Science of Computer Programming 75 (2010) 473–495

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Preventing injection attacks with syntax embeddings[☆]

Martin Bravenboer[a,*], Eelco Dolstra[b], Eelco Visser[b]

[a] *Department of Computer Science, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA 01003, USA*

[b] *Department of Software Technology, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands*

ARTICLE INFO

Article history:
Received 7 March 2008
Received in revised form 18 May 2009
Accepted 21 May 2009
Available online 31 May 2009

Keywords:
Injection attacks
Security
Syntax embedding
Program generation
Program transformation
Concrete object syntax

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 context-free host and guest languages.

1. Introduction

In this paper we propose using *syntax embedding* to prevent injection vulnerabilities in a language-independent way. Injections form a very common class of security vulnerabilities [22]. 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 very common 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'
```

☆ An earlier version appeared in GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering.
* Corresponding author.
E-mail addresses: martin.bravenboer@acm.org (M. Bravenboer), e.dolstra@tudelft.nl (E. Dolstra), visser@acm.org (E. Visser).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.05.004

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

SQL in PHP: SQL injection vulnerability

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

XPath in Java: XPath injection vulnerability

```
$searchfilter = "(cn=" . $username . ")";
$search = ldap_search($connection, $directory, $searchfilter);
```

LDAP in PHP: LDAP injection vulnerability

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

Shell calls in PHP: command injection vulnerability

```
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

Fig. 1. Examples of unhygienic sentence construction.

The condition in the WHERE-clause is now a tautology. Hence, the password check will always succeed and the user will be granted access. This is an example of an *injection attack*. The essence of the attack [35] 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 constitute one of the largest classes of security problems.[1] SQL-constructing code is more likely to be vulnerable than not [20]. But injection attacks occur in many contexts other than SQL query construction in Java, as Fig. 1 shows. SQL injections happen in any *host language* that dynamically computes SQL queries, such as PHP. Other *guest languages* are equally vulnerable. 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. Web applications often include input from a user in HTML web pages without properly checking the input. If the unhygienic input from one user is also viewed by other users (e.g. a message board), then a malicious user can modify web pages to show inappropriate content, access private information, or execute malicious scripting code. This is referred to as a *cross-site scripting* (XSS) attack.

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. For the password `' OR 'x' = 'x`, the constructed SQL query would now be:

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

The SQL processor will now correctly return all users with the specified name and the password `' OR 'x' = 'x`. However, it is easy to forget to escape all external input properly, 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. [20,24,35]) or using static analysis (e.g. [28,40]).

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. [30]). 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 could ensure well-formedness of the sentence, as opposed to the lack of a guarantee of syntax correctness of sentences embedded in string literals. 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 of an API.

[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. For this period, there were at least 30 buffer overflows reports and other memory-related problems.

```
$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

```
$searchfilter = (| (cn=$($username)) |);
$search = ldap_search($connection, $directory, $searchfilter->toString());
```

LDAP in PHP

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

Fig. 2. Examples of hygienic sentence construction.

The approach that we propose in this paper is to combine the security of using an API with the conceptual ease of string manipulation for constructing sentences. We do this by *embedding* the syntax of the guest languages into the syntax of the host language, a technique pioneered in the field of metaprogramming [3,10,38,41]. 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. This API guarantees that any external input will properly escaped.

Of course, the idea of embedding a language is not new. For instance, the SQL-92 standard [25] 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 (e.g. PHP or C#), XPath, SQL, LDAP, and Shell together in the same host language, or XPath together with a scripting language in Java. The core contribution of this paper 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.

Fig. 2 shows examples of hygienic, secure code corresponding to the unhygienic, insecure examples in Fig. 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 paper 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 (see Section 3.2).
- The approach is generic in that it can easily be adapted to new host and guest languages. Like a retargetable and language-independent 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. [15, 18–21,30]).

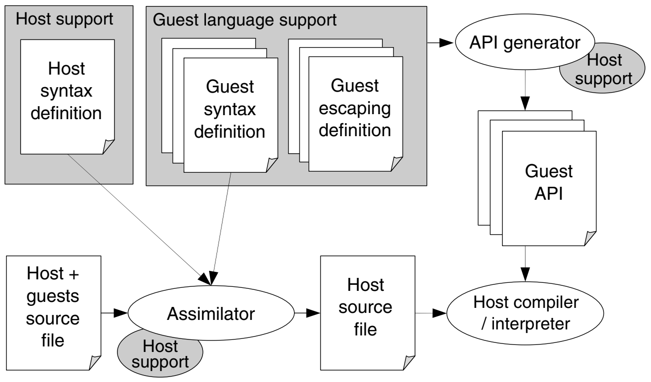

Fig. 3. Overview of the StringBorg approach.

- This genericity is accomplished through a novel application of language embedding [11], 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 sentences of the guest language is ensured at runtime, while 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 of 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 [11]), which is available as open source software at http://www.stringborg.org/. StringBorg is implemented using Stratego/XT [8], 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* [37].

Organization . In Section 2 we present our approach for preventing injection attacks. We extensively discuss design and safety issues in Section 3. Section 4 provides suggestions for future work and Section 5 compares our approach to related work in fighting injection attacks. Finally, we conclude the paper in Section 6.

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.

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 [35]. 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.

Fig. 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 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. In this way, it is impossible for a malicious user to provide an input that is intended by the programmer to be of a certain lexical category, but is later parsed differently. 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
```

Fig. 4. Syntax definition for subset of SQL.

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:

(1) The *implementation* of support for a guest language is *independent* of the supported host languages, e.g. the implementation of support for SQL can be used for Java as well as PHP.
(2) The API generator is easily *retargetable* to different host languages, e.g. the implementations of the Java and PHP back-ends contain fewer than 50 lines of code.
(3) The assimilator is *generic* in the guest language and almost generic in the host language. That is, minimal implementation effort is required for adding support for a new host language, and the assimilator does not need to be modified for adding a new guest language.

