style, and operations supported by the DOM. Here we confine ourselves to consider the HTML and DIV tags, and the $(\mathsf{dom\text{-}appchild!}\, s_0\, s_1)$ construct – the other ones are similar (see [12]). The general form of HTML constructors in HOP is $(\langle tag\rangle\, [:attr])$ where *attr* is an optional list of attributes. For simplicity, we only consider here the cases $(\langle tag\rangle)$ where there is no attribute. A more general form of creating a node with an arbitrary number of children can be defined as syntactic sugar. For example, creating a node with one child can be defined as follows:

$$(\langle tag\rangle\, s) ::= ((\mathsf{lambda}\, (x)\, (\mathsf{dom\text{-}appchild!}\, (\langle tag\rangle\,)\, x))\, s)$$

Values also include strings and $()$, which is a shorthand for the *unspecified* Scheme run-time value. As usual $(\mathsf{lambda}\, (x)\, s)$ binds x in the expression s.

A HOP *program* is a *closed* expression s, meaning that it does not contain any free variable. We shall consider expressions up to α-conversion, that is up to the renaming of bound variables, and we denote by $s\{y/x\}$ the expression resulting from substituting the variable y for x in s, possibly renaming y in sub-expressions where this variable is bound, to avoid captures. The operational semantics of the language will be described as a transition system, where at each step a (possibly distributed) *redex* is reduced. As usual, this occurs in specific positions in the code, that are described by means of *evaluation contexts*. The syntax of evaluation contexts is as follows:

$$\mathbf{S} ::= [] \mid (\mathbf{S}\, s) \mid (w\, \mathbf{S}) \mid (\mathsf{dom\text{-}appchild!}\, \mathbf{S}\, s) \mid (\mathsf{dom\text{-}appchild!}\, w\, \mathbf{S})$$
$$\mathbf{C} ::= [] \mid (\mathbf{C}\, c) \mid (v\, \mathbf{C})$$

As usual, we denote by $\mathbf{S}[s]$ (resp. $\mathbf{C}[c]$) the result of filling the hole $[]$ in context $\mathbf{S}$ (resp. $\mathbf{C}$) with expression s (resp. c).

3.2 Hop Web Application Semantics

The semantics of a HOP web application is represented as a sequence of transitions between configurations. Specific features that are modeled in the semantics in order to capture the behaviour of web applications include: dynamic client code generation and delivery, script nodes execution from a DOM tree, dynamic DOM tree modification. A configuration consists in

- a server configuration S, together with an environment μ providing the values for the variables occurring in the server configuration. For simplicity, we consider that the server configuration consists in a single thread at the time executing server's code to answer client's requests to services.
- a client configuration C, which consists in one running client (extension to multiple clients is straightforward [5] but we prefer to simplify notation here). A client is a tuple $\langle c, \mu, r\rangle$ where c is the client code and μ is the local

environment for the client distinct from the one of the server (the client and the server do not share any state). The pointer r is the *root* of the HTML page that is displayed at the client site by the browser.
- a HOP environment ρ, which binds URLs to services s (we assume given a set $\mathcal{U}rl$ of names denoting URLs);

Then a configuration Γ has the form $((S, \mu), C, \rho)$. However, to simplify the semantic rules, and to represent the concurrent execution of the various components, we shall use the following syntax for configurations:

$$\Gamma \ ::= \ \mu \ \mid \ \rho \ \mid \ s \ \mid \ \langle c, \mu, r \rangle \ \mid \ (\Gamma \parallel \Gamma)$$

We assume that parallel composition $\parallel$ is commutative and associative, so that the rules can be expressed following the "chemical style" of [4]:

$$\frac{\Gamma \rightarrow \Gamma'}{(\Gamma \parallel \Gamma'') \rightarrow (\Gamma' \parallel \Gamma'')}$$

meaning that if the components of Γ are present in the configuration, which can therefore be written $(\Gamma \parallel \Gamma'')$, and if these components interact to produce Γ', then we can replace the components of Γ with those of Γ'.

