Preventing Injection Attacks with Syntax Embeddings

4

ABSTRACT

Software written in one language often needs to construct sentences in another language, such as SQL queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, the concatenation of constants and client-supplied strings. A client can then supply specially crafted input that causes the constructed sentence to be interpreted in an unintended way, leading to an *injection attack*. We describe a more natural style of programming that yields code that is impervious to injections *by construction*. Our approach embeds the grammars of the *guest languages* (e.g. SQL) into that of the *host language* (e.g. Java) and automatically generates code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. This approach is generic, meaning that it can be applied with relative ease to any combination of host and guest languages.

4.1 INTRODUCTION

In this chapter we propose using *syntax embedding* to prevent injection vulnerabilities in a language-independent way. Injections form a very common class of security vulnerabilities [Halfond et al. 2006]. Software written in one language often needs to construct sentences in another language, such as SQL, XQuery, or XPath queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, whereby constant and client-supplied strings are concatenated to form the sentence. Consider for example the following piece of server-side Java code that authenticates a remote HTTP user against a database, where getParam() returns a string supplied by the user, for instance through a form field:

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

On testing, this code may appear to work correctly, but it is vulnerable to a widely known security flaw. For instance, if the user specifies as the password the string `' OR 'x' = 'x`, then the constructed SQL query will be

```
SELECT id FROM users WHERE name = '...' AND password = '' OR 'x' = 'x'
```

The condition in the WHERE-clause is now a tautology. Hence, the password check will always succeed and the user will be granted access, which is an example of an injection attack. The essence of the attack [Su & Wassermann 2006] is that the programmer intended the variable password to serve as an SQL string literal, but a specially crafted value like the one above causes it to be parsed as something else.

Injection attacks are one of the largest classes of security problems, possibly surpassing even buffer overflows[1]. SQL-constructing code is more likely to be vulnerable than not [Halfond & Orso 2005]. But injection attacks occur in many contexts other than SQL query construction in Java, as Figure 4.1 shows. SQL injections happen in any *host language* that dynamically computes SQL queries, such as PHP. Other *guest languages* are equally vulnerabl. For example, programs that build XPath or LDAP queries have the same problem. Likewise, many CGI scripts call the Unix shell with user-supplied data in an unhygienic way that allows arbitrary commands to be executed on the server. They also often send unhygienically constructed HTML to the client, enabling *cross-site scripting* (XSS) attacks that cause inappropriate content to be presented to a user, or malicious JavaScript code to be executed.

Injections can be prevented by *escaping* external input. For SQL string literals, this means that occurrences of the '-character must be doubled. Thus, the query construction above would become ... + escapeSQLStr(password) + ..., where escapeSQLStr performs the expected escaping. However, it is easy to forget to do so, and neither the compiler nor the runtime system can flag the omission of escape calls. There have been a number of proposals to detect unhygienically constructed sentences at runtime (e.g. [Su & Wassermann 2006, Halfond & Orso 2005, Huang et al. 2004]) or using static analysis (e.g. [Livshits & Lam 2005, Xie & Aiken 2006]).

A better solution, from a security perspective, is to use an *API* to build the sentence. Such an API can ensure that injections are impossible *by construction* (e.g. [McClure & Krüger 2005]). For instance, the query above could be expressed using some imaginary SQL-constructing API: SQL query = new Select(..., new Eq(new Var("password"), new Str(password))). The constructor for string literals Str then takes care of escaping. Furthermore, the type system ensures well-formedness of the sentence. But this style of programming is unattractive. It is inconvenient because it creates a cognitive gap between the programmer and the syntax provided by the guest language, which is after all a domain-specific language (DSL) designed to make certain kinds of tasks easier to express. Also, such APIs may not be available and may differ for each language. Finally, documentation and examples are expressed in terms of the concrete syntax of the DSL, not an API.

The approach that we propose in this chapter is to combine the security of using an API with the conceptual ease of string manipulation. We do this by *embedding* the syntax of the guest languages into the syntax of the host

[1] A scan of 168 SecurityFocus vulnerability reports updated in the period April 10–14, 2006 revealed at least 53 injection vulnerabilities: 24 SQL injections, 28 HTML injections, and 1 shell injection. By contrast, there were at least 30 buffer overflows and other memory-related problems.

```
$username = $_GET['username'];
$q = "SELECT * FROM users WHERE username = '" . $username . "'";
executeSQL($q);
```

SQL in PHP: SQL injection

```
String e = "/users[@name='" + name + "' and " +
                  "@password='" + password + "']";
factory.newXPath().evaluate(e, doc);
```

XPath in Java: XPath injection

```
$command = "svn cat \"file name\" -r" . $rev;
system($command);
```

Shell calls in PHP: command injection

```
String topic = getParam("topic");
String query = "SELECT body FROM comments WHERE topic = '" + topic + "'";
ResultSet results = executeQuery(query);
foreach (String body : results)
  println("<tr><td>" + body + "</td></tr>");
```

XML and SQL in Java: XSS vulnerability

Figure 4.1 Examples of unhygienic sentence construction

language, a technique pioneered by metaprogramming [Batory et al. 1998, Zook et al. 2004, Visser 2002, Bravenboer et al. 2005 (Chapter 3)]. For instance, the SQL-in-Java example above becomes

```
SQL q = <| SELECT id FROM users
           WHERE name = ${userName} AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

That is, the syntax of SQL is embedded directly into the syntax of Java expressions, using the *quotation* `<|...|>` to construct SQL code. Likewise, the *antiquotation* `${...}` embeds Java expressions into SQL to allow composition of SQL code. A preprocessor called an *assimilator* translates code written in this combined language into plain Java code that calls an API *generated* from the grammar of the guest language.

Of course, the idea of embedding a language is not new. For instance, the SQL-92 standard [ISO 1992] already defines an embedding of SQL in host languages such as C. However, these solutions have always been specific to a combination of guest and host languages, requiring considerable work to support other combinations. Examples of other combinations are SQL in a different host language, XPath, SQL, LDAP, and Shell together in the same host language, or XPath together with a scripting language in Java. The core

```
$username = $_GET['username'];
$q = <| SELECT * FROM users WHERE username = ${$username} |>;
executeSQL($q->toString());
```

SQL in PHP

```
XPath e = {- /users[@name=${name} and @password=${password}] -};
factory.newXPath().evaluate(e.toString(), doc);
```

XPath in Java

```
$command = <| svn cat "file name" -r${$rev} |>;
system($command->toString());
```

Shell calls in PHP

