No Strings Attached: An Empirical Study of String-related Software Bugs

Aryaz Eghbali
aryaz.egh@gmail.com
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

ABSTRACT

Strings play many roles in programming because they often contain complex and semantically rich information. For example, programmers use strings to filter inputs via regular expression matching, to express the names of program elements accessed through some form of reflection, to embed code written in another formal language, and to assemble textual output produced by a program. The omnipresence of strings leads to a wide range of mistakes that developers may make, yet little is currently known about these mistakes. The lack of knowledge about *string-related bugs* leads to developers repeating the same mistakes again and again, and to poor support for finding and fixing such bugs. This paper presents the first empirical study of the root causes, consequences, and other properties of string-related bugs. We systematically study 204 string-related bugs in a diverse set of projects written in JavaScript, a language where strings play a particularly important role. Our findings include (i) that many string-related mistakes are caused by a recurring set of root cause patterns, such as incorrect string literals and regular expressions, (ii) that string-related bugs have a diverse set of consequences, including incorrect output or silent omission of expected behavior, (iii) that fixing string-related bugs often requires changing just a single line, with many of the required repair ingredients available in the surrounding code, (iv) that string-related bugs occur across all parts of applications, including the core components, and (v) that almost none of these bugs are detected by existing static analyzers. Our findings not only show the importance and prevalence of string-related bugs, but they help developers to avoid common mistakes and tool builders to tackle the challenge of finding and fixing string-related bugs.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

strings, software bugs, string-related bugs, empirical study

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416576

ACM Reference Format:
Aryaz Eghbali and Michael Pradel. 2020. No Strings Attached: An Empirical Study of String-related Software Bugs. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416576

1 INTRODUCTION

Programs are full of strings. The reason why strings are so popular in programming is that they can express semantically rich and complex information, while providing maximum flexibility. For example, programmers use strings to filter user-provided inputs using regular expressions and to build textual outputs of a program. Strings also play an important role in dynamic programming patterns, such as reflection-like access of an object property based on the property name. To inspect, manipulate, and create strings, most programming languages provide a rich set of string APIs. Finally, strings often also serve as an interface to other formal languages, e.g., when embedding database queries, shell commands, or document markup code via a string.

The prevalence of strings in programming leads to a wide range of possible string-related mistakes. When manipulating strings, programmers may accidentally misuse some API, use a wrong regular expression, or forget about some special string value. When embedding code in another formal language into the program via a string, programmers may write syntactically incorrect code or refer to non-existing identifiers. Even when using simple string literals, e.g., to represent file paths or enum-like properties, programmers may easily introduce typos or forget to update a string when the program evolves. Unfortunately, most compilers and bug detection tools are of little help because they rarely reason about the value of a string variable or how it relates to the rest of the program.

Despite the omnipresence of strings and the various potential mistakes related to them, little is currently known about string-related bugs. This lack of knowledge has several negative consequences. First, developers are likely to repeat the same mistakes again and again. If recurring string-related bug patterns were known, developers could derive best practices to avoid them. Second, creators of bug detection techniques, e.g., static and dynamic analyses, and of automated program repair tools do not know what kinds of problems are most pressing. Third, designers of APIs and programming languages, who may want to prevent some classes of bugs by design, can only guess what mistakes developers suffer from.

This paper presents the first empirical study of string-related bugs in widely used, real-world software. To address the lack of knowledge about this class of mistakes, we ask the following research questions:

- RQ 1: What are the main root causes of string-related bugs?
- RQ 2: What consequences do string-related bugs entail, i.e., what kinds of misbehavior do they cause and how do these bugs surface?
- RQ 3: How do developers fix string-related bugs?
- RQ 4: What kinds of software components or parts of an application are most affected by string-related bugs?
- RQ 5: How effective are widely used bug detection tools at finding string-related bugs?

To address these questions, we systematically study string-related bugs in a diverse set of real-world JavaScript projects. Choosing JavaScript as the target language is motivated by the fact that strings play a particularly important role in this language. One reason is that JavaScript is widely used in the web, where strings are heavily used to manipulate websites and to send data over the network. Another reason is the dynamic nature of the language, which motivates developers to represent and manipulate various concepts as strings, including references to program elements and even code itself. We systematically gather a set of 204 string-related bugs from 13 popular open-source projects. Each bug consists of the incorrect code, the fix applied by the developers, and some informal description of the problem. Beyond our study, this dataset will support future work on string-related bugs, e.g., on finding and fixing these problems.