Before introducing and commenting the reaction rules, we define an auxiliary function transforming tilde code into client code. As we said a sub-expression $\sim t$ in server code is *not* evaluated at server side, but will be shipped to the client, usually as the answer to a service request. Since the expression t may contain references $\$x$ to server values, to define the semantics we introduce an auxiliary function Ξ that takes as arguments an environment μ and an expression t, and transforms it into a client expression c. The Ξ transformation consists in replacing $\$x$ by the value bound to x in μ. For example, if $\mu = \{x \mapsto \texttt{"text"}\}$, then we have

$$\Xi(\mu, \sim((\mathsf{lambda}\ (y)\ y)\ \$x)) = \sim((\mathsf{lambda}\ (y)\ y)\ \texttt{"text"})$$

(The interested reader can check for a formal definition in a previous paper [5]). One should notice that a function, that is a $(\mathsf{lambda}\ (x)\ s)$, or client code c cannot be sent to the client this way, because this would in general result in breaking the bindings of free variables that may occur in such an expression. Then this has to be considered as an error. The semantics of the $(\langle tag \rangle)$ construct is that it builds a node of a tree in a *forest*. In order to define this, we assume given a specific null pointer, denoted α, which is not in $\mathcal{P}ointer$. We use π to range over $\mathcal{P}ointer \cup \{\alpha\}$. Then a forest maps (non null) pointers to pairs made of a (possibly null) pointer and an expression of the form $(\langle tag \rangle\, c_1 + \cdots + c_n)$. The pointer $q \in \mathcal{P}ointer$ assigned to p is the *ancestor* of the node, if it exists. If it does not, this pointer is α. Such a node is labeled tag and has n children, which are either leaves (labeled with some client code or value) or pointers to other nodes in the tree. For simplicity we consider here the forest as joined to the environment providing values for variables. That is, we now consider that μ is

a mapping from a set $\mathsf{dom}(\mu)$ of variables and (non null) pointers, that maps variables to values, and pointers to pairs made of a (possibly null) pointer and a *node* expression. The syntax for node expressions a is as follows:

$$\begin{aligned} a &::= (\langle tag\rangle\ \ell) \\ \ell &::= \varepsilon \mid c \mid (\ell_0 + \ell_1) \end{aligned}$$

where ε is the empty list. In what follows we assume that $+$ is associative, and that $\varepsilon + \ell = \ell = \ell + \varepsilon$. We shall also use the following notations in defining the semantics, assuming that the pointers occurring in the list ℓ are distinct:

$$\begin{aligned} (\langle tag\rangle\ \ell) + p &= (\langle tag\rangle\ \ell + p) \\ (\langle tag\rangle\ \ell_0 + p + \ell_1) - p &= (\langle tag\rangle\ \ell_0 + \ell_1) \end{aligned}$$

Given a forest μ, and $p \in \mathsf{dom}(\mu)$, we denote by $\mu[p \mapsto (\pi, a)]$ the forest obtained by updating the value associated with p in μ. For $r \in \mathcal{P}ointer$, we also define $\mu \lceil r$ to be the part of the forest that is reachable from r (a formal definition can be found in [5]). For example:

$$\mu = \begin{bmatrix} r \mapsto (\alpha, \langle \mathsf{HTML}\rangle\ \texttt{"text"} + p), \\ p \mapsto (r, \langle \mathsf{DIV}\rangle\ \texttt{"text"}), \\ q \mapsto (\alpha, \langle \mathsf{DIV}\rangle\ \texttt{"text"}) \end{bmatrix}, \mu \lceil r = \begin{bmatrix} r \mapsto (\alpha, \langle \mathsf{HTML}\rangle\ \texttt{"text"} + p), \\ p \mapsto (r, \langle \mathsf{DIV}\rangle\ \texttt{"text"}) \end{bmatrix}$$

An excerpt of the semantics rules is given in Figure 4. Rules for variable look-up and function application are standard and left out (a complete set of rules can be found in [5]) . The Tilde rule transforms tilde code containing $\$x$

$$\frac{\Xi(\mu, t) = c}{\mathbf{S}[\sim t] \parallel \mu \rightarrow \mathbf{S}[\sim c] \parallel \mu}\ (\textsc{Tilde}) \qquad \frac{\mu(r) = (\alpha, (\langle \mathsf{HTML}\rangle\ \ \ell))}{r \parallel \mu \parallel \langle \langle\rangle, \emptyset, \alpha\rangle \rightarrow \mu \parallel \langle \langle\rangle, \mu \lceil r, r\rangle}\ (\textsc{ServRet})$$

$$\frac{\rho(u) = w \quad v \neq (\mathsf{lambda}\,(x)\,c)}{\rho \rightarrow (w\,v) \parallel \langle \langle\rangle, \emptyset, \alpha\rangle \parallel \rho}\ (\textsc{Init})$$

$$\frac{\mathsf{R}(r, p, \mu) \quad \mu(p) = (q, (\langle tag\rangle\ \ \ell_0 + c + \ell_1))}{\langle v, \mu, W, r\rangle \rightarrow \langle c, \mu[p \mapsto (q, (\langle tag\rangle\ \ \ell_0 + \ell_1))], W, r\rangle}\ (\textsc{Script})$$