```
String topic = getParam("topic");
SQL query = <| SELECT body FROM comments WHERE topic = ${topic} |>;
ResultSet results = executeQuery(query.toString());
foreach (String body : results)
  println(<tr><td>${body}</td></tr>.toString());
```

XML and SQL in Java

Figure 4.2 Examples of hygienic sentence construction

contribution of this chapter is that we show that modular, scannerless parsing formalisms allow such embeddings to be created *generically*. That is, by specifying the grammar of a guest language, we can embed this language in all supported host languages; and by specifying the grammar of a new host language along with an API generator, the new host immediately allows embedding of all guest languages.

Figure 4.2 shows examples of hygienic, secure code corresponding to the unhygienic, insecure examples in Figure 4.1. Since we have grammars for guest languages such as SQL, XPath, and shell, and host languages such as Java and PHP, any combination of embeddings becomes immediately available: e.g. SQL in Java, SQL in PHP, XPath in Java, shell in PHP, LDAP in PHP, XML *and* SQL in Java, SQL *and* LDAP in PHP, and so on.

Contributions The contributions of this chapter are as follows.

- We describe a comprehensive solution to injection attacks that prevents them *by construction*. This style of programming is also more convenient than both string manipulation and high-level APIs. Preventing injection attacks by construction is a fundamentally more secure approach than *detecting* injections at runtime, as the latter is still vulnerable to denial-of-service attacks based on second-order injections.

- The approach is generic in that it can easily be adapted to new host and guest languages. Like a resourceable and retargetable compiler architecture, it takes effort $\Theta(N + M)$ rather than $\Theta(N \times M)$ to support N guest languages in M host languages. This is in contrast to previous work on injections, which addresses specific language combinations (e.g. [McClure & Krüger 2005, Cook & Rai 2005, Gould et al. 2004b, Gould et al. 2004a, Halfond & Orso 2005, Halfond et al. 2006]).

- This genericity is accomplished through a novel application of language embedding [Bravenboer & Visser 2004 (Chapter 2)], which is in turn enabled by modular, scannerless parsing. Genericity is further reached by automatically generating the underlying APIs from the context-free grammars of the guest languages. Finally, the assimilator that translates guest language fragments into API calls is fully generic and can be applied to every host language and arbitrary combinations of guest languages. As a result, *no metaprogramming* is required for adding a new guest language to the system.

- The well-formedness of constructed guest language sentences can be ensured at runtime and the well-formedness of the guest fragments is ensured statically.

- Antiquotations in metaprogramming can easily lead to ambiguities. For instance, in an SQL quotation `<| SELECT id FROM names ${where} |>`, the antiquote `${where}` can have several syntactic sorts (such as a `WHERE`- or `ORDER BY`-clause). Usually, the programmer is required to explicitly disambiguate such antiquotations. We present a novel technique for dealing with ambiguity that relieves the programmer from this burden. This technique makes the previous work in metaprogramming on embedding languages usable for programmers.

We have implemented this approach in a prototype called *StringBorg* (after the *MetaBorg* method [Bravenboer & Visser 2004 (Chapter 2)]), which is available as open source software at `http://www.stringborg.org`. StringBorg is implemented using Stratego/XT [Visser 2004], which provides the *Stratego* language for implementing program transformations, and a collection of tools (*XT*) for the development of transformation systems, based on the modular syntax definition formalism *SDF* [Visser 1997b].

4.2 APPROACH

In the previous section we saw how injection attacks are made possible by composition of sentences using string concatenation. In this section we introduce the design and implementation of the StringBorg approach.

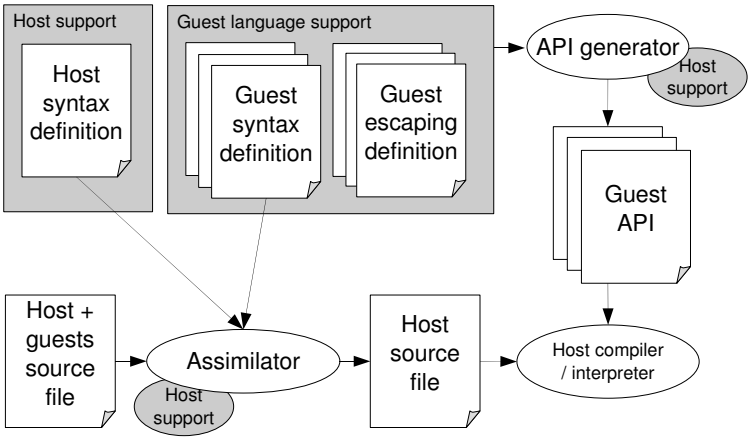

Figure 4.3 Overview of the StringBorg approach

4.2.1 *Overview*

The core problem underlying injection attacks is that a sentence (e.g. an SQL query) is parsed after its construction to a structure that does not correspond to the *intended* grammatical structure [Su & Wassermann 2006]. Unfortunately, in code using unhygienic string concatenation, the intended structure is implicit. Hence, the structure of the resulting query cannot easily be compared to the intended structure. The solution is to make the grammatical structure explicit so that any resulting sentence can be guaranteed to conform to it. StringBorg accomplishes this by parsing the guest language fragments as a pre-processing step and generating code to construct the sentences in a structured way. This ensures well-formedness of the resulting sentence and automatic escaping of antiquoted strings. StringBorg constructs sentences structurally according to the productions of the context-free grammar of the guest language.

Figure 4.3 provides an overview of StringBorg. The syntax of the guest language (e.g. SQL, Shell) is *embedded* in the syntax of the host language (e.g. Java, PHP). The programmer uses this combined syntax for writing programs. The *assimilator* uses the combined syntax definition to parse source files and transforms the embedded guest code to invocations of an API that manages the composition, escaping, and serialization of the guest code sentences. The API is produced fully automatically using an *API generator* that given a grammar of a guest language and an optional escaping definition produces an API in a specified host language. The generated API prevents injection attacks by *always* checking lexical values against the syntax for the lexical category as defined in the syntax definition. The generated API also automatically applies escaping rules to strings that are spliced into guest code using an antiquotation.

```
module SQL
exports
  context-free syntax [1]
    "SELECT" Id* "FROM" Id Where? -> Query
    "WHERE" Expr -> Where
    Expr "=" Expr -> Expr {left}
    String -> Expr
    Id -> Expr
  lexical syntax [2]
    [A-Za-z]+ -> Id
    [A-Za-z0-9\ \"\-\;] -> Char
    "'" ("''" | Char)* "'" -> String
```

Figure 4.4 Syntax definition for subset of SQL

Language Independence StringBorg combines the convenience of string concatenation and concrete syntax with the safety of an API. The main contribution of StringBorg is its genericity in the guest and host languages. That is, it is not restricted to a specific combination of a guest and host language. The language independence is not just methodological, rather the *implementation* of *(1)* support for a guest language is *independent* of the supported host languages, *(2)* the generator is easily retargetable to different host languages, and *(3)* the assimilator is generic in the guest language and almost generic in the host language. The grey boxes in Figure 4.3 indicate syntax definitions necessary to add support for a new host or guest language. The grey ellipses indicate code templates that need to be written to add support for a new host language to the assimilator and generator. These tools are written in the Stratego program transformation language [Visser 2004]. We will review the language independence of StringBorg in more detail in Section 4.2.5.

4.2.2 *Syntax Embedding and Parsing*

Throughout this chapter, we will use a small subset of SQL to illustrate StringBorg, but we stress that neither the approach nor the implementation are specific to SQL. StringBorg uses the modular syntax definition formalism SDF [Visser 1997b] to define the syntax of host and guest languages. Figure 4.4 shows the SDF grammar for our subset of SQL[2]. The SDF module defines the context-free syntax (at point [1]) of simple queries and expressions consisting of string literals, identifiers and equality comparisons. Furthermore, it defines the lexical syntax [2] of string literals. A syntax definition mainly consists of *productions* of the form $s_1 \ldots s_n$ -> s_0, declaring that a phrase of the syntactic category s_0 can be formed by concatenating phrases of categories s_1 ... s_n. Note that SDF definitions are similar to EBNF with the distinction that *(1)* productions are written left-to-right with the 'generating' nonterminal on the right-hand side, and *(2)* both lexical *and* context-free syntax are defined in the same formalism.

[2]The prototype provides a complete syntax definition for SQL.