Our main findings include the following:

- The vast majority (95.6%) of string-related bugs are caused by one or more recurring kinds of root causes. These root cause patterns will guide future efforts toward finding string-related bugs.
- The most prevalent root causes are incorrect string literals and incorrect regular expressions (42% and 37% respectively), making these two problems prime targets for bug detection techniques.
- Many bugs cause the program to produce incorrect output (30%) or to corrupt a file (5%), while only 11% of the bugs lead to an error message, i.e., an obvious sign of misbehavior. These results underline the need for clever test oracles [4], beyond generic signs of misbehavior, such as error messages.
- A significant fraction of the bugs (18%) manifest only in some environments, e.g., a specific operating system or browser. This finding motivates the automated analysis of code in different environments.
- String-related bugs affect all components of the studied software systems, with 53% of the bugs affecting the core functionality of the project, showing that string-related bugs are an important class of problems.
- Most bugs (61%) are fixed by modifying a single line of code, making them a promising target for automated program repair [25].
- For a significant fraction of the bugs (26%), all code tokens required in the fix occur within the same file in the vicinity of the fix location. Moreover, 45% of the bugs have at least 70% of the ingredients in the vicinity. The availability of such repair ingredients will facilitate automated repairing efforts.
- A widely used static code analyzer finds only one out of the 204 studied bugs. This result confirms an earlier finding that static analyzers miss most bugs [18] and shows that this finding holds for string-related bugs in particular.

In summary, this paper contributes (1) the first systematic study of string-related bugs, (2) findings about the root causes, consequences, and other properties of these bugs, (3) evidence that developers need tools to find and fix string-related bugs, (4) a documented dataset of 204 bugs. Our dataset will be available as a reference point for future work on best practices for developers, bug detection tools, automated repair tools, and the design of string-related APIs and programming language features.

2 METHODOLOGY

2.1 Scope

To conduct a systematic study of string-related bugs, it is essential to have a clear definition of what bugs are considered string-related, and which types of those bugs fall into the scope of this study. We define three criteria for selecting the subjects of our study, which we describe and illustrate with examples in the following.

Criterion 1: Bug. The study focuses on bugs, i.e., problems in the source code that cause the functional behavior of the program to diverge from the expected behavior. The problems can range from code crashes over omitted behavior to incorrect output messages. In contrast, new features and any other kinds of code improvements, e.g., refactorings, are not considered in the scope of this study.

Criterion 2: String-only problem. We focus on string-related bugs, which we define as bugs that are possible only with strings or custom string wrappers. This criterion excludes bugs that coincidentally involve a string value but that may just as well happen with another data type. For example, although the following example from the Socket.IO framework involves strings, it does not fulfill our criterion:

```
-  var origin = this.req.headers['origin']
+  var origin = this.req.headers['origin'] || ''
```

The reason is that forgetting to set a default value for a variable may happen for several other data types. For example,the above bug could also happen with a number that should be initialized to zero by default. In contrast, the following bug from the npm package manager is caused by a faulty regular expression, i.e., it cannot happen with a data type other than string:

```
-  u = u.replace(/^git\+ssh:\/\//, "")
-       .replace(/^git\+/, "")
+  u = u.replace(/^git\+/, "")
```

Although comments in the code are sequences of characters, and hence similar to strings, we exclude fixes in comments from the study, as comments are not strings. Thus, this wrong type declaration in the Mongoose tool,

```
  /**
  * ...
- * @param {Number} val
+ * @param {String} val
  * ...
  */
```

or this typo in a comment that provides a code example in Facebook's React framework

```
  /**
  * ...
- *   var ReactComponentWithPureRender =
+ *   var ReactComponentWithPureRenderMixin =
  * ...
  */
```

are not considered string-related bugs.

Criterion 3: Programming language file. String-related code changes may happen in documentation or configuration files. The study focuses on bugs in programming language files, though, because our goal is to gain insights for program analysis of code written in Turing-complete languages. For example, this criterion excludes version number changes in configuration files. Likewise, if a JSON file contains some scripts, it also is beyond the scope of this study. For example, the following incorrect command for running the tests of the Socket.IO framework is not considered in the study:

```
  "scripts": {
-     "test": "make test"
+     "test": "mocha --reporter dot --slow 200ms --bail"
  }
```

2.2 Data

All widely used programming languages support strings. We focus our study on string-related bugs in JavaScript code, which is motivated by three reasons. First, JavaScript has become one of the most popular languages, oftentimes even reported as the most popular language of all [9]. Second, JavaScript code covers a wide range of application domains. While traditionally JavaScript has been mostly used for client-side web applications, it has become popular for server-side applications, mobile applications, cloud applications, and many others. As a result of the first two points, there are many open-source projects to study, including projects backed by large software organizations, such as Facebook and Mozilla, and smaller community-driven projects. Third, strings play an important role in JavaScript, perhaps even more than in other languages, because they are heavily used to manipulate websites, to send data over the network, or to represent references to program elements and even code itself in a string.

