Lifting the curse of stringly-typed code

Eddie Antonio Santos, Karim Ali

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

How often do JavaScript programmers embed structured languages into strings literals? We conduct an empirical investigating mining nearly 500 thousand JavaScript source files from almost ten thousand repositories from GitHub. We parsed each string literal with seven separate common grammars, and found the most common data type that is hidden within the confines of string literals. To reduce the overuse of strings for structured data types, we present a static program analyzer that finds embedded languages and warns the developer, providing an optional fix.

Lifting the curse of stringly-typed code

Eddie Antonio Santos, Karim Ali
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

Abstract
How often do JavaScript programmers embed structured languages into strings literals? We conduct an empirical investigating mining nearly 500 thousand JavaScript source files from almost ten thousand repositories from GitHub. We parsed each string literal with seven separate common grammars, and found the most common data type that is hidden within the confines of string literals. To reduce the overuse of strings for structured data types, we present a static program analyzer that finds embedded languages and warns the developer, providing an optional fix.

Email(s):
`easantos@ualberta.ca` (Eddie Antonio Santos); `karim.ali@ualberta.ca` (Karim Ali)

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2723v1 | CC BY 4.0 Open Access | rec: 13 Jan 2017, publ: 13 Jan 2017

```
function hostname(text) {
  var matchResult =
    text.match(/^https?:[/][/]([^/]+)/);
  return matchResult[1];
}

console.log(hostname(
  'https://example.org'
));
```

Listing 1 Using a regular expression to retrieve structured information from a string.

1. INTRODUCTION

Stringly-typed programming is a colloquial term used to describe:

> an implementation that needlessly relies on strings when programmer & refactor friendly options are available. [1]

Consider the JavaScript program given in Listing 1. A function called `hostname` retrieves the host information from a URL that is passed to the function as a string. `hostname` uses a regular expression to partially parse the string, matching the desired content in capturing group, and returning the result. While such a regular expression is sufficient for many URLs, it is not appropriate in all cases. This is because URLs adhere to a syntax, which is formally defined by a standard as published by the IETF [2].[1] Thus, when one encounters a URL embedded within a string literal in source code, one is witnessing a *language* embedded within a programming language. The embedding of other languages in source code, without any explicit means of parsing and handling said string is a symptom of a stringly-typed implementation.

What's wrong with writing a stringly-typed implementation? The primary reason is that the string provides an *inappropriate level of abstraction* for the structured data contained within a string. Listing 1 reveals a value that adheres to an underlying URL syntax, yet the developer opted to use ad hoc mechanisms of extracting structured data from within the confines of the string. Using strings forces developers to resort to low-level string operations such as indexing, or writing ad hoc regular expressions to parse out just the bit of structured meaning that the developer needs. However, such low-level operations come with a number of drawbacks. For example, indexing is notoriously prone to off-by-one errors. In the case of programming languages that do not perform bounds checking such as C, invalid bounds checking can lead to *buffer overflow exploits* that have the capability of executing arbitrary code written by a malicious user. As Listing 1 shows, the ad hoc

[1]URLs are a subset of URIs in a semantic sense. While URIs *identify* resources, URLs are URIs that can be used to *locate* a resource—that is, they provide a means of accessing a resource [2]. Henceforth in this paper, we will be using URIs and URLs interchangeably.

regular expression can easily become inadequate if given a different, but valid URL such as `mailto:somebody@example.org`.

What would be better is if the language provided first-class mechanisms to allow for embedded languages. For example, the Wyvern language [15] has first-class support for *type-specific languages* that allows the developer to write grammars that parse their own custom data types in the language. Alas, this is not the case for JavaScript.

This paper seeks to help developers find symptoms of stringly-typed code in JavaScript. We seek to answer the following research questions:

RQ1. How often do we find evidence of stringly-typed programming?

RQ2. What types are hiding within string literals?

RQ3. How can we support developers to detect and alleviate stringly-typed code?

With RQ1, we seek to get an understanding of the degree of stringly-typed code in open source JavaScript projects. We elaborate on this with RQ2, in that we want to characterize the different languages that developers are embedding within string literals. Finally, with RQ3, we seek to create a tool that can statically find instances of stringly-typed code and offer solutions.