The grey boxes in Fig. 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 [8]. We will review the language independence of StringBorg in more detail in Section 2.5.

2.2. Syntax embedding and parsing

Throughout this paper, 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 [37] to define the syntax of host and guest languages. Fig. 4 shows the SDF grammar for our subset of SQL. 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. Combining the definition of lexical and context-free syntax is important for parsing syntax embeddings, as we will discuss in Section 2.2.1.

The use of concrete syntax for a guest language in a host language requires the embedding of the syntax of the guest language in the syntax of the host language. Such an embedding extends the grammar of the host language with production rules that define the syntax of *quotations*, i.e. for using the guest language syntax in the host language. Also, the grammar of the guest language is extended with productions for *antiquotations*, i.e. to escape from the guest language to the host language. In our prototype, the syntax of an embedding is defined by defining production rules for a selection of syntactic categories of the guest language that need to be quoted and antiquoted. Fig. 5 illustrates how this is achieved in SDF.[2] To make the embedding independent of the host language, the SDF module defining the embedding is *parameterized* 3 with the syntactic category for host language expressions (`E`). The module defines quotations for SQL queries 4, where clauses 5, optional where clauses 6, and expressions 7 (the `stringborg(...)` annotations will be explained later). Note that `Expr` is the nonterminal for SQL expressions imported from the module `SQL`, not a host language expression. These production rules respectively introduce support for the following example quotations:

```
<| SELECT body FROM comments |>
<| WHERE topic = 42 |>
<| |>
<| topic = 42 |>
```

[2] Note that such a module could also be generated from a more concise configuration file that just specifies the syntactic categories, quotation, and antiquotation syntax.

```
module EmbeddedSQL[E] 3
imports SQL
exports
  context-free syntax
    "<|" Query  "|>" -> E {stringborg(quote("SQL"))} 4
    "<|" Where  "|>" -> E {stringborg(quote("SQL"))} 5
    "<|" Where? "|>" -> E {stringborg(quote("SQL"))} 6
    "<|" Expr   "|>" -> E {stringborg(quote("SQL"))} 7

    "${" E "}" -> Where  {stringborg(antiquote)} 8
    "${" E "}" -> Where? {stringborg(antiquote)} 9
    "${" E "}" -> Expr   {stringborg(antiquote)} 10
    "${" E "}" -> String {stringborg(antiquote)} 11
```

Fig. 5. Syntax embedding of SQL.

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

Fig. 6. Composing an SQL query.

```
module JavaMix[Ctx]    12
imports Java            13
  [ CompilationUnit => CompilationUnit⟦Ctx⟧   14
    TypeDec         => TypeDec⟦Ctx⟧
    ...
    FieldAccess     => FieldAccess⟦Ctx⟧
    MethodSpec      => MethodSpec⟦Ctx⟧
    Expr            => Expr⟦Ctx⟧
  ]
```

Fig. 7. SDF grammar mixin for Java.

Furthermore, the module of Fig. 5 defines antiquotations for SQL where clauses [8], optional where clauses [9], expressions [10] and strings [11]. The first two production rules introduce support for an antiquotation immediately after the from clause:

`<| SELECT body FROM comments ${`*`some host expression`*`} |>`

The last two production rules introduce support for antiquotation in an SQL expression, where the antiquotation represents either an SQL string or a complete SQL expression, as illustrated by the example program fragment of Fig. 6 where an SQL query is composed using multiple Java statements. This example shows how quotations and antiquotations are used to compose guest sentences at runtime. The first quotation of an SQL expression uses an antiquotation to splice the Java String topic into the expression. At this point, the Java String will be escaped using the escaping rules for SQL. The SQL expression e is spliced into the final SQL query q. 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.

Production rules in SDF can be annotated with arbitrary terms, comparable to the annotation facilities of Java and C#. The production rules of the embedding defined in Fig. 5 use annotations to provide the StringBorg assimilator the necessary information on the special meaning of the productions for quotations and antiquotations. After parsing, the StringBorg assimilator traverses the parse tree, which is a mixture of nodes corresponding to production applications of the host language, guest languages, quotations, and antiquotations. The nodes of the parse tree refer to their corresponding productions in the syntax definition, which provides the StringBorg assimilator with the information necessary to invoke the generated API. At production applications annotated with stringborg(quote(...)) the StringBorg assimilator starts transforming the embedded SQL code to Java code that corresponds to the construction of this SQL query. At production applications annotated with the stringborg(antiquote) annotation the StringBorg assimilator stops transforming the parse tree, since such a production application indicates a transition back to the host language.

Finally, we need to combine the syntax definition for embedded SQL with the syntax definition of a host language, for example Java. A naive method would be to just create an SDF module that imports the two syntax definitions, for example EmbeddedSQL and Java. However, this ignores that the two grammars could use the same name for different syntactic categories (e.g. a Java Expr and an SQL Expr). To solve this, we use the Java syntax definition as a *grammar mixin* [9]. In the context of object-oriented programming, mixins are abstract subclasses that are parameterized in their superclass [6]. A grammar mixin is a syntax definition that is parameterized with the *context* in which all the syntactic categories of the syntax definition should be defined. The grammar mixin can be used with different contexts, similar to using a mixin with different superclasses. A grammar mixin in SDF is a generated SDF module that is parameterized with a context symbol. Fig. 7 shows the grammar mixin for Java, with a context parameter Ctx[12]. The generated module imports the Java syntax

```
module Java-SQL
imports
  JavaMix[ Java ]
  EmbeddedSQL[ Expr⟦Java⟧ ] 15
```

Fig. 8. Syntax embedding of SQL in Java.

definition[13] and renames[14] all the syntactic categories of the Java syntax definition to new syntactic categories that are specific to the given context Ctx. For example, by importing JavaMix with the context symbol Java the nonterminal Expr in the Java syntax definition will be named Expr[Java]. This is a parameterized syntactic category, similar to parameterized types in other programming languages.