We gather string-related bugs from 13 JavaScript repositories selected from the highest starred repositories on GitHub (Table 1). The selection of these projects aims at covering a diverse set of application domains and platforms, including stand-alone applications, tools, libraries, and frameworks. We selected JavaScript repositories on GitHub by their stars in descending order, while ignoring websites (like freeCodeCamp/freeCodeCamp) and any project that had a similar (in terms of domain and use case) project selected before itself. [1] To emphasise the diversity of our selection, we briefly describe each project. Atom is a desktop application for editing text and source code, with support for plugins written in Node.js. Mozilla's PDF.js renders PDF files using the web standards. Impress.js is a feature-rich application to create non-conventional presentations. Video.js is a video player that uses HTML5's video functionality. Babel is a transcompiler that is used to convert JavaScript code written in newer ECMAScript standards to older versions. The npm/cli project, provides the command line interface for the well-known node package manager. Backbone.js is a framework based on the model-view-presenter design paradigm that is used for developing web applications. JQuery is a famous library used for simpler HTML DOM operations. Koa is a HTTP middleware framework for Node.js. Mongoose is a MongoDB object modeling tool (ODM). Facebook's React.js is one of the most popular front-end development frameworks. Socket.io provides realtime, bi-directional communication between clients and servers. And finally, react-boilerplate is a structured project to be used as a starting point for developing react applications.

[1] Data gathered on April 5, 2020

Table 1: Repositories used in this study.

Repository	Commits	Used by	Bugs
Applications			
atom/atom	37.5k	-	11
mozilla/pdf.js	12k	100	20
impress/impress.js	383	90	1
videojs/video.js	3.5k	7.5k	13
Tools			
babel/babel	13.6k	2.2M	7
npm/cli	9k	150k	52
Libraries & Frameworks			
jashkenas/backbone	3.3k	-	16
jquery/jquery	6.5k	395k	49
koajs/koa	1k	111k	0
Automattic/mongoose	12k	938k	14
facebook/react	13k	3.4M	6
socketio/socket.io	1.7k	1.7M	7
react-boilerplate/react-boilerplate	1.4k	-	8

2.3 Bug Extraction

To gather string-related bugs from the development histories of the projects in Table 1, we semi-automatically filter the commits made to these repositories. The filtering is guided by the hypothesis that most string-related bugs are fixed by changing just a few lines of code. Section 3.3.1 provides evidence to support this hypothesis. After cloning the repositories, we automatically extract all commits that modify at least one JavaScript file and filter the commits by the number of changed lines, i.e., the sum of added and removed lines. We keep all commits up to four changed lines of code, resulting in 11,875 total commits.

We inspect each of the commits manually to identify bug fixes. A commit is considered a bug fix if it explicitly mentions that a programming mistake is corrected, e.g., by referring to an issue tracker or by describing the mistake in the commit message. In case of doubt whether a change is indeed a bug fix, or when we do not fully understand what the fix is about, we do not include a commit, to ensure that our study focuses on actual string-related bugs. Across all studied projects, this process results in 204 string-related bugs. Table 1 gives the per-project breakdown, showing that the bugs cover all but one project. For addressing the individual research questions, the 204 bugs are further inspected manually and also analyzed automatically, as explained in detail in Section 3.

Data availability. The full list of all studied bugs, along with the results of our manual inspection, is publicly available.[2]

2.4 Categorization

To answer RQ 1 and RQ 2 the authors inspected each bug and determined the category of root cause or consequence from personal judgment. The process included discussions on complex cases or disagreements, and the categories were finalized when the authors agreed on the same set or not categorized at all, otherwise. Both authors reviewed the dataset again after the above process was finished.

3 RESULTS

3.1 RQ 1: Root Causes

What causes string-related bugs? Understanding the root causes of these bugs is an important first step toward techniques that help developers find and fix the bugs. Furthermore, knowing how prevalent specific kinds of string-related bugs are helps prioritizing efforts toward such techniques.

To better understand the root causes of string-related bugs, we identify recurring patterns of root causes, categorize these patterns, and investigate how many of all studied bugs match these patterns. Figure 1 shows the resulting taxonomy of root causes. The number given for each leaf node in the taxonomy is the number of bugs that match the given pattern. The patterns are not mutually exclusive, i.e., a bug can have multiple of the root causes, and some bugs do not match any recurring patterns. The following describes the root cause patterns in more detail and illustrates them with examples.