```
module SQL-Java
imports Java SQL[3]
exports
  context-free syntax
    "<|" Query "|>" -> Expr[[Java]] {quote("SQL")} [4]
    "<|" Expr  "|>" -> Expr[[Java]] {quote("SQL")} [5]
    "${" Expr[[Java]] "}" -> Expr   {antiquote} [6]
    "${" Expr[[Java]] "}" -> String {antiquote} [7]
```

Figure 4.5 Syntax embedding of SQL in Java

```
String topic = getParam("topic");
SQL e = <| topic = ${topic} |>;
SQL q = <| SELECT body FROM comments WHERE ${e} |>
```

Figure 4.6 Composing SQL queries in Java

The use of concrete syntax for a guest language requires the embedding of the guest language in the syntax of the host language. Figure 4.5 illustrates how this is achieved in SDF. First, the syntax definition for SQL-Java imports the syntax definitions of Java and SQL [3]. Next, it adds new productions for *quotations*, i.e. for using the guest language syntax in the host language, and *antiquotations*, i.e. to escape from the guest language to the host language. In this case, the module defines quotations for SQL queries [4] and expressions [5] and antiquotations for SQL expressions [6] and strings [7]. Note that Expr is the nonterminal for SQL expressions imported from the module SQL, while Expr[[Java]] is the nonterminal for Java expressions imported from Java.

Figure 4.6 shows how quotations and antiquotation are used to compose guest sentences at runtime. The first quotation of an SQL expression uses an antiquotation to splice a Java String into the expression. At this point, the Java String will be escaped using the escaping rules for SQL. The SQL expression itself is spliced into the final SQL query. In this way, guest sentences can be composed at runtime not just by inserting string values, but also by using antiquotations for arbitrary syntactic categories of the guest language.

Parsing

Parsing source files that use a combination of a host language and various guest languages is a challenge for many parsing techniques. In our approach, the parser is generated fully automatically from the combined syntax definition, in this case the module in Figure 4.5. The problem with most mainstream parsing techniques is that grammars cannot easily be composed. Grammars for LR or LL parser generators cannot be composed in general, since LR and LL grammars are not closed under composition [Hopcroft et al. 2006]. Furthermore, lexical analyzers do not compose, since the lexical analysis of combinations of languages requires the recognition of a different lexical syntax for different locations of a source file, i.e. the lexical syntax is context-sensitive. For example, a guest language often has different keywords, operators, and

literals than the host language. Lexical analyzers are usually generated from a definition of a set of tokens, which cannot be unified into a single set of tokens for analyzing a combination of languages.

StringBorg is based on the SDF syntax definition formalism, which is implemented using *scannerless generalized LR parsing*. The parser is *scannerless*, which means that no separate lexical analyzer is used. (The lexical and context-free sections of SDF modules are desugared into a single context-free grammar.) In this way, the differences in lexical syntax between the host and guest languages, such as different keywords, operators, and literals, become a non-issue. The parser is based on the *generalized LR* (GLR) algorithm, which supports the full class of context-free grammars, which *is* closed under composition. Since SDF has a feature rich module system, we can *embed* languages in other languages in a very natural way. For an extensive discussion of the issues involved in parsing syntactic language embeddings we refer to [Bravenboer & Visser 2004 (Chapter 2), Bravenboer et al. 2006 (Chapter 5)].

Ambiguities

In applications of syntactic embeddings, ambiguities are an ubiquitous problem. For example, the antiquotation `${topic}` in Figure 4.6 can be interpreted as an SQL expression as well as an SQL string, because the same antiquotation syntax is used for both syntactic categories of the guest language, i.e. there is no way to syntactically distinguish the two antiquotations. In many applications, these ambiguities need to be resolved, either by requiring the programmer to tag the quotations and antiquotations [Batory et al. 1998, Visser 2002] with their syntactic category (e.g., `$str{topic}`), or by using a disambiguating type-checker to select the intended derivation [Zook et al. 2004, Bravenboer et al. 2005 (Chapter 3)]. Both solutions are rather unappealing for StringBorg. Tagging requires the programmer to be intimately familiar with the grammar of the guest language, which seriously affects the usability. A specially crafted disambiguating type-checker would require substantial work for every host language and would only be possible for statically typed languages.

Fortunately, for well-formedness of guest sentences in StringBorg, there is no need to know the exact syntactic category of all values. It is sufficient to verify that the value of an antiquotation actually syntactically 'fits' in the quotation into which it is spliced. StringBorg employs generalized LR parsing to *preserve* the ambiguities by letting the generalized LR parser produce all alternative derivations (i.e. a parse forest). In this way, the programmer does not have to tag quotations and antiquotations and also the approach is easy to implement for arbitrary host languages. In the next section we discuss how the ambiguities are efficiently represented at runtime by objects that can have *multiple* types, corresponding to the possible syntactic categories.

4.2.3 *API Generation*

StringBorg generates for a specific guest language an API that covers all aspects of constructing guest language strings. The API that is generated for a