By importing JavaMix more than once with different contexts, multiple instances of the syntax definition of Java will be available in those different contexts. Those instances form a family of related syntax definitions. Every instance can be customized or extended for the specific context in which it is defined [9]. In StringBorg, we use grammar mixins just to keep the different languages that are involved separate, comparable to a namespace mechanism (as available in many programming languages). However, a namespace mechanism, which would simpler than grammar mixins, is by itself not sufficient. For example, we want to be able to use Java itself as a guest language in Java. The two languages have distinct roles in this composition, but they have the same nonterminals, even if their names are fully qualified. Using a simple namespace mechanism it is not possible to distinguish the two languages and their roles. Therefore, it is not possible to specify that Java should be embedded as a guest language in Java as host a language, and at the same time not embed Java in Java as a guest language. Indeed, the essence of the grammar mixin approach is that the same syntax definition can be imported multiple times, and be can extended or modified for every different context.

Fig. 8 shows the SDF module that combines embedded SQL with Java. The parse table generated from this module will be used to parse a Java source file that uses embedded SQL. The module Java-SQL imports the grammar mixin JavaMix with the context parameter Java. This is just a symbolic name: we could have chosen an arbitrary symbol. The embedding of SQL, i.e. the module EmbeddedSQL, is parameterized with the expression category of the host language. Therefore this module is imported with the parameter Expr[Java]. Ideally, this composition module should not be written by hand. It can be generated by the StringBorg assimilator, based on a selection of guest languages by the user. In future work, we plan to implement this efficiently using parse table composition (see Section 4.3.1).

2.2.1. *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 our example of SQL embedded in Java this is the module of Fig. 8. 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 [23]. 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 in 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 separate 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 parsing context acts naturally as lexical state.

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 previous work [9,11].

2.2.2. *Ambiguities*

In applications of syntactic embeddings, ambiguities are a ubiquitous problem. For example, the antiquotation ${topic} in Fig. 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 [3,38] with their syntactic category (e.g., $str{topic}), or by using a disambiguating type-checker to select the intended derivation [10,41]. Both solutions are rather unappealing for StringBorg. First, tagging quotations and antiquotations with their syntactic category requires the programmer to be intimately familiar with the grammar of the guest languages, which seriously affects the usability. It might be acceptable in metaprogramming to assume that the metaprogrammer is familiar with the syntactic categories of a language [3,38], but for normal programming tasks this would make the approach too inaccessible. Second, a specially crafted disambiguating type-checker would require substantial work for every host language and would only be possible for statically typed languages. Also, the programmer still has to be familiar with the types of abstract syntax tree expressions in this approach.

Fortunately, for guaranteeing well-formedness of guest sentences in StringBorg, there is no need to know the exact syntactic category of all values. This means that StringBorg can allow ambiguous quotations and antiquotations and does not need to resolve these ambiguities immediately.

- For an ambiguous quotation, all alternative syntactic categories of the guest language fragment are preserved.
- For an unambiguous antiquotation, i.e. an antiquotation that expects a value of one specific syntactic category to be spliced into guest code, the actual runtime value will either be unambiguous or ambiguous. In either case, well-formedness of the constructed guest sentence is guaranteed if the value has *at least* the expected syntactic category. It does not matter that the spliced fragment also has alternative parse trees. In other words, it is sufficient to verify that the value of an antiquotation actually syntactically 'fits' in the quotation into which it is spliced.
- An ambiguous antiquotation does not actually exist as such in a parse tree. Instead, the guest code surrounding the antiquotation is ambiguous. Again, the actual runtime value can be ambiguous or unambiguous. In both cases, well-formedness is guaranteed for every alternative parse tree if the value has *at least* the expected syntactic category. Usually, there are also alternatives that turn out to be invalid. Those are eliminated, as will be explained later.

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.

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 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. Fig. 9 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 Fig. 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. Fig. 10 shows a fragment of the generated Java API, focusing on the handling of string literals and user input.

For a guest language *L*, a class *L* is generated, whose instances (objects) represent sentences of *L*. Each object has two private fields (line [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 … 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]. Lexical values are represented by an *L* instance with the special symbol inputstring (see Section 2.3.1). An *L* instance representing a lexical value always has a singleton set of symbols: {inputstring}. For this reason, the factory method for String → Expr in Fig. 10 just checks if the set of symbols *contains* the symbol INPUTSTRING.

After escaping, the factory method checks 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 [32]. 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 standard 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 generated API constructs an *L* instance with an empty set of symbols [16], as opposed to raising an exception, because of the requirement to support ambiguities (see Section 2.3.2). For the same reason, *L* instances have a set of symbols, instead

```
public final class L {
    private String sentence, Set symbols;
    private L(String sentence, Set symbols) {...}

    for each context-free production p : $s_1 \ldots s_n$ -> $s_0$ :
        public static L p(L $arg_1, \ldots,$ L $arg_n$) {
          for each $arg_i$ where $s_i$ is a lexical category:
              if $arg_i$.symbols = {inputstring} then
                  $arg_i$ := escape_$s_i$($arg_i$)
              if !match(dfa($s_i$), $arg_i$) then
                  throw exception

          if $\forall_{1 \leq i \leq n}$: $s_i \in arg_i$.symbols then
              syms := $\{s_0\}$
          else
              syms := $\emptyset$
          pp := unparse $arg_1, \ldots, arg_n$
          return new L(pp, syms)
        }

    for each lexical category $s_i$:
        public static L literal_$s_i$(String s) {
          return new L(s, {$s_i$})
        }

        private static L escape_$s_i$(String s) {
          pp := escape s according to the escaping definition
          return new L(pp, {$s_i$})
        }

    public static L ambiguity(L $arg_1, \ldots,$ L $arg_m$) {
      pp := $\bigcup_{i=1}^{m}$ {$arg_i$.sentence | $arg_i$.symbols $\neq \emptyset$}
      syms := $\bigcup_{i=1}^{m}$ $arg_i$.symbols
      return if |pp| = 1 then new L(pp, syms) else new L("", $\emptyset$)
    }

    public static L lift(arg) {
      if arg is of type L then
          return arg
      else
          return new L(arg, {inputstring})
    }
}
```