3.1.1 *Bugs in String Literals.*

The overall most prevalent class of bugs are incorrect string literals. For each of these bugs, the incorrect code contains a hard-coded string that has been modified by the developers to address the bug. We distinguish two subgroups based on what exactly is wrong about the string literal.

Incorrect string literal. These bugs, which account for 85 (42%) of all studied bugs, are due to a mistake in a specific string literal. For example, these bugs include typos in a string literal, using one string literal that is also used in the program instead of the correct one, and string literals that are missing some information or that include incorrect information.

As a concrete example, consider this example from the Mongoose project:

```
-  const City = db.model('City', new Schema({
+  const City = db.model('City2', new Schema({
```

Another example is from the npm package manager:

```
  if (!url) {
-   url = "https://npmjs.org/package/" + d.name
+   url = "https://www.npmjs.org/package/" + d.name
  }
```

[2]https://github.com/sola-st/string-bugs

In both cases, the developers fix the bugs by modifying the string literal in place, without touching any of the other code, which is common for this bug pattern.

Should not use string literal at all. The second kind of string literal bugs occurs four times, i.e., in 2% of all studied bugs. Here, the incorrect code uses a string literal in a context where no literal should be used at all. Instead, the code already contains a property, a function, or a configuration value that yields the string to use. Hard-coding a specific string instead may lead to the wrong string being used, e.g., when the value needs to be updated.

This is an example of such a bug from the Atom text editor:

```
-  const executablesToSign = [ path.join(packagedAppPath,
     'Atom.exe') ]
+  const executablesToSign = [ path.join(packagedAppPath,
     CONFIG.executableName) ]
```

As in the example, such bugs are typically fixed by replacing the string literal with a reference to the property, function, or configuration value that yields the correct string.

Bugs caused by incorrect string literals are hard to find, both by humans and automated tools. For humans, the main challenge is to keep track of the different string literals spread across a code base and the (typically implicit) consistency constraints between them. For automated program analyses, the main challenge is to reason about the content of string literals. Many existing analyses (see Section 6 for some exceptions) do not even attempt to reason about string literals, but simply abstract their content away. Reasoning about the semantics of string literals, and how they related to their surrounding code, is a promising direction for future work.

3.1.2 *Bugs in Regular Expressions.*

Another highly prevalent class of string-related bugs are bugs in regular expressions. Prior work reports regular expression to be widely used [10] yet poorly tested [47]. We find 75 out of the 204 studied bugs (37%) to be within a regular expression, which confirms the earlier results and shows that the poor state of testing regular expression results in various bugs.

To get an impression of what kinds of mistakes developers typically make when writing regular expressions, Table 2 lists six recurring root cause patterns among the 75 regular expression bugs. As for our overall classification, these patterns are not mutually exclusive. The first two patterns are both the result of forgetting to consider a specific case when designing a regular expression. In pattern 1, the regular expression explicitly lists cases separated by `|`. In pattern 2, the programmer searches and replaces particular substrings in a given string using the `replace()` API, but forgets to search for a specific kind of substring. Together, these two patterns account for 38% of all regular expression bugs in our dataset. For both patterns, the bug is typically fixed by adding the missing case, as illustrated in the examples in Table 2.

Three of the remaining patterns are about incorrectly using, or not using at all, a specific feature of the regular expression syntax. In pattern 3, some part of the regular expression should be optional or repeated, which the developer forgets to express using the `?`, `{}`, or `*` syntax. Pattern 5 is the result of not anchoring the regular expression to the start or end of the matched string using the `^` and `$` symbols. Without such anchoring, the regular expression may match any substring of the given string, which sometimes is not