```
1  public final class L {
2        private String string, Set symbols;
3        private L(String string, Set symbols) {...}
4
5        for each context-free production p : s_1 ... s_n -> s_0 :
6              public static L p(L arg_1, ..., L arg_n) {
7                for each arg_i where s_i is a lexical category:
8                      if arg_i.symbols = {string} then
9                          arg_i := escape_s_i(arg_i)
10                     if !match(dfa(s_i), arg_i) then
11                       throw exception
12
13               if ∀_{1≤i≤n}: s_i ∈ arg_i.symbols then
14                   syms := {s_0}
15                 else
16                   syms := ∅
17                pp := unparse arg_1, ..., arg_n
18               return new L(pp, syms)
19             }
20
21       for each lexical category s_i:
22             public static L literal_s_i(String s) {
23               return new L(s, {s_i})
24             }
25
26             private static L escape_s_i(String s) {
27               pp := escape s according to the escaping definition
28               return new L(pp, {s_i})
29             }
30
31       public static L ambiguity(L arg_1, ..., L arg_m) {
32          pp := ⋃_{i=1}^{m} {arg_i.string | arg_i.symbols ≠ ∅}
33          syms := ⋃_{i=1}^{m} arg_i.symbols
34          return if |pp| = 1 then new L(pp, syms) else new L("", ∅)
35       }
36
37       public static L lift(arg) {
38         if arg is of type L then
39           return arg
40         else
41           return new L(arg, {string})
42       }
43 }
```

Figure 4.7 API generation for Java-like languages

guest language is responsible for preventing injection attacks. That is, even without using the syntax embedding and assimilator, the use of the API guarantees that injection attacks cannot occur. The generated APIs have no runtime dependencies, support ambiguities, and provide support for unparsing, escaping, and checking of lexical values. The generator of the StringBorg prototype comes with back-ends for PHP and Java. Figure 4.7 outlines in pseudocode the API that is generated for Java-like languages from the syntax definition of a guest language, for example from the SQL syntax definition of Figure 4.4. The API generation for other languages such as PHP follows a substantially similar structure and is straightforward to implement. In the pseudocode *italic for each* loops are evaluated at generation-time to generate code for each production or symbol of a grammar.

For a guest language L, a class L is generated, whose instances (objects) represent sentences of L. Each object has two private fields [2]: its string representation and a set of syntactic symbols. The class has no public constructors [3]; instances of L can only be created via static factory methods. For each context-free production of the grammar, there is a corresponding static factory method [6] that constructs an L object. The formal parameters of this method correspond to the list of symbols $s_1 \ldots s_n$ in the left-hand side of the SDF production. For example, for the production `Expr "=" Expr -> Expr` the method `public static SQL` *`newEquality`*`(SQL arg1, SQL arg2)` is generated. *newEquality* is a symbolic name we use in the examples. The actual names of the factory methods are cryptographic hashes of the productions. Literals used in the left-hand side of the production, such as "=", are not passed to the factory methods.

The factory methods first apply automatic escaping to lexical values [9] (Section 4.2.3) and check if the resulting strings match the syntax of the lexical categories [10], using deterministic finite-state automata (DFA). For each lexical category, an automaton is generated from the regular grammar for this category in the syntax definition. The automata are constructed using the BRICS Automaton package [Møller 2005]. For example, this check will result in an error if a string with a newline is spliced into an SQL query, since SQL strings do not allow newlines and SQL does not provide escaping for newlines. The escaping rules are configurable in StringBorg, for example the MySQL dialect uses different escaping rules that do support newlines.

Next, the factory methods check if all the arguments of the methods have the required symbol in their set of symbols [14]. This set of symbols represents the possible syntactic categories of the L instance. For example, the factory method for an SQL `Query` checks that the last argument has the symbol `Where?`. If one of the arguments does not contain the required symbol, then the resulting L instance will have an empty set of symbols, which means that it is invalid. Note that this can only happen in antiquotations, since the syntactic correctness of a literal guest code fragment implies that all arguments will have the appropriate syntactic category in their set of symbols. If this would not be the case, then a parse error would have occurred. The construction of an L instance with an empty set of symbols does not raise an exception because of the requirement to support ambiguities (see Section 4.2.3). For the

same reason, L instances have a set of symbols, instead of just a single one.

Finally, the factory methods reconstruct the sentences of the guest language based on their arguments [17]. The generator analyzes the syntax definition of the guest language to generate the implementation of unparsing in the factory methods. The unparsers insert minimal whitespace between the symbols. Note that the actual construction of the string is hidden in the API and can be optimized by lazy unparsing to create only a single string, after the required interpretation has been determined.

Literals and Escaping

Strings that are used to construct guest sentences can originate literally from guest code templates or can be spliced in using an antiquotation. These two cases have to be handled differently, because literal strings are already escaped, whereas spliced strings are not. For literal strings, the generated API contains methods [22] to construct an L instance of symbol s for each lexical category s. Antiquoted strings are first lifted [37] to an L instance with a symbol that indicates that this is an unescaped string. Such an L instance is later used as an argument of a factory method, where it will be escaped according to the escaping rules of the lexical category. The escaping rules that need to be applied depend on the lexical category, so the strings are not immediately escaped in the lift method. In both cases, the L instances are checked by the factory methods using an DFA for the lexical category, whether they are literal or antiquoted strings. Lexical L instances are never used directly, but are only used to construct other L instances. If they are used directly, then an exception will be thrown, because this error could be a vulnerability.

The implementation of escaping [26] cannot be derived from the syntax definition, which defines the *syntax* of the escapes, but not the corresponding characters. Hand-written code for escaping strings is not an option, since preferably the escaping should be host language independent: if escaping functions have to be implemented for every particular combination of a guest and host language, then more effort than $\Theta(N + M)$ is required to support N guest languages in M host languages. To make the implementation generic, the API generator accepts an escaping definition, written in a small domain-specific language. Figure 4.8 shows two examples of escaping definitions. For SQL string literals single quotes have to be escaped using a single quote. For example, for the string `' OR 1=1` the `escapeString` method produces the safe String `''' OR 1=1'`. For XML attribute values within double quotes, several characters have to be replaced with an entity reference. The conversion for XML attribute values does not define a prefix and suffix because this embedding uses antiquotation *inside* double-quoted attribute values (string interpolation). The escape rules are optional in the configuration file, so plain conversions from host strings to lexical values can be defined as well.

Ambiguities

The API supports ambiguities in quotations and antiquotations by unifying the alternative representations to a single L instance. This is possible in String-

```
conversion string -> String {
  prefix "\'";
  suffix "\'";
  escape {
    [\'] -> "\'\'";
  }
}

conversion string -> DoubleQuotedString {
  escape {
    [\<] -> "&lt;";
    [\&] -> "&amp;";
    [\"] -> "&quot;";
  }
}
```

Figure 4.8 SQL string and XML attribute value escaping definitions

Borg because it does not matter syntactically which alternative is intended (i.e. they represent the same string). The generated method ambiguity [31] takes an arbitrary number of L arguments and composes them into a single L instance. The symbol set of the resulting L instance is the union of all the symbols of the alternatives [33]. The string of the new L instance can be the string of an arbitrary *well-formed* L instance (they are all the same), but to make this precise, we still test the cardinality [34] of the set of strings [32]. Note that some alternatives may have no symbols at all, which means that they are not well-formed. The filtering that needs to be performed in ambiguity is the reason for not throwing an exception earlier: to enable filtering of the alternatives, it is necessary to temporarily allow L instances that do not have any valid syntactic category. The toString method throws an exception when applied to such invalid L instances to guarantee that they are not used to produce a guest sentence.

Retargetable API Generation

The generator has been designed to be retargetable to different host languages by separating the implementation in a generic front-end, which produces an abstract representation of an API, and a host language specific back-end. For each host language, code templates need to be provided for generating automata, escaping, and pretty-printing. For the Java and PHP back-ends the templates amount to 420 and 400 lines of code, respectively.

To make the implementation of a new back-end as lightweight as possible, the generator back-ends produces *parse trees* of the host API, which can be unparsed to a source API without the need for a pretty-printer for the host language. In this way, only a syntax definition of the host language is required.

Programmer Protection

In injection attacks, the user of the system is the person the systems needs be protected against. Yet, if an API is present, but programmers still use strings to 'quickly' compose a sentence, then a potential enemy is the laziness of the programmer. In the case of the generative Java APIs, the constructor of the

L class is private, ensuring that it is not possible to create valid *L* instances from raw Java strings without the appropriate checks. Also, the *L* class is final to disallow subclassing. This level of safety is not available in all languages. Another issue is that access to the string-based API for evaluating guest sentences is usually still available. The detection of classical string-based use of such a library does not require complex static analysis. Arguments to the library interface should get the string value of the guest program directly from the API, for example `executeQuery(q.toString())`, where `q` is an instance of *L*.

4.2.4 *Assimilation*

The embedding of guest language syntax in the host language syntax enables parsing of sources using the combined syntax. The next step is to transform the quoted fragments to calls to the APIs for the guest languages. This transformation is called *assimilation*, as it assimilates the guest language into the host language [Bravenboer & Visser 2004 (Chapter 2)]. As an example of this transformation, Figure 4.9 shows a simple SQL quotation in Java and PHP, a sketch of the parse tree, and the result of assimilation. The sketch of the parse tree in Figure 4.9 presents the productions (see Figure 4.4) in italic using symbolic names. The argument of `quote` is an SQL fragment. The arguments of `antiquotes` are pieces of literal Java code. The example leaves out many details of the real parse tree format, which is a complete description of how the productions of the syntax definition are applied to produce the original source program, including literals, layout and comments. The result of assimilation is a one-to-one mapping from the parse tree to invocations of factory methods in the generated API. It illustrates an ambiguity and the lifting of antiquoted strings to SQL.

The assimilator operates on the full parse tree of the source program. Thus, the assimilator is layout preserving and like the API generator does not require a pretty-printer for the host language. The assimilator is fully *generic in the guest language*, i.e. there is no guest language specific code at all. This is possible because all the information about the guest language is already in the parse tree and the mapping from the syntax definition to the API is fixed, since the API is generated. This makes it easy to map the quoted guest code to the factory methods, which correspond directly to production applications.

In the actual APIs generated by StringBorg, the names of the factory methods are cryptographic hashes of the productions instead of symbolic names such as `newStringExpr`. This ensures the uniqueness of method names. Figure 4.5 shows that the productions for quotations are annotated with the name of the guest language. The assimilator uses this information to invoke methods of the appropriate factory, i.e. SQL in this example. Hence, by design the assimilator can deal with *combinations* of guest languages in a single source file.

Similar to the API generator, the assimilator is split in a front-end and a host language specific back-end. The front-end assimilates the embedded guest code to an *abstract* language that describes the factory method invocations, ambiguities, quotations, and antiquotations in a way that makes it trivial for