Fig. 9. API generation for Java-like languages.

of just a single one. An alternative solution would be to use null to represent invalid SQL instances and handle the possible null arguments in the factory methods.

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.

2.3.1. *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

```
public final class SQL {
  private String _sentence;
  private java.util.HashSet _symbols;
  private static final Object INPUTSTRING = new Object();

  private SQL(String sentence) {
    _sentence = sentence;
    _symbols = new java.util.HashSet();
  }

  public boolean isValid() {
    return !_symbols.isEmpty()
  }

  public String toString() {
    if(isValid())
      return _sentence;
    else
      throw new IllegalStateException(...);
  }

  // factory method for production String -> Expr
  public static SQL new_<hash of production>(SQL  arg1) {
      if(arg1._symbols.contains(INPUTSTRING))
        arg1 = escape_String(arg1);

      // check if arg1 is an SQL String
      boolean valid = arg1._symbols.contains("String");
      if(valid) {
        match_String(arg1);
      }

      SQL result;
      if(valid) {
        StringBuilder builder = new StringBuilder();
        builder.append(arg1._sentence);
        result = new SQL(builder.toString());
        result._symbols.add("Expr");
      } else {
        result = new SQL("");
      }

      return result;
  }

  ... // more factory methods

  public static SQL literal_String(String s) { ... }
  private static SQL escape_String(SQL arg) { ... }
  private static void match_String(SQL arg) { ... }

  public static SQL ambiguity(SQL ... alts) { ... }

  public static SQL lift(SQL value) { return value; }
  public static SQL lift(String s) {
    SQL result = new SQL(s);
    result._symbols.add(INPUTSTRING);
    return result;
  }
}
```

Fig. 10. Java API generated from SQL syntax definition.

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 inputstring (`INPUTSTRING` in Fig. 10) that indicates that this is an unescaped string. This method can be overloaded if the target language supports this (see Fig. 10).

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 a DFA for the lexical category, whether they are literal or antiquoted strings. Lexical *L* instances are never used directly, but are only

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

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

Fig. 11. SQL string and XML attribute value escaping definitions.

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. Fig. 11 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 conversions from host strings to lexical values without escaping special characters can be defined as well. This is not a security risk, because all strings are checked *after* escaping.

2.3.2. *Ambiguities*

The API supports ambiguities in quotations and antiquotations by unifying the alternative representations to a single *L* instance. This is possible in StringBorg 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 new *L* instance as follows. 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 [16], which means that they are not well-formed. These alternatives are not considered. If there are no well-formed alternatives at all, then the new *L* instance returned by the `ambiguity` method is not well-formed either. This filtering that needs to be performed in `ambiguity` is the reason for not throwing an exception if one of the arguments of a factory method is invalid [16]. 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 invalid *L* instances to guarantee that they are not used to produce a guest sentence.

2.3.3. *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, a generator back-end produces a *parse tree* of a host API (e.g. a parse tree of a Java source file). The parse tree contains every single character of the source code it corresponds to, thus the generated host API can be unparsed to source code without the need for a pretty-printer for the host language. In this way, only a syntax definition of the host language is required. Note that this is unrelated to the pretty-printer for the *guest* language, which is necessary, but generated, and is implemented by the API.

2.3.4. *Programmer protection*

In injection attacks, the user of the system is the person the system 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.

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

$\Rightarrow$ *(parsing)*

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

$\Rightarrow$ *(assimilation)*

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

Fig. 12. Assimilation of SQL in Java.

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 requires only straightforward 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*.

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 [11]. As an example of this transformation, Figs. 12 and 13 show simple SQL quotations in Java and PHP, a sketch of the parse tree, and the result of assimilation. The sketches of the parse trees present the production rules (see Fig. 4) in italics using symbolic names. The actual parse tree contains the full production, including its annotations. The arguments of quotes are SQL fragments. 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 for the host language 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. Fig. 8 shows that the productions for quotations are annotated with the name of the guest language (e.g. stringborg(quote("SQL")). 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 into 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 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).

As an illustration of the ease of retargeting the assimilator, Fig. 14 shows the complete PHP back-end of the StringBorg assimilator. The assimilator is implemented in the Stratego program transformation language [8]. The first declaration, assim-php, is the main *strategy* for traversing the abstract syntax tree, which at this point is a mixture of the host language and the

```
$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)
       )
     )
```

Fig. 13. Assimilation of SQL in PHP.

abstract intermediate language. The second declaration, assim-php-quote handles quotations, and the last four declarations, abstract-assim-to-php, are the assimilation rules. The assimilation rules translate the abstract intermediate language that is produced by the front-end of the assimilator to PHP. The intermediate language only has an abstract syntax, therefore the left-hand sides of the assimilation rules are abstract syntax patterns. The right-hand sides of the assimilation rules use quotations with the concrete syntax of PHP. The main traversal strategy assim-php first tries to apply the strategy assim-php-quote for handling quotations (left of <+). The strategy assim-php-quote mainly sets a scoped dynamic variable FactoryName to the name of the language of this quotation, as specified in the annotation stringborg(quote(L)) of a production rule. After that, it applies the main traversal strategy, assim-php, recursively using the rec parameter. If the current term is not a quotation, then the main strategy assim-php tries to apply the assimilation rules (right of the <+). The assimilation rules are simple mappings from the abstract intermediate language to PHP. They use the scoped dynamic variable FactoryName to determine the factory class of the current quotation. The main traversal strategy applies itself recursively after applying the assimilation rules (using all(assim-php)). The implementation of the PHP back-end illustrates that thanks to the intermediate language, not every back-end needs to implement the problem of enabling assimilation in quotations and disabling assimilation in antiquotations. The front-end already takes care of this, resulting in a back-end that only consists of trivial mappings from the intermediate language to the host language.