$$\frac{p \notin \mathsf{dom}(\mu)}{\mathbf{S}[(\langle tag\rangle)] \parallel \mu \rightarrow \mathbf{S}[p] \parallel \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\,\varepsilon))\}}\ (\textsc{TagS})$$

$$\frac{\mu(p) = (\pi, a_0) \quad \mu(q) = (q', a_1) \quad \mu(q') = (\pi', a_2) \quad p \neq q' \quad \neg\mathsf{R}(q, p, \mu)}{\mathbf{S}[(\mathsf{dom\text{-}appchild!}\,p\,q)] \parallel \mu \rightarrow \mathbf{S}[\langle\rangle] \parallel \mu \begin{bmatrix} p \mapsto (\pi, a_0 + q), \\ q \mapsto (p, a_1), \\ q' \mapsto (\pi', a_2 - q) \end{bmatrix}}\ (\textsc{AppendS1})$$

Fig. 4. Excerpt of Hop Semantics

expressions into a server value. The SERVRET rule is the key rule in the semantics that, in the case of the high-level semantics, allows us to specify which code is executed on the client site. The SERVRET rule can be used once a service in the server has finished its evaluation, and its result is shipped to the client. The kind of service results we are interested in are pointers representing a HTML fragment or document, possibly containing one or more script nodes to execute. In the high-level semantics, the HTML document with script nodes is constructed by HOP HTML constructors. Tilde codes are dynamically evaluated by possibly embedding values obtained with $\$x$ expressions. The SERVRET rule sets the root of the client document to be the result value pointer r and the client environment to $\mu \lceil r$ that represents exactly the HTML tree that hangs from pointer r in the server store.

The INIT rule creates a new instance of a service and initializes its execution with any argument v. This rule is intentionally made non-deterministic to model that the client can provide any input to the service. This argument is (untrusted) input provided by the client. On the client site, there is the SCRIPT rule that models execution of script node contained in the client HTML document. We use the predicate $\mathsf{R}(r, p, \mu)$ to state that pointer p is a descendant of r in μ, and that the code that we find at node p, and which is to be triggered, is the leftmost one in the tree $\mu \lceil r$ determined by r. (We should also check that this tree is still a valid HTML document. We do not formally define this predicate here – this is straightforward.) An example illustrates the predicate R in Figure 5. Finally the APPENDS1rule modifies the DOM by appending a child to an existing node in the store. Notice that the rule preserves the uniqueness of pointers in list presentation.

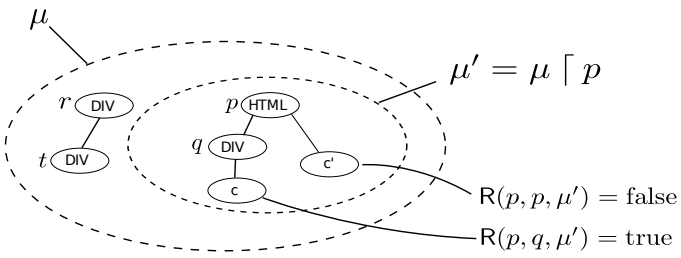

Fig. 5. Example: The predicate R

3.3 A More Fine-Grained Semantics

The high-level SERVRET rule abstractly defines HOP client code and its intended behavior, that is, the HTML tree should be delivered intact to the client. We present a more detailed semantics that consists in a dynamic compilation phase compiling a HOP AST tree to HTML + JavaScript code to transfer the document constructed in the server's store to a client thread. This appears in a new version of the rule SERVRET. At the client-side, the HTML parser (in the browser) interprets this HTML document, possibly invoking the JavaScript engine to execute script nodes.

$$
\begin{array}{lll}
str \in \mathcal{S}tring & & string \\
c_{js} ::= x \mid v_{js} \mid c_{js}(c'_{js}) & & code \\
v_{js} ::= \mathsf{function}(x)\{\mathsf{return}\ c_{js}\} \mid \langle\rangle \mid str & & values
\end{array}
$$

Fig. 6. JavaScript (abstract) Syntax

Client-side JavaScript Syntax and Semantics. We give a formal treatment to the syntax and semantics of JavaScript in order to prove correctness in following sections. The abstract syntax of a small core of JavaScript is shown in Figure 6. Its syntax is self-explanatory. We omit usual definitions such as variable substitution. The evaluation context is defined as follows:

$$\mathbf{J} ::= [] \mid \mathbf{J}(c_{js}) \mid v_{js}(\mathbf{J})$$

We overload notation $C = \langle c_{js}, \mu, r\rangle$ for client configuration in the low-level semantics, where c_{js} and μ are JavaScript code and store, respectively. The definition of list ℓ for node representation is also updated, since in the low-level semantics script nodes are now JavaScript expressions. The shape of a server configuration and global configuration are left unchanged. As shown in Figure 7, rule VarJS and AppJS are unsurprising, which evaluate redex for client-side JavaScript code.