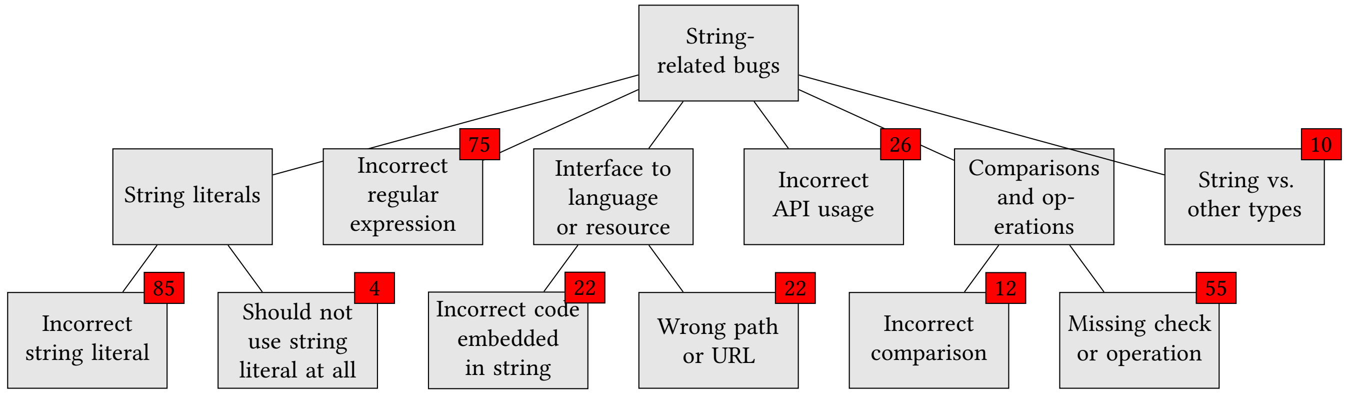

String-related bugs
String literals
Incorrect regular expression
75
Interface to language or resource
Incorrect API usage
26
Comparisons and operations
String vs. other types
10
Incorrect string literal
85
Should not use string literal at all
4
Incorrect code embedded in string
22
Wrong path or URL
22
Incorrect comparison
12
Missing check or operation
55

Figure 1: Common root causes of string-related bugs.

Table 2: Recurring bug patterns related to regular expressions.

Id	Bug Pattern	Example	Occurrences
1	A case is missing in a list of alternatives.	`- loop|o(?:ff?|[rn])`	15
		`+ loop|for|o(?:ff?|[rn])`	
2	Incorrect chain of `replace` calls.	`- n.replace(/[,\_]/g, "-")`	15
		`+ n.replace(/[,\_]/g, "-").replace(/\s/g, "")`	
3	Incorrect or missing use of optional or repeated pattern.	`- /(\d\.\d)\.\d/`	14
		`+ /(\d+\.\d+)\.\d+/`	
4	Mistake in literal part of a pattern.	`- /Edge/i`	7
		`+ /Edg/i`	
5	Missing anchors (start and end of string).	`- /---([\s\S]+)---/g`	7
		`+ /^---([\s\S]+?)---/g`	
6	Character should be escaped.	`- /[^/]$/`	5
		`+ /[^\/]$/`	

the intended behavior. In pattern 6, the developer forgets to escape a character that has a special meaning in the regular expression syntax of the underlying programming language, such as the slash character.

Finally, pattern 4 is about mistakes in a literal part of a regular expression, i.e., a sequence of characters that do not have any special meaning in regular expressions, but simply match as they are. These bugs are similar in nature to incorrect string literals, and techniques to detect one may also help detecting the other.

The high total number of bugs in regular expressions and the fact that a significant fraction of them fall under recurring patterns motivates work on techniques to detect and repair such bugs. While some techniques have been proposed in the past, e.g., a type system for regular expressions [38] and a visualization tool [6], there clearly is a need for additional testing and analysis techniques for regular expressions.

3.1.3 *Interface to Other Languages and External Resources.*

One important role of strings in programming is that they often serve as an interface to other (programming) languages and external resources. For example, a program may create the source code of another program, manipulate a document described in a markup language, or issue operating system-level commands that refer to paths in the file system. Because the well-formedness and content of such strings is typically not checked before the execution, bugs may easily arise. In our study, we find two common patterns of such bugs.

Incorrect code embedded in string. This class of bugs is caused by embedding code in one language as a string into another language. The embedded code may, e.g., be HTML code embedded into a UI component or command line arguments embedded into a build script. We find 22 (11% of all studied bugs) to belong to this class of bugs. The following is an example from Facebook's React library:

```
  ReactDOM.render(
-    <div className="pgErr">{err.toString()}</div>,
+    <pre style={{overflowX: 'auto'}} className="pgErr">{
    err.toString()}</pre>,
    mountNode
  );
```

Wrong path or URL. When programs refer to resources stored on the same or another machine, they often encode a path or URL into a string. We find that 22 (11%) of the studied bugs are due to a wrong path or URL. The following example is a bug that occurred in the Mongoose tool:

```
-  require('../lib/utils').random()
```

```
+   require('../../lib/utils').random()
```

In general, a wrong path or URL may be the result of an incorrect string literal, as in this example, or of a mistake in manipulating a string.

Detecting bugs related to interfacing with another language or external resources seems possible because the syntax and semantics of the content of the string is well-defined. For example, paths that refer to the local file system could be checked, and some lint-like static checkers reason about local paths. Another approach is to prevent such bugs through programming language or API design, e.g., in the form of embedded DSLs or APIs that allow programmers to construct code while providing more safety guarantees than when creating a raw string. Despite the fact that at least some of these bugs seem in reach, the high number of bugs in this category suggests that more effective techniques are needed.