2. DATA COLLECTION

Repositories	9,886
`.js` files	594,681
Parsed `.js` files	494,352

Table 1 Overview of the October 2016 GitHub JavaScript corpus.

To answer RQ1 and RQ2, regarding the frequency of stringly-typed implementations, we require a corpus to study, In order to determine the frequency of embedded languages within strings, we mined a corpus JavaScript projects (Table 1. This is the same corpus used by Santos [18]). This corpus was collected by mining JavaScript repositories from GitHub in October 2016.[2] Using GitHub's search API, we queried for the top starred JavaScript repositories. Since the search API only returns a maximum of 1000 results, after every 1000 repositories, we ran a new query with repositories whose stars were lower than the lowest-starred repository of the previous batch.

For each of the 9,886 repositories, we downloaded the latest snapshot of its default branch (usually `master`), and kept any files with the `.js` extension. We computed the

[2]Available: `https://archive.org/details/javascript-sources-oct2016.sqlite3`

SHA512 hash of each file, and used it to avoid storing any byte-for-byte duplicates of already downloaded files. For each of the byte-for-byte unique 594,681 files, we used Esprima [6] to obtain an abstract syntax tree (AST) of the JavaScript source code. Not only does this ensure that we are studying only syntactically-valid JavaScript files, but an AST greatly facilitates the extraction of all string literals and their contexts in the source code. Esprima was not able to parse 100,329 files (16.87% of the corpus) according to the ECMAScript 2016 specification [5]; thus, we were left with 494,352 parsed JavaScript source files. All results that follow in this paper are based on these parsed JavaScript files.

3. LANGUAGES

Name	Standard	Grammar	Example
IPv4 [4]	RFC 1123	Core PEGjs	127.0.0.1
IPv6 [7]	RFC 3513	Core PEGjs	2606:2800:220:1:248:1893:25c8:1946
ISO Date [8]	ISO 8601:2004	Core PEGjs	1994-03-19
ISO Datetime [8]	ISO 8601:2004	Core PEGjs	1994-03-19T10:00:00+09:00
SemVer [16]	Semantic Versioning	npm	2.0.0-rc.1
URI [2]	RFC 3986	Core PEGjs	mailto:nobody@example.org
UUID [10]	RFC 4122	Paper's authors	123e4567-e89b-12d3-a456-426655440000

Table 2 Summary of languages studied. "Grammar" refers to the source of the grammar we used when parsing string literals.

We chose seven languages to find within JavaScript string literals. A summary of these languages is provided in Table 2. These seven were chosen arbitrarily by the first author, however with the rationale that that these languages may be popular among developers. Empirically discovering an exhaustive list of the most popular embedded languages is future work. Four of these languages—URIs, IPv4 addresses, IPv6 addresses, and UUID—are standards as specified by the Internet Engineering Task Force (IETF). Since JavaScript is a programming language primarily made for the web, we expected many instances of these within source code. "ISO date" and "ISO Datetime" are common date/time formats as specified by ISO 8601 [8]. "SemVer" refers to *Semantic Versioning* strings which is a guideline for specifying software version numbers in a way to communicates progress and gives a idea of backwards-compatibility [16]. The Node.JS community predominantly uses SemVer to version their software [3].

We used three main grammar sources to parse these languages. With the exception of SemVer, we used grammars written as parsing expressions grammars given as input to the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2723v1 | CC BY 4.0 Open Access | rec: 13 Jan 2017, publ: 13 Jan 2017

PEG.js parser generator [11]. For IPv4, IPv6, ISO time and date formats, and URIs, we used grammars provided by the Core PEGjs repository [13]. For UUIDs, the authors wrote a PEG.js grammar according to the IETF specification [10].[3] SemVer was parsed by the official Node npm client [19].

4. RESULTS

	Min	Max	Range	Median
All strings	1	367,414	367,413	14
Stringly-typed	0	27,097	27,097	0

Table 3 Summary of the number string literals found per file. We filtered only files that contained at least one string literal.