```
SQL e = <| topic = ${topic} |>;
```

⇒ *(parsing)*

```
LocalVarDecStm(
  ...
, VarDec(
    Id("e")
  , quote(
      Equality(
        IdExpr(Id("topic"))
      , amb([
          StringExpr(antiquote(ExprName("topic")))
        , antiquote(ExprName("topic"))
        ])
      ))))
```

⇒ *(assimilation)*

```
SQL e = SQL.newEquality(
          SQL.newIdExpr(SQL.newId(SQL.literalId("topic")))
        , SQL.ambiguity(
            SQL.newStringExpr(
              SQL.lift(topic)
            )
          , SQL.lift(topic)
          )
        )
```

```
$e = <| topic = ${$topic} |>;
```

⇒ *(parsing)*

```
Assign(
  Var("e")
, quote(
    Equality(
      IdExpr(Id("topic"))
    , amb([
        StringExpr(
          antiquote(Var("topic"))
        )
      , antiquote(Var("topic"))
      ])
    )))
```

⇒ *(assimilation)*

```
$e = SQL::newEquality(
       SQL::newIdExpr(SQL::newId(SQL::literalId("topic")))
     , SQL::ambiguity(
         SQL::newStringExpr(
           SQL::lift($topic)
         )
       , SQL::lift($topic)
       )
     )
```

Figure 4.9 Assimilation of SQL in Java and SQL in PHP

the back-end to generate host language specific code. Hence, the assimilator is easy to retarget (for Java only 48 lines of code, for PHP 44).

4.2.5 *Summary of Language Independence*

The genericity of our approach is important for making the method viable for practical use. Our goal is to make it possible to have a market of guest language embeddings that can be used in any host language *and* can be combined by users *without any metaprogramming experience*, just like programmers can already combine arbitrary libraries in a single program. StringBorg is even more generic: the implementation of a guest language is available to all supported host languages. To summarize the effort required to implement support for new guest and host languages:

- For adding a new *guest language*, no metaprogramming is required. No pretty-printer for the guest language is necessary. The assimilator is not modified to deal with a new guest language. To add support for a new guest language, one must only define its syntax, configure the escaping of literals, and define the quotation and antiquotations, all in a host language independent way.
- For adding a new *host language*, a syntax definition for the language is required. For the API generator and assimilator, only simple code templates need to be provided to their back-ends. No pretty-printer for the host language is necessary.
- For a *combination of guest languages* in a host language, no additional work is required. From the generic embeddings of the guest languages a parser can be generated fully automatically. The assimilator is designed to handle multiple embedded guest languages and the API generator does not need to be applied to combinations of guest languages: it is applied to their individual syntax definitions.

4.3 DISCUSSION

We have implemented StringBorg back-ends for the host languages Java and PHP and experimented with several (combinations of) guest languages: SQL, XPath, Shell, and XML. Our method guarantees *by construction* that injection attacks cannot occur, assuming, of course, that the API generator is correct. For evaluation, the *usability* of our method is more important: how difficult is it to rewrite an existing application to use StringBorg? We extracted use-cases of the patterns in which SQL queries and HTML responses are typically constructed from a number of web applications available from gotocode.com. Both sentences that are constructed all at once (i.e. in a single string-concatenating expression) and sentences that are constructed dynamically are fully supported. Thanks to the user-friendly support for ambiguities in StringBorg, the programmer does not need to learn disambiguation tags for quotations and antiquotations.

4.3.1 *Static versus dynamic type-checking*

The StringBorg-generated API we have presented checks *dynamically* if guest sentences are composed correctly. However, we have also implemented a Java back-end that performs these checks *statically* by using the Java type system. In this back-end every syntactic category of the guest language is represented by its own class and these are used as the return and parameter types of factory methods. Static type-checking has two major disadvantages for usability: *(1)* the programmer has to know all these syntactic categories and their mapping to types of the host language and *(2)* no ambiguities are allowed, which makes the syntax embedding more difficult to use. Obviously, the advantage is that static checking provides more static guarantees, but it is important to observe that this is not a *security* advantage. That is, both the statically and dynamically typed back-ends guarantee *statically* that an injection attack cannot occur. The dynamic or static type-checking only checks for *programming* errors, not for problems with input provided by the user. The generated APIs will never throw an 'injection attack exception'; the exceptions that can occur are either related to illegal characters in the input (e.g. the newline in SQL) or programming errors. The last category of exceptions does not depend on particular inputs, but only on execution paths, which are easier to detect using testing.

4.3.2 *Prevented classes of injection attacks*

A wide range of injection attack techniques are in use. Halfond et al. have proposed a classification of SQL injection attacks [Halfond et al. 2006]. For example, attacks can be classified by injection *mechanism* or the *intent* of the attack. We now discuss how our method deals with the classes of injection attacks identified by Halfond et al.

Injection through user input is the mechanism of using specially crafted user input to construct a query that has a different parse tree than originally intended. StringBorg prevents these attacks by checking the syntax of lexical values and automatic escaping of *all* strings.

Injection through cookies differs from injection through user input by exploiting input from cookies, which are sometimes naively assumed to be controlled by a web application. StringBorg checks and escapes *all* strings, irrespective of their origin, thus disabling this injection mechanism as well.

Injection through server variables employs yet another origin of strings, such as HTTP headers. Similar to attacks through cookies, these attacks are prevented since StringBorg escapes *all* strings.

Second-order injection attacks indirectly perform the attack by first introducing a malicious input in the system (e.g. a database), which is used later as the input of an affected query. Again, these attacks are prevented since StringBorg checks and escapes *all* strings, whether they originate directly from the user or not.

Tautology based attacks use an injection mechanism to craft a query where the condition always evaluates to true. StringBorg prevents the *mechanisms* of injection attacks from being applied, which implies that crafting tautologies is impossible.

Union query attacks are related to tautologies, but allow access to different tables than the ones originally involved in the query. Similar to tautology attacks, StringBorg prevents the mechanisms that are used.

Piggy-backed queries are malicious queries added to be executed in addition to the original query, for example by terminating an SQL query statement using a semicolon and adding a malicious one. Again, StringBorg prevents the mechanisms that are used.

Illegal query attacks are used to trigger syntax, type or logical errors. This often results in an error report that reveals information about possible exploits. StringBorg only throws an exception if an input string contains invalid characters that could not be escaped. StringBorg disables the construction of syntactically invalid queries.

Thanks to the prevention of injections, methods for triggering type and logical errors are disabled as well. The only exception is an embedding that allows conversion of input strings to table and column names (which is not the case in our embeddings). It is advisable to disallow this conversion and only allow literal table and column names. In general, allowing users to input identifiers can introduce a plenitude of options for manipulating the intended semantics of the constructed guest sentence (see also semantic injection attacks).

Inference attacks are related to illegal query attacks. They can be applied if a site is protected not to show error messages. By observing the success or failure of queries, the setup of the database can indirectly still be examined. The prevention of inference attacks does not differ from illegal query attacks.

Stored procedure attacks are a class of all known attacks applied to stored procedures. If stored procedures compose queries based on user input, then the same method for structured construction should be applied.

Alternate encoding attacks avoid detection and prevention of an attack by concealing the actual query in a different syntax or character encoding, which tricks the detection and prevention techniques into interpreting the query in a different way than the actual processor of the guest language does. In all known embeddings, StringBorg prevents encoding attacks since the encoding itself is escaped and lexical strings are checked syntactically.

However, due to the genericity of our method, it is not guaranteed that encoding attacks are prevented for *all* guest languages. For example, Java features Unicode escapes that can be used for *any* input character,