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 than libraries: 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 needs to be implemented: the API generator derives a simple pretty-printer from the syntax definition of the guest language. The assimilator is not modified or recompiled 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.

```
assim-php =
  assim-php-quote(assim-php)
  <+ try(abstract-assim-to-php)
     ; all(assim-php)

assim-php-quote(rec) =
  ?Quote(<id>, attrs)
  ; {| FactoryName :
      where(
        <contains-stringborg(?quote(l))> attrs
        ; rules(FactoryName := l)
      )
    ; rec
    |}

abstract-assim-to-php :
  AntiQuote(e, attrs) -> php:expr |[ y::lift(e) ]|
  where
    y := <FactoryName>

abstract-assim-to-php :
  Literal(x', s)  -> php:expr |[ y::x(e) ]|
  where
    y := <FactoryName>
    ; x := <conc-strings> ("literal_", x')
    ; e := <string-to-php-string> s

abstract-assim-to-php :
  New(x', es) -> php:expr |[ y::x(param*) ]|
  where
    y := <FactoryName>
    ; x := <conc-strings> ("new_", x')
    ; param* := <separate-by(|',')> es

abstract-assim-to-php :
  Amb(es) ->  php:expr |[ y::newAmbiguity(param*) ]|
  where
    y := <FactoryName>
    ; param* := <separate-by(|',')> es
```

Fig. 14. Stratego assimilation rules for the PHP host language.

3. Discussion

We have implemented StringBorg back-ends for the host languages Java and PHP and experimented with several (combinations of) guest languages: SQL, LDAP, XPath, Shell, and XML. Our method guarantees *by construction* that injection attacks cannot occur. For this reason, the usual evaluation of injection attack *detection* mechanism by determining the number of detected injection attacks does not make sense for StringBorg. Widespread adoption of our approach would make errors in the implementation of the system an interesting attack vector. This could potentially be any possible mistake, as is illustrated by the great variety of security issues in compilers, operating system kernels, and virtual machines.

For evaluation, the *usability* of our method is more important. Our method provides useful guarantees, but is it reasonable to expect programmers to write programs in this way? How difficult is it to rewrite an existing application to use StringBorg? Can the quotations and antiquotations of StringBorg replace most current patterns in which guest sentences are constructed without too much refactoring of the host program? To evaluate this, 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.

In this section we discuss some design issues that affect usability and (static) safety in some more detail. Another important issue is the introduction of a new parsing technique in the compiler used by end-programmers. We discuss the future work that is required in this direction in Section 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. This back-end is similar to the translation scheme of our earlier work on metaprogramming with concrete object syntax applied to Java [10].

Unfortunately, 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. Using a disambiguation type-checker [10] would allow ambiguous quotations and antiquotations, but as mentioned earlier this requires considerable effort for a specific host language.

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. Unlike approaches that detect injection attacks at runtime, 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.

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 [22]. 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). Thus, if users are allowed to input syntactic categories like identifiers, then it is still possible to trigger *semantically* illegal queries, but not syntactically illegal ones.

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 syntax and escaping 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 could be restricted to not 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 the 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 [35] is restricted to syntactic injection attacks, 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.

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: we do not require the query to be executed immediately after construction, i.e. the query can safely be stored and executed later. 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.

```
String name = getParam("name");
SQL condition = <| name = ${name} |>;
String externalQuery = condition.toString();
SQL condition1 = SQL.unsafeExpr(externalQuery);
SQL query = <| SELECT * FROM users WHERE ${condition1}; |>;
```

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.

3.4. Disambiguation design space

In applications of syntactic embeddings, ambiguities are a 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 categories of the guest language. Similarly,