A summary string literals found per file is given in Table 3. We found 66,936,913 string literals in 468,129 files. Given 494,352 parsed files in our corpus total, that means that 94.70% of all files contained at least one string literal. Of these nearly 67 million string literals, we found that 1,532,089 were parsed by at least one of our grammars. These stringly-typed literals were spread over 90,004 files—19.23% of files with strings, or 18.21% of all files. We wanted to know, of the 90,004 files that contains at least one stringly-typed literal, what proportion of that file is stringly-typed? Figure 1 is a density plot of this data. The density seems to follow a power-law distribution, where 79,482 or 88.31% of files are 20% stringly-typed or lower.

RQ1 How often do we find evidence of stringly-typed programming? About 18.21% of files are stringly-typed. Expect at most 20% of the literals within the file to be stringly-typed.

For each of the chosen languages (Section 3), we counted how many string literals parsed against the grammars. The results are ranked, from most common to least common in Table 4. URIs are the most prevalent embedded language by far. Summing up the rest of the languages gives 637,589 string literals. Thus, URIs outnumber all other embedded languages combined. ISO dates are second, followed by ISO datetimes, which is an extension ISO dates.

RQ2 What types are hiding within string literals? URIs are by far the most prevalent type, followed by ISO date and time formats.

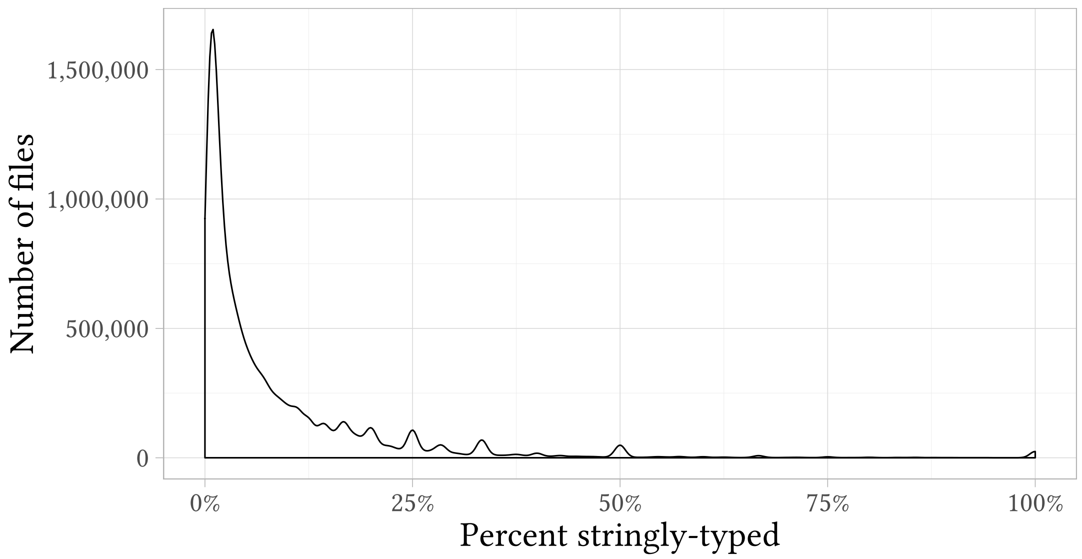

Figure 1 Density plot showing the percentage of strings in a file that are stringly-typed.

Rank	Syntax	Occurrences	Histogram
1	URI	894,507	████████
2	ISO Date	410,369	████
3	ISO Datetime	119,071	█
4	SemVer	57,679	▌
5	UUID	37,119	▌
6	IPv4	10,197	▏
7	IPv6	3,154	▏

Table 4 Occurrences of each language parsed in the corpus.