Client Dynamic Compilation. The dynamic compilation is parametrized with a Hop client compiler C and a Html parser P. The Hop client compiler C transforms a server Hop store and a pointer that represents an abstract Hop tree with Hop client code ($\mu \lceil r$) into an actual Html document that contains JavaScript code. The Html parser P (which includes a JavaScript parser), on the client-side, will parse any Html document and produce a pair of store and root pointer (μ_c, p) as the DOM tree to be rendered (see Figure 1). As shown in Figure 7, instead of abstractly transferring part of the server's store $\mu \lceil r$, the new ServRetLow rule first use Hop client compiler to compile the abstract tree into a Html document *doc*. Then it uses the Html parser P to parse *doc*, in order to produce a DOM tree for the client. The network transmission of

$$\frac{\mu(x) = v_{js}}{\langle \mathbf{J}[x], \mu, p\rangle \rightarrow \langle \mathbf{J}[v_{js}], \mu, p\rangle}\ (\textsc{VarJS})$$

$$\frac{y \notin \mathsf{dom}(\mu)}{\langle \mathbf{J}[\mathsf{function}(x)\{\mathsf{return}\ c_{js}\}(v_{js})], \mu, p\rangle \rightarrow \langle \mathbf{J}[c_{js}\{y/x\}], \mu \cup \{y \mapsto v_{js}\}, p\rangle}\ (\textsc{AppJS})$$

$$\frac{doc = \mathsf{C}(\mu, r) \quad \mathsf{P}(doc) = (\mu_c, p) \quad \mu_c(p) = (\alpha, (\langle \mathsf{HTML}\rangle\ \ell))}{\mu \parallel \langle \langle\rangle, \emptyset, \alpha\rangle \rightarrow \mu \parallel \langle p, \mu_c, p\rangle}\ (\textsc{ServRetLow})$$

Fig. 7. Low-level semantics

the HTML document is made implicit in the rule. The rest of the rules are left unchanged from the high-level semantics.

4 Client-Code Compilation

In this section we first show that a naive compiler may cause the fine-grained semantics to have more undesired behaviors than the high-level semantics, which are code injection attacks. Then we show how to modify the naive compiler by using the *tree-comparison technique* to obtain a secured HOP client compiler that prevents code injection attacks. We give formal proof that the secured compiler is code-injection free, that is, it has no more behavior than the high-level semantics.

4.1 A Naive Client Compiler

The compiler C_a translates a server's store and a pointer to a HTML document *doc* by simply concatenating HTML tags and content. It also uses a JavaScript compiler C_j that compiles HOP client code into JavaScript. This naive compiler is defined in Figure 8(c), in which letters in `typewriter font` represent string characters and "·" represent string concatenation (omitted when unambiguous)[2].

$$\mathsf{C}_j(c) ::= \begin{cases} \texttt{"}\mathit{unquote}(\mathit{str})\texttt{"} & c = \mathit{str} \\ \texttt{\#unspec} & c = () \\ \mathit{ident}(x) & c = x \\ \texttt{function(}\mathit{ident}(x)\texttt{)}\ \{\texttt{return}\ \mathsf{C}_j(c_0)\} & c = (\mathsf{lambda}\,(x)\,c_0) \\ \mathsf{C}_j(c_0)\texttt{(}\mathsf{C}_j(c_1)\texttt{)} & c = (c_0\,c_1) \end{cases}$$

(a) JavaScript compilation

$$\mathsf{C}_a(\mu, w) ::= \begin{cases} \mathit{str} & w = \mathit{str} \\ \texttt{""} & w = () \\ \varepsilon & \ell = \varepsilon \\ \mathsf{C}_a(\mu, \ell_0) \cdot \mathsf{C}_a(\mu, \ell_1) & \ell = \ell_0 + \ell_1 \\ \texttt{<tag>}\mathsf{C}_a(\mu, \ell)\texttt{</tag>} & w = p \text{ and } \mu(w) = (q, (\langle \mathit{tag} \rangle\, \ell)) \\ \texttt{<script>}\mathsf{C}_j(c)\texttt{</script>} & w = {\sim}c \end{cases}$$