3.1.4 *Incorrect (Usages of) String APIs.*

To create, manipulate, and validate strings, developers often use built-in APIs. We find 26 bugs (13%) caused by incorrect usages of string APIs. These bugs are often caused by using an API that is not appropriate for the intended purpose, by using the API incorrectly, or by combining multiple API calls in an unexpected way.

The following example is a bug in the npm package manager, where the programmer accidentally used one API instead of another:

```
-   && !!p.substr(0, -1).match(re[pattern]) )
+   && !!p.slice(0, -1).match(re[pattern]) )
```

Another example is from Mozilla's PDF.js, where the programmer expected the `toLowerCase()` API to modify a string in place, while it actually returns the modified string:

```
-   ch.toLowerCase();
+   ch = ch.toLowerCase();
```

Since many clients of popular string APIs exist, techniques for bug detection and repair based on API usage mining [2, 26, 32] could help find such bugs.

3.1.5 *Comparisons and Operations that Involve Strings.*

Another common root cause of string-related bugs is mistakes in comparing strings and in operations applied to strings. We distinguish the following two subcategories.

Incorrect comparisons. We find 12 bugs (6%) that are caused by an incorrect comparison of two strings. Half of them are related to the case-sensitivity of the comparison, such as the following example from the Socket.IO framework:

```
-   this.req.headers.upgrade !== 'websocket'
+   this.req.headers.upgrade.toLowerCase() !== 'websocket'
```

Some of the comparison-related bugs occur when checking some string variable against an expected valued, e.g., in testing code, such as this example from the npm package manager:

```
-   t.equal(stdout, 'cool\t\t\tprehistoric\t\t\nfoo')
+   t.equal(stdout, 'cool\t\t\tprehistoric\t1.0.0\t\nfoo')
```

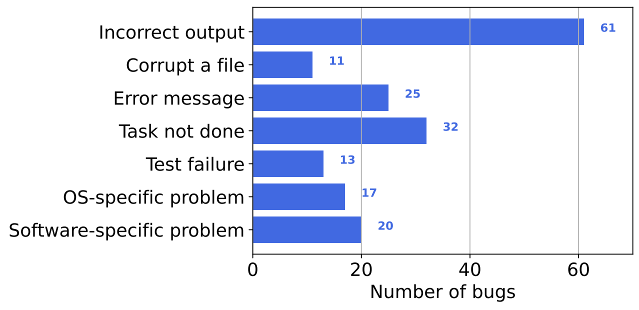


Figure 2: Consequences of string-related bugs.

Missing check or operation. 55 (27%) out of the studied bugs are caused by a missing check or operation related to strings. These bugs include code where only part of the expected string manipulation is performed or where the programmer forgot to guard some operation by an appropriate check. For example, the following bug occurred in Mozilla's PDF.js due to a missing call to `trim()`:

```
-   info.Version + ' ' + (info.Producer || '-')
+   info.Version + ' ' + (info.Producer || '-').trim()
```

3.1.6 *Strings vs. Other Types.*

The final class of root causes is about treating a value as a string, and not some other type. Among the studied bugs, 10 (5%) fall into this category. For many of these bugs, some variable is not of string type, but the programmer expects it to be a string. Typically, these bugs are fixed by converting the value into a string, which in JavaScript can be achieved by concatenating a value with a string, as in the following example from jQuery:

```
-   s.data = s.data.replace(jsre, "=" + jsonp);
+   s.data = (s.data + "").replace(jsre, "=" + jsonp);
```

Such type-related bugs could be detected by static type checking, e.g., using gradual type checkers for dynamically typed languages, such as Flow for JavaScript, or Pyre and Mypy for Python.

3.2 RQ 2: Consequences

The goal of this research question is to understand the consequences of string-related bugs, i.e., what kinds of misbehavior they cause and how the bugs manifest. Understanding the consequences will help determine effective techniques for finding string-related bugs, in particular, testing-based and dynamic analysis-based techniques that rely on a manifestation of misbehavior.

3.2.1 *Common Kinds of Consequences.*

Figure 2 shows the most common kinds of consequences of string-related bugs. We identify these consequences by carefully studying the code in which each bug occurs and issue descriptions associated with a bug. As for RQ 1, the categories for consequences are not mutually exclusive, i.e., a bug may appear in multiple categories.