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

a quotation can represent multiple syntactic categories 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 categories (e.g. [3,38]). Usually, this has to be done by the programmer by explicitly tagging the quotations and antiquotations with the intended syntactic category 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 categories. 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 categories 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 of the Java program. 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 (i.e. to reduce implementation effort), since more user-friendly solutions to the problem of ambiguities are already available, which we will discuss in Section 3.4.1. 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;
if (...)
  e = ORDER BY? <| ORDER BY User |>;
else
  e = ORDER BY? <| |>;
Stm stm = <| SELECT * FROM users ORDER BY? ${e}; |>;
```

The first and the second quotation 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.

3.4.1. 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 [41], a language for generating AspectJ programs using quotes and antiquotes, and has later been generalized [10] 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. Future work

4.1. Implementation optimization

The API that is currently generated by our prototype can be optimized in several ways. First, pretty-printing of the arguments of a factory method could be delayed until the expression has been disambiguated. Also, for Java, a single StringBuilder could be used to compose the final guest sentence. Currently, every factory method allocates a new StringBuilder. Fortunately, those details can be hidden behind the interface of the API. Second, the representation of the DFA is currently rather naive: every state of the DFA is a method in a local class declaration. An optimized encoding of the DFA should be introduced. Third, it might be beneficial to put more trust in the programmer. Currently, literal lexical arguments (i.e. occurring literally in a quotation) are matched against the DFA, just like user input. This avoids having a leak in the API that allows the programmer to create an object, possibly based on user input, that has not been checked. This protection might be overly enthusiastic. Literal lexical arguments have been parsed, which implies that they match the DFA.

4.2. Grammar engineering

To make our solution for preventing injection attacks work in practice, grammars need to become a software artifact with solid engineering practices and supporting tools. For example, reliable methods are necessary to migrate a grammar from one grammar formalism to another. Unfortunately, tool support for semi-automatic grammar migration is currently ad hoc. In previous work [5] we presented grammar engineering support for the reliable migration of precedence rules, which was one of the major obstacles we encountered in our applications of syntax embedding. However, more work is necessary in this direction. For example, there should be tool support for deriving an encoding of precedence rules in different expression nonterminals from a set of precedence rules. Also, a complete semi-automatic migration tool from a lexical analyzer specification (e.g. flex) and grammar (e.g. yacc) to an SDF syntax definition would be most useful.

Also, grammar engineering support is necessary for testing, profiling, and analyzing grammars. The advantage of LR-like parser generators is that the developer is forced to develop an unambiguous grammar, i.e. the existence of an LR grammar is a proof that the grammar is unambiguous. In general, it is not possible to prove this for arbitrary context-free grammars. Hence, grammars developed for a generalized LR parser generator can always turn out to be ambiguous in some obscure, unexpected cases. To minimise this risk, it is important to develop grammar analysis and testing tools.

4.3. Generalized parsing techniques

Our application of modular syntax definition and scannerless generalized parsing is a strong motivation for continued (or renewed) research into fully automatic parser generation. For our solution to preventing injection attacks to become mainstream, it is crucial for the generated parsers to feature production quality, language-specific error reporting, error recovery, and acceptable performance. Surprisingly, deriving a production quality parser from a declarative, possibly ambiguous, syntax definition is still one of the open problems in research on parsing techniques. Perhaps, one of the reasons is that the main application of parser generators has been compilers for single programming languages that are designed to be parsed using algorithms that can easily be implemented by hand. Considering the user-base of compilers, it is worth the effort to spend considerable time on handcrafting such a compiler. For these use cases, there is no strong argument for using parser generators, in particular because the generated parsers usually do not provide a better user experience.[4]

However, crafting a parser for specific combinations of a host language and its extensions does not scale to the full vision of applications such as StringBorg. As a result, there is a strong motivation for renewed research into bringing the user experience of fully automatic generated parsers closer, or possibly beyond, the user experience of handcrafted parsers.

4.3.1. Efficient parser composition

Ideally, the user of a compiler should be able to select a series of guest language extensions in the invocation of a compiler. Therefore, the implementation of the embedding of a guest language should be deployed as a *plugin* to the compiler of the host language. Unfortunately, extensions implemented using current extensible compilers [16,33,39] cannot be deployed as true plugins. Current extensible compilers focus on *source-level extensibility*, which requires the user of the compiler to rebuild the compiler for every combination of extensions. This might be acceptable for experiments with single language extensions, but it is not for a practical system where a non-expert user must be able to combine extensions efficiently.

In the current StringBorg prototype implementation, a parser needs to be generated for every combination of host and guest languages. A naive implementation of a plugin mechanism could just generate a parser when the compiler is invoked by the user. Unfortunately, parser generation is too expensive to do as part of the compilation of the program that uses this combination of languages. Generating a parser separately is not difficult to do, yet it has some impact on the 'plugin experience' of StringBorg.

To improve this, we have developed an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components* [7]. Parse table components can be composed efficiently just before parsing by a minimal reconstruction of the parse table. Based on this algorithm, we can now develop a system where third parties can deploy guest language implementations as binary components. For StringBorg, this means that a parse table component is generated from the embedding module; see Fig. 5 for our running example. The module of Fig. 8 is no longer necessary. Instead, the parser composes the parse table component of the Java host language with the desired guest languages just before parsing. In [7] we have already evaluated the performance of parse table composition for this scenario, with promising results, but we have not integrated this yet into our StringBorg prototype.