(b) HTML compilation

Fig. 8. HOP dynamic compilation

Example. Let us illustrate how the fine-grained semantics has more behaviors than the high-level one. Figure 9(a) shows a HOP program constructing a simple HTML document depending on input `name`. The compilation using C_a is shown in Figure 9(b), if the input provided is the string `Alice`. Furthermore, the initial client configuration obtained by SERVRETLOW rule is $\langle \mu_c, \emptyset, p \rangle$, where $\mathit{str} =$ `Alice`, and

$$\mu_c = \{p \mapsto (\alpha, (\langle \mathsf{HTML} \rangle\, q)), q \mapsto (p, (\langle \mathsf{DIV} \rangle\, \mathit{str}))\}$$

[2] In JavaScript compiler C_j, we assume that the *unquote* function escapes a string to its string literal representation (*e.g.,* "`a"b`" is escaped to "`a\"b`"), and *ident* is a bijection that maps each variable name to a unique string (*e.g.,* variable name x is mapped to `x`).

```
1 (lambda (name)
2    (<HTML>
3       (<DIV>
4          name)))
```

(a) Hop program

```
1 <HTML>
2   <DIV>
3     Alice
4   </DIV>
5 </HTML>
```

(b) Compilation results

Fig. 9. Example: Fine-grained Semantics

We can observe that the store obtained by ServRetLow rule and ServRet are equivalent up to pointer renaming.

However, the fine-grained semantics is not simulated by the high-level semantics presented in the previous section, since more behaviors may appear on the client side by means of a code injection attack [3]. If input `name` is

```
<script>function(x){return x;}("str")</script>
```

we would obtain a Html page by compilation as follows:

```
1 <HTML><DIV><SCRIPT>
2 function(x){return x;}("str")
3 </SCRIPT></DIV></HTML>
```

Observe that the corresponding store μ_c obtained by ServRetLow rule and the parser P is :

$$\begin{aligned}\mu_c = \{&p \mapsto (\alpha, (\langle \mathsf{HTML} \rangle\, q))\} \\ &\cup \{q \mapsto (p, (\langle \mathsf{DIV} \rangle\, str + \mathsf{function}(x)\{\mathsf{return}\ x\}(str)))\}\end{aligned}$$

The same configuration cannot be obtained by the ServRet rule in the high-level semantics.

4.2 A Secure Client Compiler

In order to define a secure Hop client compiler, we need to introduce a relation $\approx$ between a high-level client store μ and a low-level client store μ'. Informally, two stores are in the relation if their tree structures are the same. Definition of $\approx$ is given in Figure 10. For simplicity, we assume in the definition that two pointers are indistinguishable if they are equal (however, this assumption could be easily relaxed by standard indistinguishability definitions up to a bijection on pointer names). Note that pointers are used here just to describe tree representations in the semantics. In the actual implementation, the parser does not return a store but rather a DOM tree. The translation $\mathcal{C}_{js}$ from Hop client code to JavaScript is as follows:

$$\frac{}{\mathcal{C}_{js}(()) = ()} \qquad \frac{}{\mathcal{C}_{js}(str) = str} \qquad \frac{\mathcal{C}_{js}(c) = c_{js} \quad \mathcal{C}_{js}(c') = c'_{js}}{\mathcal{C}_{js}((c\,c')) = c_{js}(c'_{js})}$$

[3] This attack is not particular harmful. This is just for demonstrating that the adversary has the ability to inject code.

$$\frac{}{\varepsilon \approx \varepsilon} \qquad \frac{\mathcal{C}_{js}(c) = c_{js}}{c \approx c_{js}}$$

$$\frac{\ell_0 \approx \ell_0' \quad \ell_1 \approx \ell_1'}{\ell_0 + \ell_1 \approx \ell_0' + \ell_1'} \qquad \frac{\ell \approx \ell'}{(p, (\langle tag \rangle\, \ell)) \approx (p, (\langle tag \rangle\, \ell'))}$$

$$\frac{\mathsf{dom}(\mu) = \mathsf{dom}(\mu') \qquad \forall p \in \mathsf{dom}(\mu).\mu(p) \approx \mu'(p)}{\mu \approx \mu'}$$

Fig. 10. Tree indistinguishability

$$\frac{\mathcal{C}_{js}(c) = c_{js}}{\mathcal{C}_{js}((\mathsf{lambda}\,(x)\,c)) = \mathsf{function}(x)\{\mathsf{return}\ c_{js}\}}$$

The secure Hop client compiler is then defined as follows:

$$\mathsf{C}_s(\mu, r) = \begin{cases} \mathsf{C}_a(\mu, r) & \text{if } \mathsf{P}(\mathsf{C}_a(\mu, r)) = (\mu_c, p) \text{ and } \mu \lceil r \approx \mu_c \\ & \text{and } p = r \\ \bot & \text{otherwise} \end{cases}$$

It is built on top of the naive compiler C_a. The secure version of the compiler returns the result generated by C_a, only if the source tree $(\mu \lceil r, r)$ is indistinguishable from the tree (μ_c, p) obtained by parsing (on the server-side) the generated document. Otherwise, it returns $\bot$, which raises an exception on the server-side.