Incorrect output. Many of the string-related bugs (61, i.e., 30%) cause the program to produce an incorrect output, such as an incorrect text printed to the console or an incorrect UI element rendered, e.g., on a website. While incorrect outputs are often easy to spot for humans, identifying them in an automated tool is much more challenging, because the expected output often is not specified. One promising direction for automatically detecting incorrect outputs is cross-checking different, supposedly consistent outputs [14].

Corrupt a file. There are 11 bugs (5% of the studied bugs) that cause the program to write incorrect data into a file or to store data into a wrong file. Both are severe consequences, as they permanently corrupt the state. From a bug detection perspective, these bugs are similar to the "incorrect output" category discussed above.

Error message. Out of all the bugs found in our study, 23 bugs (11%) lead to an error message being printed, e.g., to the command line or the browser console. This relatively low percentage is likely to be influenced by the "no crash philosophy" of JavaScript [3], where many potential runtime mistakes do not cause errors but are silently handled by the language, e.g., through implicit type conversions [33]. From a bug detection perspective, the lack of obvious error messages for most bugs poses an interesting challenge.

Task not done. A significant number of bugs (32, i.e., 16%) results in a specific task or expected behavior not being done. For example, these bugs include code that accidentally fails to download some input data, fails to sanitize a string in some cases, or ignores some of the give input files. Such omission errors are difficult to detect because nothing wrong happens, but the correct behavior simply does not happen.

Test failure. Among the studied bugs, 13 bugs (6%) manifest through a test failure. Because we gather the studied bugs from bug fixes in version histories, the dataset misses problems that developers fix locally, before committing to the shared repository. These missed bugs are likely to include some bugs exposed by test failures triggered when a developer runs the test suite locally.

OS-specific problem. Some bugs manifest only on specific operating systems, e.g., because they relate to how file system paths are encoded or how whether some external tool is available. We find 17 such OS-specific bugs (8%). Detecting such bugs through any kind of runtime analysis requires the program to be executed on a specific operating system, which increases the bug detection cost in practice.

Software-specific problem. Similar to the above category, some bugs manifest only when the code is executed within or in combination with a specific other software. These bugs include browser-specific bugs that, e.g., manifest only in some versions of Internet Explorer. We find 20 such bugs (10%) in our dataset. Combining this and the previous category of consequences, 37 bugs (18%) manifest only in a specific environment.

3.2.2 Relation between Root Causes and Consequences. Our classification of bugs based on their root causes and consequences allows for studying the relation between both. Figure 3 shows how much bugs with a specific kind of root cause lead to a specific kind of consequence. There are a few noteworthy relations. First, we observe that incorrect regular expressions often lead to incorrect output. Almost half of all regular expression-related bugs have this consequence. Second, test failures are frequently caused by incorrect literals. The reason is that literals often appear in test cases as hard-coded inputs or as part of assertions about the output of the tested code. Finally, we find that bugs caused by treating a non-string

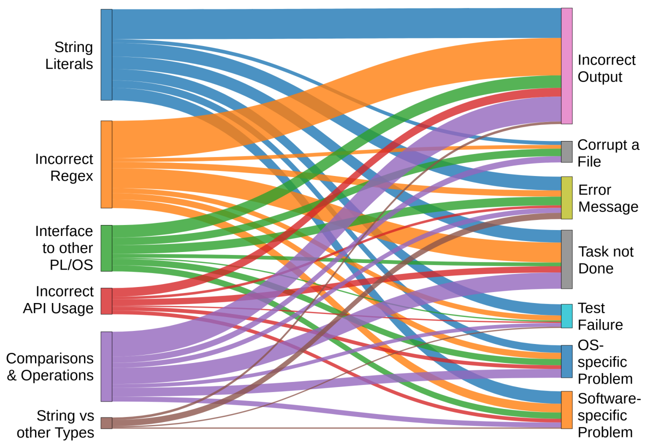


Figure 3: Root causes leading to consequences.

as a string, or vice versa (category "String vs. other types") often manifest through an error message. We attribute this observation to the fact that JavaScript raises errors for some type-related mistakes, e.g., when trying to call a non-existing method of an object.

Overall, we conclude that string-related bugs are not limited to a few kinds of consequences, but manifest on a variety of ways. The fact that 18% of all bugs manifest only in a specific environment implies that dynamic analysis-based approaches for finding string-related bugs should execute the code under test in different environments. Another interesting finding is that the vast majority of bugs does not lead to an obvious sign of misbehavior, such as an error message, but rather manifest through the absence of some expected behavior or an incorrect output. This finding calls for more work toward implicit oracles, e.g., through differential testing [30], metamorphic testing [12, 35], or some kind of consistency check.