An alternative approach is to deploy the parse tables for guest languages, but not combine them with the parse table of the host language. Rather, the parser of the host language could switch to a parser for the guest language when it encounters a specific token. This method is easier to implement than parse table composition, but imposes restrictions on the tags: a begin tag should unambiguously identify the switch to a parser of a specific guest language. Most likely, this will lead to reserving keywords and introducing more exotic tags.

4.3.2. 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

[4] Surprisingly, PHP users accept the very poor error reports of the generated PHP parser. The parser reports errors in terms of symbolic names for tokens, e.g. "parse error: unexpected T_ECHO in foo.php on line 2", where figuring out the definition of T_ECHO is left as an exercise to the user.

languages with an indentation rule (such as Haskell or 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. Ambiguous grammars are already supported by StringBorg.

4.3.3. Error reporting and recovery

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. Due to the forking LR parsers of the generalized LR algorithm, it is not immediately obvious how existing techniques for error reporting and recovery of LR parsers can be applied.

Scannerless parsing introduces another challenge for error reporting, since errors are usually reported in terms of tokens in conventional scanner-based parsers. Scannerless parsers have no notion of tokens, i.e. they operate on individual characters and nonterminals. Currently, the implementation of SGLR reports errors in terms of unexpected characters, without any knowledge about sequences of characters the user might experience as a token. The parser does not even report *expected* characters. Moreover, reporting expected *tokens* is usually more informative. While the ideal situation would be to have an error reporting strategy that is completely based on a grammar, error messages may improve considerably by providing language-specific examples [26].

The current implementation of SGLR does not perform any error recovery. Declarative mechanisms are needed to specify strategies for error recovery. In practice, grammars are often extended to handle syntactic errors and gracefully continue parsing to report as many errors as possible. If languages are being extended, these error recovery rules might become invalid and might conflict with the language extensions. Preferably, error recovery should be fully automatic [13], but again an example-based approach might be a valuable edition.

Valkering [36] has done early experiments with applying error reporting and recovery techniques for LR parsers to scannerless generalized LR parsers.

5. 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.

5.1. 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. [29]); 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.

Though StringBorg escapes and checks user-supplied strings as well, it does not have the same disadvantages.

- Escaping as well as checking the syntax of user-supplied strings is *automatically* done for user-supplied strings. The developer does not explicitly invoke those routines, so there is no way he can forget this either. Note that explicitly avoiding StringBorg by resorting to the classical string-based approach is immediately visible, and can be detected using straightforward static analysis (see Section 2.3.4).
- StringBorg accepts only those user-supplied strings that match a "good" pattern, but this pattern is directly based on the grammar of the guest language. This grammar should define the exact lexical syntax of the guest language. As a result, StringBorg accepts all the valid inputs, thus not unduly restricting users.
- Our approach is not based on rejecting user-supplied strings that are known to be bad inputs. Instead, StringBorg accepts only (and all) user-input that matches the syntax of the lexical category.

5.2. APIs

SQL DOM [30] 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 [15], 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 [27] 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).

5.3. LINQ

Syntactic hygiene is an important aspect of Haskell Server Pages [31], Cω [4] 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.

5.4. Static analysis techniques

JDBC Checker [18,19] 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 modify the code to add those calls automatically. A tool that uses static analysis to find various kinds of injections is described in [24]. Xie and Aiken [40] developed an interprocedural static analysis algorithm for PHP and apply it to SQL injections.

Livshits and Lam [28] 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.

Static analysis approaches for detecting security vulnerabilities require sophisticated, whole program analysis to avoid reporting large numbers of false positives or being unsound [28]. 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. Some of this effort can be shared for families of languages if the analysis is implemented at the level of a common intermediate representation, e.g. Java bytecode or the .NET common intermediate language. The obvious benefit of approaches based on static analysis is that usually no modification of the source code is required, nor does the programmer need to adopt a different programming style.

5.5. Runtime detection techniques

AMNESIA [20] statically builds an automaton corresponding to the ways in which query strings can be constructed. Nodes in the automaton are terminals in the language, and special nodes represent 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 [14], 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 [14] 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 [21] 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 [12].) The grammar of the guest language is augmented to accept the markers around certain symbols in the grammar, e.g, so that it accepts (|'*s*'|) in SQL wherever a string literal is 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 cross-site scripting (XSS) attack in Fig. 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. Note that a detected injection attack could be handled more gracefully, for example by a global exception handler, but this essentially still makes the service unavailable. 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.

5.6. SQL-specific techniques

The SQL-92 standard [25] 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 [1]. Unfortunately, as with prepared statements, stored procedures are sometimes called in an unhygienic way, negating the approach [2].

5.7. MetaBorg

The method presented in this paper is an extended application of our previous work on concrete object syntax, or *MetaBorg* [11], 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 paper 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 paper 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.

6. Conclusion

As noted in the introduction, injection attacks are one of the largest classes of security problems. Modern programming languages already defend against other common software vulnerabilities, such as buffer overflows and dangling pointers. The work presented in this paper adds protection against injection attacks to that. The main advantage over previous approaches is that it makes injection attacks 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 individual host and guest language. The approach presented in this paper contributes over earlier work on concrete object syntax by improving the usability of the embeddings for programmers without metaprogramming experience. Also, the StringBorg method distinguishes itself from this earlier work by the independence of support for guest and host languages.

