Type-Specific Languages to Fight Injection Attacks

Darya Kurilova  
Carnegie Mellon University  
darya@cs.cmu.edu

Cyrus Omar  
Carnegie Mellon University  
comar@cs.cmu.edu

Ligia Nistor  
Carnegie Mellon University  
lnistor@cs.cmu.edu

Benjamin Chung  
Carnegie Mellon University  
bwchung@cs.cmu.edu

Alex Potanin  
Victoria University of Wellington  
alex@ecs.vuw.ac.nz

Jonathan Aldrich  
Carnegie Mellon University  
aldrich@cs.cmu.edu

1. PROBLEM AND MOTIVATION

Injection vulnerabilities have topped rankings of the most critical web application vulnerabilities for several years [1, 2]. They can occur anywhere where user input may be erroneously executed as code. The injected input is typically aimed at gaining unauthorized access to the system or to private information within it, corrupting the system’s data, or disturbing system availability. Injection vulnerabilities are tedious and difficult to prevent.

Modern programming languages provide specialized notations for common data structures, data formats, query languages, and markup languages. For example, a language with built-in syntax for HTML and SQL, with type-safe interpolation of host language terms via curly braces, might allow:

```javascript
let webpage = HTML = <html><body>
<ol id="results">{to_list_items(query(db, [ULElement(Dict.add(Dict.empty(), [TextNode("Results for " + keyword)], [BodyElement(Dict.empty(), [H1Element(Dict.empty(), "<h1>Results for {keyword}</h1>" + let [WhereClause(InPredicate(StringLit(keyword), 6 "WHERE '{" + keyword + "' in title"))])) + 7 "]/")</body></html>"))}) + 7 "]/")</body></html>"
</ol></body></html>
```

to mean:

```
let webpage = HTML = <html><body>
<ol id="results">{to_list_items(query(db, 4 SELECT title, snippet FROM products 5 WHERE (keyword in title))} + 6 
</ol></body></html>
```

Although recovering much of the notational convenience of the literal version, this code is fundamentally insecure as it is vulnerable to cross-site scripting (XSS) attacks and SQL injection attacks. For example, the keyword in line 2 may introduce a `<script>` tag and provide malicious JavaScript code to execute, or alternatively, a user may provide the keyword “DROP TABLE products --” in line 6, and the entire product database will be erased.

Developers are cautioned to sanitize their input; however, it can be difficult to verify that this was done correctly throughout a codebase. The most straightforward way to avoid these problems is to use structured representations, aided by specialized notation like that above. Unfortunately, prior mechanisms either limit expressiveness or are not safely composable, i.e., individually-unambiguous extensions can cause ambiguities when used together. We introduce type-specific languages (TSLs), where logic associated with a type that determines how generic literal forms, which are able to contain arbitrary syntax, are parsed and expanded into general-purpose syntax.

2. BACKGROUND AND RELATED WORK

As input injection vulnerabilities are one of the most common and dangerous web security issues, numerous projects have attempted to solve this problem. To begin with, there are radically different solutions that do runtime verification, for example, by employing taint analysis, such as [4, 10, 11, 14]. Then, there are solutions that use a type system to resolve these security problems but rely upon adding special annotations, such as [5, 6, 7]. The main disadvantage of these approaches is that, requiring sophisticated notations, they impose significant mental overhead on software developers, which our approach strives to avoid.

From the language extensibility perspective, implementing new notations in an existing programming language requires cooperation of the language designer as, with conventional parsing strategies, not all notations can safely coexist. For example, a conflict can arise if there are simultaneous extensions for HTML and XML, different variants of SQL, etc., and a designer is needed to make choices about which syntactic forms are available and what their semantics are (e.g., in systems like [8] or, more generally, [9]). In contrast, TSLs guarantee safe and unambiguous language extension composition. Bravenboer et al. [3] describe a way to protect against
Figure 1: Wyvern Example with Multiple TSLs

injection attacks by embedding guest languages (e.g., SQL) into host languages (e.g., Java) that is akin to ours. However, their system pieces together existing languages whereas we are going one step further and develop a new programming language that is secure by construction and supports modular language extensions “out-of-the-box.” Additionally, in terms of language extensibility, their approach targets specifically injection attacks while ours is more general and can be used to extend a language with other features as well.

3. THE SCIENCE

Our work applies programming language theory to discover a new foundational principle for modular language extensibility [13]. This, in turn, adds to the body of scientific knowledge concerning defense against injection attacks. Hence, our mechanism makes a contribution in both research fields—Programming Languages and Software Security.

4. APPROACH

We develop our work as a variant of an emerging programming language called Wyvern [12]. We propose an alternative strategy that allows to extend the host language with easy-to-understand user-defined literal forms while preventing injection attacks. This is achieved by shifting responsibility for parsing certain generic literal forms into the typechecker. The typechecker, in turn, defers responsibility to user-defined types, by treating the body of the literal as a term of the type-specific language (TSL) associated with the type it is being checked against. The TSL Rewrites this term to use only general-purpose notations and can contain expressions of the host language. This strategy eschews the problem of ambiguous syntax, because neither the base language nor TSLs are ever extended directly. It also avoids ambiguous semantics and frees notation from being tied to the variant of a data structure built into the standard library, which sometimes does not provide the exact semantics that a developer needs.

Figure 1 presents a 19-line piece of Wyvern code that comprises 8 different TSLs (marked with different colors) used to define a fragment of a web application showing search results from a database. Using the code layout and whitespaces to delimit the host language, customizable literals, such as ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,”Easy to read and thus reduces cognitive load on the developer.

**HTML Interpolation** At any point where a tag should appear, we can also interpolate a Wyvern expression of type HTML by enclosing it within curly braces (e.g., on lines 13, 15, and 16-19 of Figure 1). This functionality is implemented in the Wyvern type HTML (not shown). Because interpolation must be structured, i.e., a string cannot be interpolated directly, injection and cross-site scripting attacks cannot occur. Safe string interpolation (which escapes any inner HTML) could be implemented using another delimiter.

**SQL Interpolation** The TSL used for SQL queries on lines 17-18 of Figure 1 follows an identical pattern, allowing strings to be interpolated into portions of a query in a safe manner. This prevents SQL injection attacks.

5. CONTRIBUTIONS

Our contribution is two-fold: Firstly, we introduced type-specific languages (TSLs), a mechanism for safely composing language extensions that associates the logic determining how generic literal forms are parsed and expanded into general-purpose syntax with a type and that aims at lessening the developer’s cognitive burden. We incorporated this mechanism into the Wyvern programming language. Secondly, we showed how, using TSLs, injection attacks, such as cross-site scripting (XSS) attacks and query injection attacks, are prevented enhancing the security of software written in Wyvern.

6. REFERENCES