not just in string literals. If Java were used as a guest language, then Java's Unicode escapes can be used to terminate a string literal and inject code. This is currently not caught by our lexical checking, since the DFA does not unescape the Unicode escape. The fundamental reason for this problem is that the current set of definitions does not fully specify the language. The unicode escapes are basically a different surface language. The syntactic meaning of such unicode escapes is not formally defined in the syntax definition. This can be solved in several ways. *(1)* The escape sequence can be escaped. We do this in all of our embeddings, but this makes the escaping rules important for *security*, which was not the case until now. *(2)* Unescaping rules could be defined next to escape rules and applied before escaping and checking strings. *(3)* The syntax definition of the guest language can be restricted not to support Unicode escape sequences at all. *(4)* The syntax definition formalism could be extended to support lexical escape sequences.

The use of unexpected character encodings (not escape sequences) is another mechanism to hide an attack. For example, PostgreSQL was recently affected by an injection problem with multibyte character encodings [3]. This issue is host language specific and depends on the way strings are handled by the string data types that are used. This is beyond the scope of prevention techniques that check the *syntax* of queries.

Semantic injection attacks are a theoretical class of attacks that go beyond the current *syntactic* injection attacks by crafting a guest sentence that syntactically has the intended structure, but semantically has an unintended meaning. This theoretic class of attacks has not been mentioned in existing classifications due to the restricted expressiveness of the query languages that are studied. An example of such an attack could be the unintended capturing of a variable name, which is a well-known issue in metaprogramming. This attack vector becomes relevant for embeddings of languages that feature variable bindings, for example an embedding of JavaScript in Java.

StringBorg does not protect against semantic injection attacks, since the protection is based on syntax definitions. Similar to prevention of illegal query attacks, the embedding of a guest language can be restricted to only allow the conversion of strings to literals and not to identifiers. This guarantees that variable capture attacks are not possible. In general, allowing users to input identifiers can introduce a plenitude of options for manipulating the intended semantics of the constructed guest sentence. Moreover, it is usually unnatural to allow users to specify identifiers, since identifiers and variable bindings are not something the user is aware of, i.e. it is not the kind of variability in a query to leave to the user to specify.

The current formal definition of the essence of command injection attacks [Su & Wassermann 2006] is restricted to syntactic injection attacks,

[3] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2314`

which illustrates that the scope of injection attacks is still unclear. The lack of a precise definition of an injection attack makes it impossible to formally prove that injection attacks cannot occur at all. In practice, many factors are involved that can defeat theoretic soundness proofs.

4.3.3 *External queries*

Some applications use queries that are not composed and executed directly in source code, but rather stored in files or written to a database and executed later. The safety of our method is based on the fact that injection attacks are impossible by construction. To make this work, all queries executed by a system have to be constructed, in one way or another, in a structured way. This applies to external queries as well. If the external queries are constructed by the program itself, then our method can be used: there is no need to directly execute a query after constructing it. If other tools are involved in the construction of the queries (potentially used by attackers), then the same structured way of query composition has to be used in these tools.

If the external query is complete (i.e. is not composed further), then it can just be executed as is. If the query needs to be composed further, then it needs to be converted to an L instance, the representation used by the API. The query could be parsed at runtime, but this has some runtime overhead, and complicates the portability of the method to other host languages, since a parser written in this host language has to be available for every guest language. Fortunately, parsing is not necessary, since StringBorg APIs are basically wrapper classes for producing strings. Hence, it is sufficient to have a typed representation of the external query string. To support importing guest code fragments from strings, an API could provide *unsafe* methods, which convert strings to the representation used by the API. For example, a conditional expression of a query could be constructed, unparsed to a string, stored in a database, and loaded later as a conditional expression, without parsing it.

```
SQL c = <| name = $str{s} |>;
String s = c.toString();
SQL c1 = SQL.unsafeExpr(s);
SQL q = <| SELECT * FROM users WHERE ${c1}; |>;
```

Clearly, the unsafe methods could be abused by a programmer and should only be used for strings for which it is absolutely guaranteed that they have been constructed in a safe way. Currently, the StringBorg API generator does not generate unsafe methods.

4.3.4 *Generalized parsing techniques*

Syntactic limitations StringBorg relies heavily on modular syntax definition and parser generation, implemented by SDF and scannerless generalized LR parsing. This requires the syntax of the host as well as the guest language to be expressible in a context-free grammar. Unfortunately, some languages do not have such a context-free grammar. For example, SDF does not support

languages with an indentation rule (Haskell, Python). A potential solution to the problem of indentation rules is to parse these programs using an ambiguous grammar or add basic features for context-sensitive languages to the parser. The StringBorg assimilator already supports ambiguous syntax definition for the host language.

Error reporting For generalized parsing techniques to be accepted in production compilers, the quality of error messages is most important. The current error reporting of scannerless generalized LR parsing is rather Spartan; the parser only gives the line and column numbers where parsing fails. Research on error reporting of scannerless and generalized LR parsers is necessary to make generalized parsing techniques applicable in production environments.

Efficient parser composition In the StringBorg prototype, a parser needs to be generated for every combination of host and guest languages. Parser generation is too expensive to do as part of the compilation of the program that uses this combination of languages, so parsers are currently generated separately. This is not difficult to do, yet it has some impact on the 'plugin experience' of StringBorg. To improve this, we expect to present parse table plugins in future work, which enables efficient composition of scannerless generalized LR parsers. Chapter 6 on *parse table composition* is an important step in this direction.

4.3.5 *Disambiguation design space*

In applications of syntactic embeddings, ambiguities are an ubiquitous problem. For example, the antiquotation in `SELECT * FROM users ${e}` could refer to an SQL where, group-by, having, or order-by clause. The underlying problem of this ambiguity is that the same antiquotation syntax (in this case `${...}`) is used for several syntactic sorts of the guest language. Similarly, a quotation can represent multiple syntactic sorts if the same quotation syntax is used for all of them. For example, an SQL quotation `<| Name |>` could be intended as a 'select item' (e.g. `SELECT Name FROM`), but it could also be a 'row constructor' (e.g. `WHERE Name = 'Foo'`).

The most common solution for resolving ambiguities is to disambiguate explicitly by using different quotation and antiquotation symbols for distinct syntactic sorts (e.g. [Batory et al. 1998, Visser 2002]). Usually, this has to be done by the programmer by explicitly tagging the quotations and antiquotations with the intended sort of their content. This is rather unappealing, since it requires the programmer to be very familiar with the structure of the grammar of the guest language and the quotations used for its syntactic sorts. Another solution is to restrict the number of quotations and antiquotations to avoid ambiguities. For example, if the quotations and antiquotations of SQL are restricted to statements, conditional expressions, and literals, then ambiguities do not arise, because these sorts syntactically exclude each other.

In this paper, we have used runtime disambiguation of ambiguities. However, for the Java back-end with static type-checking this is not possible, since the exact type of a sentence needs to be known at compile-time. In this alternative implementation, we have used quotations and antiquotations without tags for the most common language constructs. But, explicit tagging is used for the less common constructs, or for constructs that are inherently ambiguous, such as the antiquotation of the optional where clause of a query expression. In the static back-end, this solution has been chosen mostly for pragmatic reasons, since more user-friendly solutions to the problem of ambiguities are already available, which we will discuss next. In the following example of the generation of a query with an optional order-by clause, more explicit disambiguation is necessary than usual:

```
Option<OrderByClause> e; Stm stm;
if (...)
  e = ORDER BY? <| ORDER BY User |>;
else
  e = ORDER BY? <| |>;
stm = <| SELECT * FROM users ORDER BY? ${e}; |>;
```

The first and the second quotations need to be explicitly disambiguated using the syntax ORDER BY? to indicate that the content of the quotation is an optional order-by clause. Clearly, the second quotation is ambiguous because all optional clauses can be produced by the empty string. The first quotation distinguishes the *optional* order-by clause from a plain, non-optional order-by clause. The antiquotation, which is ambiguous with all other optional clauses of a query, is disambiguated using the same ORDER BY? syntax.

Type-based disambiguation

The type system of the host language can be used to disambiguate the embedded code fragments. In the order-by example, the type of the variable `e` already indicates that the type of the right-hand side of both assignments should be optional order-by clauses. Similarly, the type of `e` indicates that the antiquotation `${e}` in the quoted query refers to an optional order-by. This method of type-based disambiguation is supported by Meta-AspectJ [Zook et al. 2004], a language for generating AspectJ programs using quotes and antiquotes, and has later been generalized [Bravenboer et al. 2005 (Chapter 3)] by employing generalized LR parsing. The main idea of the generalized approach is to preserve the ambiguities by letting the generalized LR parser produce all alternatives (i.e. a parse forest), followed by a disambiguating type-checker that operates on a *forest* of host programs. This generalized approach is guest language independent, but obviously it is not host language independent, since it requires a specially crafted host type-checker. The disambiguating type-checker we have developed for Java can immediately be applied to the Java applications of StringBorg, but using this approach has great influence on the required infrastructure per host language and is only applicable to statically typed languages.

4.4 RELATED WORK

Injection attacks have attracted a great deal of attention in recent years, and consequently there has been a substantial amount of research in developing techniques to counter them. Our approach differs from the work discussed below in either or both of two ways:

- It is generic over a large number of host and guest languages, rather than being tied to a specific combination such as SQL in Java.
- It prevents injections *by construction* rather than detecting them in existing code.

We emphasize that the present work does not obviate the need for static or dynamic analysis techniques, as they enable *existing* programs written in a traditional style to be secured. The present approach, on the other hand, provides a fundamentally safer way to develop *new* programs that need to construct guest language sentences.

4.4.1 *Explicit escaping and filtering*

The standard response to injection attacks is to tell developers to either diligently escape all user-supplied strings, or to filter out malicious inputs. Filtering can be done by rejecting known bad inputs, an approach that is unlikely to capture all bad inputs (see e.g. [Maor & Shulman 2004]); or by accepting only those inputs that match a very specific "good" pattern, e.g., that contain only certain safe characters. The latter approach has the disadvantage that it may unduly restrict users, e.g., by not allowing user names with apostrophes such as O'Brien). Both escaping and filtering suffers from the fundamental flaw that they require developers to never forget to insert the appropriate code. As with buffer overflows, relying on programmers to "get it right" every time is a recipe for disaster.

4.4.2 *APIs*

SQL DOM [McClure & Krüger 2005] makes SQL safe by hiding the SQL query construction behind an API that ensures that string literals are properly escaped by construction. However, SQL DOM goes beyond the API that we generate from the SQL grammar: it is not merely a "static" API to build SQL abstract syntax trees, but rather is *generated* from a specific database schema. Thus, it can statically ensure that all queries are well-typed with respect to that database schema. Clearly, this is a valuable property. It is important to note, however, that whether a query is ill-typed is in most cases not determined by user input. If a query produced by some code path is well-typed with respect to its schema for some input, then it is likely to be well-typed for all inputs. This is not the case for the hygiene of string concatenation: a concatenation that produces correct results for some inputs may very well fail for others, namely those that contain unescaped characters.

A somewhat similar approach is Safe Query Objects [Cook & Rai 2005], which allows queries to be defined in plain Java expressions, which are compiled using OpenJava into the necessary JDO calls. This can be viewed as embedding a convenient syntax for queries, namely Java expressions, into a host language, which happens to be Java also; the assimilation is the translation into JDO calls. Like SQL DOM, HaskellDB [Leijen & Meijer 1999], provides type safety with respect to the database schema. This API can also ensure proper escaping. These approaches have the downside of introducing a cognitive distance from the SQL language, and are specific to a particular host language and a domain of guest languages (namely query languages).

4.4.3 *LINQ*

Syntactic hygiene is an important aspect of Haskell Server Pages [Meijer & van Velzen 2001], Cω [Bierman et al. 2005] and its successor LINQ. All three provide XML literals, enabling XML output generation in a sound way. The fact that the latter two provide XML literals and an SQL-like query syntax to languages such as Visual Basic illustrates the desire to have embedded syntax for output and query generation. Similar to Safe Query Objects in OpenJava, LINQ allows host expressions to be converted implicitly to an expression tree that can be processed in arbitrary ways. LINQ is not extensible, however, in that it is not possible to plug in the syntax of other guest languages.

4.4.4 *Static analysis techniques*

JDBC Checker [Gould et al. 2004b, Gould et al. 2004a] statically checks that SQL queries built through string concatenation in Java are type-correct. It does so by building a model of the ways in which the query can be built through data-flow analysis, and then comparing that against the database schema. While this work did not address injection attacks, it should be possible to extend this approach to either discover those sites where escape functions should be called, or modifying the code to add those calls automatically. A tool that uses static analysis to find various kinds of injections is described in [Huang et al. 2004]. Xie and Aiken [Xie & Aiken 2006] developed an interprocedural static analysis algorithm for PHP and apply it to SQL injections.

Livshits and Lam [Livshits & Lam 2005] describe a general approach that allows unsafe code to be identified through a specification of code patterns for the sources of "tainted" data, consuming functions such as executeQuery ("sinks") which must not be reached by tainted data, and propagators of tainted data (e.g., string concatenation functions). However, the fact that tainted data can flow from a source to a sink is only a security problem if the data is not validated, so user inspection may be necessary to determine whether an injection is in fact possible.

All static analysis approaches require a substantial effort to apply them to a different host language, due to, e.g., the complexity in implementing the precise data flow semantics of the language.

AMNESIA [Halfond & Orso 2005] statically builds an automaton corresponding with the ways in which query strings can be constructed. Nodes in the automaton are terminals in the language, and special nodes representing external user input. At runtime, each full SQL query is matched against the automaton. In the case of an injection, the query will almost certainly not be accepted by the automaton as additional terminals are present that do not occur in the automaton. In general, any approach that attempts to check for injections in string concatenating code cannot be both sound and complete due to the undecidability of string analysis [Christensen et al. 2003], but the scenarios under which AMNESIA reports a false negative are unlikely to occur in real code.