Since different languages are good for different purposes, it is important to help programmers combine them. In the case of dynamically generated sentences such as SQL queries or shell invocations, the main challenge is to ensure that this 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.

References

[1] C. Anley, Advanced SQL injection, 2002. http://www.ngssoftware.com/papers/advanced_sql_injection.pdf.
[2] C. Anley, (more) Advanced SQL injection, 2002. http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf.
[3] D. Batory, B. Lofaso, Y. Smaragdakis, JTS: Tools for implementing domain-specific languages, in: Proceedings 5th International Conference on Software Reuse, ICSR'98, IEEE Computer Society Press, Washington, DC, USA, 1998, pp. 143–153.
[4] G. Bierman, E. Meijer, W. Schulte, The essence of data access in Cω, in: ECOOP 2005 — Object-Oriented Programming, 19th European Conference, in: Lecture Notes in Computer Science (LNCS), vol. 3586, Springer, 2005, pp. 287–311.
[5] E. Bouwers, M. Bravenboer, E. Visser, Grammar engineering support for precedence rule recovery and compatibility checking, in: A. Sloane, A. Johnstone (Eds.), Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications, LDTA 2007, in: Electronic Notes in Theoretical Computer Science (ENTCS), vol. 203, Elsevier Science Publishers, 2008, pp. 85–101.
[6] G. Bracha, W. Cook, Mixin-based inheritance, in: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming systems, Languages, and Applications, OOPSLA/ECOOP '90, ACM Press, New York, NY, USA, 1990, pp. 303–311.
[7] M. Bravenboer, Exercises in free syntax. syntax definition, parsing, and assimilation of language conglomerates, Ph.D. Thesis, Utrecht University, Utrecht, The Netherlands, January 2008.
[8] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming 72 (1–2) (2008) 52–70. special issue on Second issue of experimental software and toolkits (EST).
[9] M. Bravenboer, E. Tanter, E. Visser, Declarative, formal, and extensible syntax definition for AspectJ — A case for scannerless generalized-LR parsing, in: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, ACM Press, New York, NY, USA, 2006.
[10] M. Bravenboer, R. Vermaas, J.J. Vinju, E. Visser, Generalized type-based disambiguation of meta programs with concrete object syntax, in: R. Glück, M. Lowry (Eds.), Generative Programming and Component Engineering: 4th International Conference, GPCE 2005, in: Lecture Notes in Computer Science (LNCS), vol. 3676, Springer, Tallinn, Estonia, 2005, pp. 157–172.
[11] M. Bravenboer, E. Visser, Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions, in: D.C. Schmidt (Ed.), Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA'04, ACM Press, New York, NY, USA, 2004, pp. 365–383.
[12] G.T. Buehrer, B.W. Weide, P.A. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: Fifth International Workshop on Software Engineering and Middleware, SEM 2005, ACM Press, New York, NY, USA, 2005.
[13] P. Charles, A practical method for constructing efficient LALR(k) parsers with automatic error recovery, Ph.D. Thesis, New York University, May 1991.
[14] A.S. Christensen, A. Møller, M.I. Schwartzbach, Precise analysis of string expressions, in: Static Analysis Symposium, SAS 2003, in: Lecture Notes in Computer Science (LNCS), vol. 2694, Springer, 2003, pp. 1–18.
[15] W.R. Cook, S. Rai, Safe query objects: Statically typed objects as remotely executable queries, in: Roman et al. [34], pp. 97–106.
[16] T. Ekman, G. Hedin, The JastAdd Extensible Java Compiler, in: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications, OOPSLA 2007, ACM Press, New York, NY, USA, 2007, pp. 1–18.
[17] J. Estublier, D.S. Rosenblum (Eds.), Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, IEEE Computer Society Press, Washington, DC, USA, 2004.
[18] C. Gould, Z. Su, P. Devanbu, JDBC checker: A static analysis tool for SQL/JDBC applications, in: Estublier and Rosenblum [17], pp. 697–698.
[19] C. Gould, Z. Su, P. Devanbu, Static checking of dynamically generated queries in database applications, in: Estublier and Rosenblum [17], pp. 645–654.
[20] W.G. Halfond, A. Orso, AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks, in: 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, Long Beach, California, USA, 2005, pp. 174–183.
[21] W.G. Halfond, A. Orso, P. Manolios, Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, in: Foundations of Software Engineering, FSE 14, 2006.
[22] W.G. Halfond, J. Viegas, A. Orso, A classification of SQL-injection attacks and countermeasures, in: Proceedings of the International Symposium on Secure Software Engineering, ISSSE 2006, 2006.
[23] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd ed., Addison-Wesley, Boston, MA, USA, 2006.
[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing web application code by static analysis and runtime protection, in: 13th International World Wide Web Conference, WWW2004, ACM Press, New York, NY, USA, 2004, pp. 40–52.
[25] ISO, ISO/IEC 9075:1992: Database language SQL, 1992.
[26] C.L. Jeffery, Generating LR syntax error messages from examples, ACM Transactions on Programming Languages and Systems (TOPLAS) 25 (5) (2003) 631–640.
[27] D. Leijen, E. Meijer, Domain specific embedded compilers, in: Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 1999), ACM Press, New York, NY, USA, 1999, pp. 109–122.

[28] V.B. Livshits, M.S. Lam, Finding security vulnerabilities in Java applications with static analysis, in: Proceedings of the 14th Usenix Security Symposium, USENIX, 2005, pp. 271–286.
[29] O. Maor, A. Shulman, SQL injection signatures evasion, White paper, Apr. 2004. http://www.imperva.com/resources/adc/sql_injection_signatures_evasion.html.
[30] R.A. McClure, I.H. Krüger, SQL DOM: Compile time checking of dynamic SQL statements, in: Roman et al. [34], pp. 88–96.
[31] E. Meijer, D. van Velzen, Haskell Server Pages: Functional programming and the battle for the middle tier, in: 2000 ACM SIGPLAN Haskell Workshop, in: Electronic Notes in Theoretical Computer Science (ENTCS), vol. 41/1, Elsevier Science Publishers, 2001.
[32] A. Møller, dk.brics.automaton — finite-state automata for Java, 2005. http://www.brics.dk/automaton/.
[33] N. Nystrom, M.R. Clarkson, A.C. Myers, Polyglot: An extensible compiler framework for Java, in: G. Hedin (Ed.), Compiler Construction, 12th International Conference, CC 2003, in: Lecture Notes in Computer Science (LNCS), 2622, Springer, 2003, pp. 138–152.
[34] G.-C. Roman, W.G. Griswold, B. Nuseibeh (Eds.), Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, IEEE Computer Society Press, Washington, DC, USA, 2005.
[35] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, USA, 2006, pp. 372–382.
[36] R. Valkering, Syntax error handling in scannerless generalized LR parsers, Master's Thesis, Programming Research Group, University of Amsterdam, Amsterdam, The Netherlands, Aug. 2007.
[37] E. Visser, Syntax definition for language prototyping, Ph.D. Thesis, University of Amsterdam, September 1997.
[38] E. Visser, Meta-programming with concrete object syntax, in: D. Batory, C. Consel, W. Taha (Eds.), Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, in: Lecture Notes in Computer Science (LNCS), 2487, Springer, 2002, pp. 299–315.
[39] E. van Wyk, L. Krishnan, A. Schwerdfeger, D. Bodin, Attribute grammar-based language extensions for Java, in: European Conference on Object Oriented Programming, ECOOP 2007, in: Lecture Notes in Computer Science (LNCS), Springer, 2007.
[40] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: 15th USENIX Security Symposium, USENIX, 2006, pp. 179–192.
[41] D. Zook, S.S. Huang, Y. Smaragdakis, Generating AspectJ programs with Meta-AspectJ, in: G. Karsai, E. Visser (Eds.), Generative Programming and Component Engineering: Third International Conference, GPCE 2004, in: Lecture Notes in Computer Science (LNCS), 3286, Springer, Vancouver, Canada, 2004, pp. 1–19.