5. LINTER PLUGIN

Given the results in Section 4, we were motivated to create a tool that would detect usages of URIs in string literals. The server-side JavaScript runtime, Node.JS, is distributed with the

[3]`https://github.com/eddieantonio/string-rewriter/blob/cmput620/grammars/rfc4122-uuid.pegjs`

```
const URL = require('url-tagged-template');

let myBadURL1 = 'http://example.com/index.php?page=1';
let myBadURL2 = String.raw`http://example.net/#header`;

let myGoodURL = URL`http://example.org`;

console.log(myBadURL1.hostname);
console.log(myBadURL2.hostname);
console.log(myGoodURL.hostname);
```

Listing 2 A file with bare and parsed URLs

```
 $ eslint index.js

/Users/eddieantonio/my-project/index.js
  3:17  warning  Unexpected bare URL: http://example.com/index.php?page=1  stringly-
      typed/no-bare-url
  4:27  warning  Unexpected bare URL: http://example.net/#header           stringly-
      typed/no-bare-url
```

Listing 3 Using `eslint-plugin-stringly-typed`

`url` library [14] for parsing and manipulating URLs as standard JavaScript objects rather than passing around bare string literals. To help developers detect bare URLs and encourage them to use objects, we wrote an extension for the ESLint JavaScript linter.[4]

A linter is a tool that statically analyzes source code for stylistic issues and usage of deprecated or otherwise discouraged behaviour. ESLint [9] is a linter for JavaScript with an extensible plugin architectures. ESLint provides several *rules* that can detect and report issues and allows for plugin authors to extend ESLint with additional rules. Example rules include `no-extra-semi` that forbids extraneous semi-colons in source code; `indent`, a customizable rule that enforces the developers preferred amount of indentation (tabs, 2 spaces, 4 spaces, etc.); and `eqeqeq` which forbids the use of the weakly-typed equality operator (==).

Our plugin implements one ESLint rule: `no-bare-url`. Given a file with string literals that encode URLs, our rule reports all instances that are not immediately parsed. Listing 3 shows the output of our tool given a JavaScript file, Listing 2, which contains bare URLs as well as parsed URLs.

ESLint allows rules to implement automated fixes. We implemented a fixer that simply prefixes URL tagged template [17], which automatically parses the given string literal and

[4]Available: https://github.com/eddieantonio/eslint-plugin-stringly-typed

returns a URL object, despite appearing in the source like a string literal. Hence, our ESLint plugin does not warn when it encounters string literals with the URL tag.

6. FUTURE WORK

One of the drawbacks of our technique for finding embedded languages was the reliance of hand-picked languages. As mentioned, this is a threat to construct validity. For future mining tasks, we would like to use an automated way of finding embedded languages. That is, a method that discovers systematic structures present among a small proportion of string literals in an entire corpus. As a corollary to this, we would like for an analysis that discovers many more languages than the seven we considered in this paper.

Our ESLint plugin is limited to string literals alone, and considers only the immediate context of the literal. Instead, an interprocedural data-flow analysis may be desired to find the ultimate sinks of such string literals, to see *how* they are ultimately used. For example, a string containing a URL may be considered safe if it is passed directly to a well-known API call—for example, to perform an HTTP request. However, if in the process the string undergoes some low-level string operations, then our tool would provide a warning, and suggest using a parser instead.

Although the stringly-typed programming is a code smell, to our knowledge, there has been minimal empirical study on determining whether code that exhibits stringly-typed programming is more faulty, or perhaps introduces security vulnerabilities. One possible study is to correlate the degree of stringly-typed programming, as determined in this paper, with the amount of discovered security vulnerabilities produced by a program analysis tool such as WALA [20]. Moreover, analyzing the structure of strings statically can unlock new information for more precise taint analysis of JavaScript programs.

Even better is if the language had first-class support for textual literals that are parsed at compile-time into an appropriate type. The JavaScript community is fond of languages that "transpile" into JavaScript, including a statically-typed variant called TypeScript [12]. One could extend TypeScript such that it automatically parsed string literals with embedded structure, and instead of emitting ordinary string literal code, it would emit an object that contains a parsed representation of the string literal. Then, the type annotations provided in the language could be used to designate the desire to use a well-defined type, such as a URL, but use the exact same string literal notation as is used today.

7. THREATS TO VALIDITY

Internal The amount of true matches of each grammar may be less than what is reported in this paper; simply matching a string against a grammar does not capture the programmer's *intent*, hence we may be reporting a number of false positives.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2723v1 | CC BY 4.0 Open Access | rec: 13 Jan 2017, publ: 13 Jan 2017

External Our conclusions may not be safely generalized beyond the scope of open source JavaScript repositories. There may be a different distribution of stringly-typed literals in other domains such as closed-source code.