Assumption on the Html *parser.* Our technique requires just a standard Html parser on the server-side. The implication is that only *valid* Html page is delivered. Therefore injection attacks caused by ill-formed Html and browser parsing quirks are not possible in our approach, since ill-formed Html is never output to the client.

Formal correctness. We use $\rightarrow_h$ to denote high-level semantics transitions, and $\rightarrow_l$ to denote low-level semantics transitions in order to distinguish them in the following definition. We define the simulation $\Gamma \succ \Gamma'$, where Γ is a high-level configuration and Γ' is low-level configuration.

Definition 1. *Let* $\Gamma_0 = ((S_0, \mu_0), C_0, \rho_0)$ *and* $\Gamma_1 = ((S_1, \mu_1), C_1, \rho_1)$*). The simulation* $\succ$ *is the largest relation* $\mathcal{S}$ *on configurations such that* $\Gamma_0\ \mathcal{S}\ \Gamma_1$ *implies:*

1. $\mu_0 = \mu_1$
2. $C_0 = \langle c, \mu, r\rangle$, $C_1 = \langle c_{js}, \mu_{js}, r\rangle$ *such that* $\mu \approx \mu_{js}$

and if $\Gamma_1 \rightarrow_l \Gamma_1'$*,then there exists* Γ_0' *such that* $\Gamma_0 \rightarrow_h \Gamma_0'$ *and* $\Gamma_0'\, \mathcal{S}\, \Gamma_1'$.

Finally, we state a theorem that implies that the client secure compiler is fully abstract, given the correctness of the Hop to JavaScript compiler.

Theorem 1. *Let $\Gamma = \Gamma' = ((\epsilon, \epsilon), \epsilon, \rho)$, where Γ is a high-level configuration and Γ' is a low-level configuration, and ρ is a* Hop *services environment. If the fine-grained semantics use the secured client compiler C_s, we have $\Gamma \succ \Gamma'$.*

The theorem claims the soundness of our approach to eliminate code injection attacks. The proof is a standard simulation proof: we show that the semantics of delivered client code (standard Html+JavaScript) simulates the high-level semantics of client Hop code. The proof can be found in an accompanying technical report[4].

5 Practical Discussion

In this section we report about an experiment that evaluates the penalty imposed by the integrity enforcement, and we discuss how the tree comparison technique can be applied to traditional programming languages for Web applications. In Appendix A we discuss some practical issues of the implementation.

Performance. We have compared the execution times of several Hop programs with security enforcement enabled and disabled (Figure 11). We have measured the time needed by the server to deliver dynamic contents. Using the `httperf` tools [24], we have exercised the server with a repeated request up to an execution time (`usr + sys`) of about 10 seconds for the unchecked version. All requests are sent in the same HTTP connection using HTTP keep-alive annotations. The server is executed on an Intel Xeon W3570 running at 3.2GHz with 6GB of memory. For each execution, we report on the execution time expressed in seconds. Here is a short description of each tested program. `Hoppanel` (700 loc) generates a web page presenting all the installed Hop programs on a host. `Hopclock` (1100 loc), is a web wall clock. `Hoppmt` (160 loc) is a web loan calculator. `Hoptris` (740 loc) is Tetris on a web browser. `Hopstick` (800 loc) provides stickers on a web browser. `Hopphoto` (2130 loc) is a web photo browser. `Hopfile` (550 loc) is a web server-side file browser.

The impact on the performance of security enforcement depends on the nature of the executed programs, and can be noted only on the server side. Once the code is installed on the browser, there is no performance cost with respect to unsecure programs. The slowdown ratio in the server ranges from 1.38 to 4.23. We believe, however, these are not significant performance penalty for several following reasons.

1. It must be noted that at this early stage of the implementation no optimization is applied to the tree comparison. The security manager writes and parses the entire Html documents. This is expensive when the documents are large. Minimizing this cost will be subject of future work and may include sound sanitization techniques for untrusted inputs.