Any approach involving static analysis takes considerable effort to port to another host language. For instance, JDBC Checker and AMNESIA use the Java String Analysis library [Christensen et al. 2003] to track string concatenations. Implementing such a library for a different language would be a non-trivial undertaking, much more difficult than writing an SDF grammar and API generator rules for the language.

WASP [Halfond et al. 2006] prevents SQL injections by keeping track for each character in a string whether it is "trusted" (e.g., originates from a constant in the source). If SQL tokens containing untrusted characters contain unsafe characters (such as a '), the query is rejected. While WASP is very effective at preventing injections, it is not trivial to port the taint-tracking implementation to other host languages.

SqlCheck prevents injection attacks by wrapping user input in special markers, e.g., (|*s*|). (A similar approach is described in [Buehrer et al. 2005].) The grammar of the guest language is then augmented by accepting the markers around certain symbols in the grammar, e.g, so that it accepts (|'*s*'|)' in SQL wherever string literals are accepted. An injection attack would then fail to parse since in, e.g., ... WHERE password = (|" OR 'x' = 'x'|) there is no production that allows an arbitrary condition inside the markers. The markers are assumed to be special strings that the client cannot produce. If the client can do so, an injection is still possible.

The weakness in SqlCheck is its assumption that the client will not be able to produce the magic marker symbols. The paper argues that by choosing the markers as sufficiently long random strings, the chance of a malicious client guessing the markers can be minimised. However, it is tenuous to assume that markers will not be leaked: for instance, web applications have an unfortunate tendency to "echo" SQL queries to the user if an error occurs. Thus, it may be quite easy for the user to trick the web application into revealing its markers.

A more fundamental problem of runtime approaches such as AMNESIA, WASP and SqlCheck is that, at runtime, they can only flag errors and prevent them from escalating into a full security compromise. But since there is no way to dynamically *recover* from the error condition, a denial-of-service attack is still possible. Consider for example the XSS-attack in Figure 4.1, where a

string containing XML injections is inserted in a database and subsequently presented to each user. In this case, if the XML injection is first detected during page generation, no user will be able to view the page anymore, receiving a server error instead. So in the case of higher-order attacks, it is not enough to detect injections: they must not be possible at all. In the hygienic approach, the malicious string will be escaped according to the XML syntax and will trigger neither an error nor a security problem.

4.4.6 *SQL-specific techniques*

The SQL-92 standard [ISO 1992] defines embeddings for various host languages, but the implementation of these embeddings is highly specific to each host language. Its equivalent of antiquotations is syntactically heavy, requiring "shared variables" between the host and guest to be declared explicitly. Queries cannot be constructed dynamically (at least not hygienically), which may explain why this approach is not widely used.

Prepared statements allow an SQL query to be constructed safely. A prepared statement contains *placeholders* (or *dynamic parameter specifications*) that are replaced hygienically by the SQL query processor; e.g., `SELECT * FROM users WHERE name = ?` has a single placeholder. Placeholders are an inconvenient antiquotation mechanism, since the programmer must ensure that the arguments in the SQL processor call match up with the placeholders, which may be tricky in dynamically generated queries with a variable number of placeholders. In addition, programmers frequently abuse prepared statements and compute the prepared statement unhygienically, rather than passing a constant.

The use of *stored procedures* prevents injection attacks, provided that the stored procedure is called in a safe way [Anley 2002a]. Unfortunately, as with prepared statements, stored procedures are sometimes called in an unhygienic way, negating the approach [Anley 2002b].

4.4.7 *MetaBorg*

The method presented in this chapter is an extended application of our previous work on concrete object syntax, or *MetaBorg* [Bravenboer & Visser 2004 (Chapter 2)], which makes the use of libraries more convenient by providing an embedded domain-specific syntax for using them. For MetaBorg, we motivated the use of the scannerless generalized LR algorithm for parsing embedded domain-specific languages and the Stratego program transformation language for assimilation of the embedded code to the host language. Compared to this earlier work, the StringBorg method presented in this chapter is more generic, thanks to the independence of the support for guest and host languages. Also, the usability of embeddings has been improved considerably by supporting ambiguities, which makes the method applicable for use by programmers without metaprogramming experience. For StringBorg, the assimilation does not translate to an existing API, but to the StringBorg

API generated by the system itself. This makes the assimilation generic for all embedded guest languages. In that sense, this chapter describes "identity assimilations": the embedded guest language and its underlying API are simply used to reconstruct the embedded guest sentences as host language strings, but in such a way that well-formedness is guaranteed. The existence of this identity mapping is what makes the system generic and easy to extend with new guest languages. The StringBorg API is generated automatically from the grammar of the guest language and covers all aspects of generating strings for the guest language.

4.5 CONCLUSION

As noted in the introduction, injection attacks are one of the largest classes of security problems, possibly surpassing even buffer overflows. Modern programming languages already defend against the latter; the work presented here protects against the former. The main advantage over previous approaches is that it makes injections impossible by construction, and that it is generic — it is not necessary to produce APIs and assimilators for each element of the cross-product of host and guest languages {Java, C#, PHP, Perl, ...} × {SQL, JDOQL, HQL, EJBQL, OQL, XML, HTML, XPath, XQuery, Shell, ...}, but only to perform a relatively small amount of work for each host and guest language.

Since different languages are good for different things, it is important to help programmers plug them together. In the case of dynamically generated sentences such as SQL queries or shell invocations, the main challenge is to ensure that this plugging happens in a grammatically well-formed way. Indeed, given the interest in technologies such as LINQ, it appears clear that there is a great deal of enthusiasm for embedding languages such as SQL and XML. However, such embeddings are generally done in a rather *ad hoc* way. It would be a good thing if the languages of the future supported modularity "out of the box". Modular syntax definition makes it possible to accomplish this goal in an efficient and principled way.

ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. The PHP grammar used by our StringBorg prototype was developed by Eric Bouwers, sponsored by the *Google Summer of Code 2006*. We thank Mark van den Brand, Giorgios Robert Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team at CWI for their work on SDF. We thank the anonymous reviewers of GPCE 2007 for providing useful feedback.