Construct To ensure we were parsing code from non-trivial JavaScript projects, we chose to take the top starred projects on GitHub. However, these projects skew towards libraries and frameworks such as jQuery, and Angular.JS. Thus, the corpus we studied may not be representative of application code. Stars on GitHub is more-or-less a popularity metric. Thus, with so many eyes on these codebases, stylistic concerns, such as a high-degree of stringly-typed code, may be less prevalent than in private, application code. We did not filter for test files, so our results may include any string literals found in tests for parsing libraries.

The particular languages we chose in this paper add a huge amount of bias to our conclusions. We did not consider many possible embedded languages, thus our results may only report a small fraction of possible stringly-typed code.

8. CONCLUSION

In this paper, we sought to find out how prevalent stringly-typed coding is in open source JavaScript projects. We found that about 18.21% of files contain at least one string literal than can be parsed by a common grammar. If a string literal is parsable by any grammar, it is most likely a URI. Finally, we created `eslint-plugin-stringly-typed` that detects and offers fixes for instances of bare URIs found in JavaScript source files.

Stringly-typed code occurs often enough to warrant special attention. One way to imbue string literals with meaning is creating new programming languages that allow the definition of arbitrary literal types, such as Wyvern [15]. However, we cannot disregard existing languages such as JavaScript. To help developers find suspicious instances of stringly-typed code, we have provided a simple linting tool.

REFERENCES

[1] Stringly typed. `http://wiki.c2.com/?StringlyTyped`, January 2013. (Accessed on 2016-11-26).

[2] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, RFC Editor, January 2005.

[3] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 109–120, New York, NY, USA, 2016. ACM.

[4] Robert Braden. Requirements for Internet Hosts—Application and Support. RFC 1123, RFC Editor, October 1989.

[5] ECMA TC39. *ECMA-262: ECMAScriptÂő 2016 Language Specification.* Ecma International, Geneva, Switzerland, 7th edition, June 2016.

[6] Ariya Hidayat. Esprima. Avaiable: http://esprima.org/, 2016.

[7] Robert M. Hinden and Stephen E. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513, RFC Editor, April 2003.

[8] Date and time format. Standard, International Organization for Standardization, Geneva, CH, December 2004.

[9] JS Foundation and other contributors. Eslint—pluggable JavaScript linter. http://eslint.org/, November 2016. (Accessed on 2016-11-28).

[10] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, RFC Editor, July 2005.

[11] David Majda. PEG.js âĂŞ parser generator for JavaScript. http://pegjs.org/, 2016. (Accessed on 2016-11-26).

[12] Microsoft Corporation. TypeScript—JavaScript that scales. https://www.typescriptlang.org/, December 2016. (Accessed on 12/09/2016).

[13] Andrei Neculau. for-get/core-pegjs: A collection of core PEGjs grammars (IETF, ISO, etc.). https://github.com/for-GET/core-pegjs/, 2014. (Accessed on 2016-11-26).

[14] Node.js Foundation. Url | node.js v7.2.1 documentation. https://nodejs.org/api/url.html, November 2016. (Accessed on 12/08/2016).

[15] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *European Conference on Object-Oriented Programming*, pages 105–130. Springer, 2014.

[16] Tim Preston-Werner. Semantic Versioning 2.0.0—Semantic Versioning. http://semver.org/spec/v2.0.0.html, June 2013. (Accessed on 2016-11-26).

[17] Eddie Antonio Santos. eddieantonio/url-tagged-template: ES6 URL tagged template for use with Node.JS. https://github.com/eddieantonio/url-tagged-template, November 2016. (Accessed on 12/08/2016).

[18] Eddie Antonio Santos. Syntax and sensibility: Using LSTMs to detect and fix syntax errors. Class project, December 2016.

[19] Isaac Z. Schlueter. semver. https://www.npmjs.com/package/semver, July 2016. (Accessed on 12/08/2016).

[20] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, 2013. ACM.