[4] `http://www-sop.inria.fr/members/Zhengqin.Luo/papers/acipwa-long.pdf`

benchmark	*secure*	*unsecure*	δ
`hoppanel`	43.56s	10.29s	*4.23*
`hopclock`	31.77s	10.50s	*3.02*
`hoppmt`	34.68s	12.59s	*2.75*
`hoptris`	33.57	12.56s	*2.67*
`hopsticker`	14.43	9.98s	*1.45*
`hopphoto`	16.36	11.87s	*1.38*
`hopfile`	23.97s	10.44s	*2.30*

Fig. 11. This performance evaluation measures the impact of security enforcement in Hop. The *secure* column reports on the execution times with security enforcement. The *unsecure* column reports on the execution times in seconds without security enforcement. The column δ shows the slowdown ratio.

2. It must be observed that these measures represent worst cases because for this experiment only the times to get the dynamic parts of the applications have been measured. For displaying this page in the browser, there are several other static content to be load, such as Hop run-time library and images. Since those content are not dynamically generated, there is no run-time penalty for these content. By counting the time that the server need to deliver them, we can bring down the overall run-time penalty.
3. As already demonstrated in a previous experiment [28], Hop can be more than one order of magnitude faster than common web platforms based on Apache or Tomcat for delivering dynamic documents. Hence, even with security enforcement enabled, Hop still is one of the fastest runtime environments for delivering Web 2.0 content.

Remarks on Other Programming Languages. The tree comparison technique presented in Section 4 can also be applied to other traditional programming languages for Web applications, with additional efforts. For example, to apply this technique on a Web server that supports PHP requires to modify the Web server in a non-trivial way:

1. For each PHP program that generates a Html page, associate it with a separate function that computes an AST as specification depending on given inputs.
2. Upon receiving a HTTP request from a client of a PHP program, the Web server invokes the corresponding specification function on received input, obtaining an AST. It then executes the PHP program with a PHP interpreter, and parses the output of the program with a Html parser, obtaining another Html tree. The server delivers the Html output only if the trees are of the same shape.

Comparing to the multitier programming languages approach, applying tree comparison technique on traditional programming languages for the Web requires further modification on Web servers and programmers' intervention to write specification function.

6 Conclusion

We have shown that multitier languages provide appropriate tools to eliminate code injection. In particular these kind of languages provide an essential ingredient to detect code injection: the intended semantics of the web application. We propose a compilation technique and prove its correctness in the Hop language. The technique does not require any kind of browser modification nor any annotation from the programmer. We believe that its simplicity makes it suitable for any kind of multitier compiler.

References

1. Abadi, M., Plotkin, G.D.: A model of cooperative threads. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 29–40. ACM, New York (2009)
2. Athanasopoulos, E., et al.: xJS: Practical XSS Prevention for Web Application Development. In: Proceedings USENIX Conference on Web Application Development (WebApps 2010), Boston, USA (June 2010)
3. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: IEEE Symposium on Security and Privacy, pp. 387–401 (2008)
4. Berry, G., Boudol, G.: The chemical abstract machine. In: Proceedings of the ACM International Conference on Principle of Programming Languages (POPL), pp. 81–94. ACM Press, New York (1990)
5. Boudol, G., Luo, Z., Rezk, T., Serrano, M.: Towards reasoning for web applications: an operational semantics for hop. In: APLWACA 2010, pp. 3–14 (2010)
6. Cenzic Inc. Web application security trends report Q1-Q2, 2009 (2010), `http://www.cenzic.com/`
7. Chlipala, A.: Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In: PLDI (2010)
8. Chong, S., Liu, J., Myers, A., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Building secure web applications with automatic partitioning. Communications of the ACM 52(2), 79–87 (2009)
9. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web application via automatic partitioning. In: SOSP, pp. 31–44 (2007)
10. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
11. Corcoran, B.J., Swamy, N., Hicks, M.W.: Cross-tier, label-based security enforcement for web applications. In: SIGMOD Conference, pp. 269–282 (2009)
12. Gardner, P., Smith, G., Wheelhouse, M., Zarfaty, U.: DOM: Towards a formal specification. In: Proceedings of the ACM SGIPLAN workshop on Programming Language Technologies for XML (PLAN-X), California, USA. ACM Press, New York (January 2008)
13. Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y.: Securing web application code by static analysis and runtime protection. In: WWW, pp. 40–52 (2004)
14. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: WWW, pp. 601–610 (2007)

15. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: PLAS 2006: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, pp. 27–36. ACM, New York (2006)
16. Jovanovic, N., Krügel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: IEEE Symposium on Security and Privacy, pp. 258–263 (2006)
17. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: ACM Conference on Computer and Communications Security, pp. 272–280 (2003)
18. Kelsey, R., Clinger, W.D., Rees, J.: Revised^5 report on the algorithmic language scheme. SIGPLAN Notices 33(9), 26–76 (1998)
19. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In: SAC 2006: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 330–337. ACM, New York (2006)
20. Li, P., Mao, Y., Zdancewic, S.: Information integrity policies. In: Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST) (September 2003)
21. Livshits, V.B., Erlingsson, Ú.: Using web application construction frameworks to protect against code injection attacks. In: PLAS, pp. 95–104 (2007)
22. Louw, M.T., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: IEEE Symposium on Security and Privacy, pp. 331–346 (2009)
23. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW, pp. 432–441 (2005)
24. Mosberger, D., Jin, T.: httperf: A tool for Measuring Web Server Performance. In: First Workshop on Internet Server Performance, pp. 59–67. Association for Computing Machinery (ACM), New York (1998)
25. The Perl Programming Language, `http://www.perl.org`
26. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: Browsershield: Vulnerability-driven filtering of dynamic html. ACM Trans. Web 1(3), 11 (2007)
27. Robertson, W.K., Vigna, G.: Static enforcement of web application integrity through strong typing. In: USENIX Security Symposium, pp. 283–298 (2009)
28. Serrano, M.: HOP, a fast server for the diffuse web. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 1–26. Springer, Heidelberg (2009)
29. Serrano, M., Gallesio, E., Loitsch, F.: HOP, a language for programming the web 2.0. In: Proceedings of the First Dynamic Languages Symposium, DLS, Portland, Oregon, USA (October 2006)
30. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL, pp. 372–382 (2006)
31. The MITRE Corporation. 2010 CWE/SANS top 25 most dangerous programming errors
32. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI, pp. 32–41 (2007)
33. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: ICSE, pp. 171–180 (2008)
34. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium, pp. 179–192 (2006)
35. Xu, W., Bhatkar, E., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: In 15th USENIX Security Symposium, pp. 121–136 (2006)
36. Yu, D., Chander, A., Islam, N., Serikov, I.: Javascript instrumentation for browser security. In: POPL, pp. 237–249 (2007)

A Practical Code Injection Prevention in Hop

A code is considered to be injected when a user input string ends up being interpreted by the browsers as a client-side expression. To prevent this Hop imposes that all client-side codes should be generated by the Hop client-side compiler.

In Html there are two ways to specify client-side expressions: *i)* by using `SCRIPT` nodes in the Html tree, and *ii)* by including JavaScript expressions in the Html nodes attributes. Hop handles these two situations differently.

First, the tree comparison presented in this paper ensures that no `SCRIPT` node is maliciously generated by the untrusted input in the server. The security enforcement takes place during the compilation of the AST into HTML. Prior to generating the actual text of a HTML response, a pre-processor first writes that response into a temporary file, parses it back, and compares the spine of the two trees. The response tree is immune to code injection if and only if the spines of the two trees are equivalent.

Second, a simple filtering rejects all attributes of nodes. In the Html specification, those attribute strings as event handler will be interpreted as script expressions. Hop imposes that these attributes are bound to Hop client-side expressions, not to strings. For instance, it rejects

```
(<DIV> :onclick "alert( msg )" ...)
```

but it accepts

```
(<DIV> :onclick ~(alert msg) ...)
```

Since Hop offers no means for transforming a string of characters into a *tilde code* expression, this simple filtering technique ensures that attributes as event handlers cannot be used to inject arbitrary expressions.

Beyond the Html specification, most browsers interpret attributes values prefixed with the string `javascript:` as listener attributes. For instance, the following Html link, when clicked, evaluates the `alert` function call.

```
<A href="javascript:alert('foo')">click me</a>
```

Obviously this extension could also lead to code injection. To solve that problem we have adopted a conservative solution that disables this extension by forbidding the `javascript:` prefix for all attributes. This is enforced by Hop for Html trees as well as for CSS declarations.

Finally, Hop also ensures that pure client-side manipulation cannot yield to executing new codes. For that, the client-side runtime library only binds JavaScript functions that are safe. For instance, if functions such as `eval` or `document.write` were accessible in Hop they could be used to evaluate arbitrary user forged code. The dangerous functions are either not included or slightly modified. For instance facilities such as `innerHTML` that parses an inserted string to a Html tree is kept, but its argument must be a Html node instead of a string.