3.3 RQ 3: Fixes

This research question is about the way developers fix string-related bugs. Specifically, we are interested in the size of bug fixes (Section 3.3.1) and in the degree to which the "ingredients" for a bug fix are available in the buggy code (Section 3.3.2).

3.3.1 Size of Fixes. We measure the size of bug fixes in two ways. Figure 4 shows how many lines a bug fix includes, where each removed and added line counts separately. That is, modifying a single line counts as two lines in the bug fix, as one line is removed and another line is added instead. The figure shows that most fixes of string-related bugs (125 out of 204, i.e., 61%) affect only a single line, i.e., they have one or two lines in the bug fix. When interpreting these results, one needs to consider that the bugs considered in the study are filtered by the number of changed lines (at most four added and removed lines, Section 2.3), i.e., our dataset is inherently biased towards small bug fixes. Nevertheless, the fact that most bugs are even smaller than our limit imposed when gathering bugs shows that many string-related bugs indeed have small fixes.

The second measurement is about the number of characters changed in a bug fix. For each bug fix, we take the sequence difference, using Python's difflib library, between the added and removed

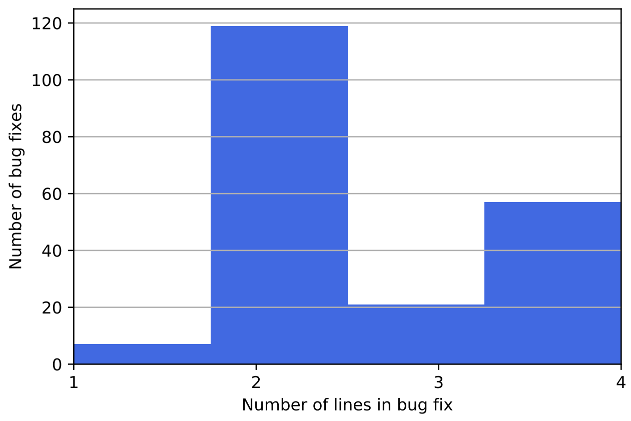


Figure 4: Histogram of lines changed in bug fixes.

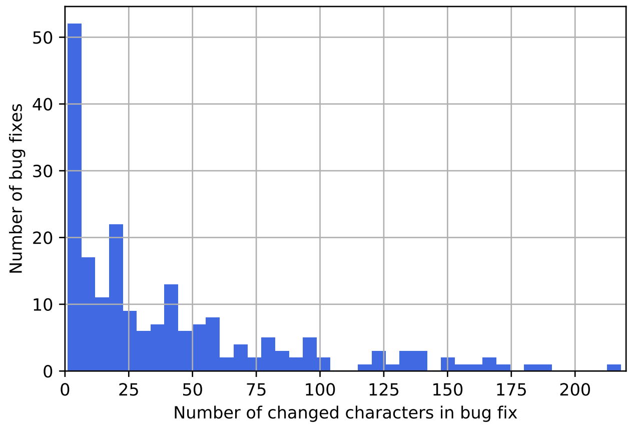


Figure 5: Histogram of characters changed in bug fixes.

lines to detect the minimum required character change. We then use the sum of added, removed, and modified characters as the total number of changed characters. As an example, this is the output of the sequence difference script for a bug in the JQuery library.

```
- -     var m = er.message.match(/^E[A-Z]+/)
? ^
+ +     var m = er.message.match(/^(?:Error: )?(E[A-Z]+)/)
? ^                                 +++++++++++++       +
- -       m = m[0]
? ^             ^
+ +       m = m[1]
? ^             ^
```

Using the markers provided by the difflib tool, we can easily count the number of modified characters. Figure 5 shows the histogram of the number of changed characters. The results confirm the observation that most string-related bugs require only small fixes. For example, 111 out of the 204 bugs (54%) are fixed by changing fewer than 30 characters.

3.3.2 Fix Ingredients. The fact that string-related bugs follow recurring patterns and that they often get fixed by modifying just a single line of code may bring this class of bugs into the reach of automated program repair [25]. One challenge in automated repair is to find the program code that is required in the fix but not part of the buggy code location, sometimes called "fix ingredients". To better understand the potential for finding such fix ingredients for string-related bugs, we analyze whether the tokens added when fixing a bug are available close to the location where the string-related bug gets fixed.

In the first step, we compare the old and the new code that fix a string-related bug by extracting the tokens in these code fragments. We then search up to a 100 lines before and 100 lines after the bug for these tokens and measure how many of the otherwise missing tokens are within this window of nearby available fix ingredients. Not all bug fixes add tokens, e.g., some fixes require removing entire lines or specific tokens from the source code. Our analysis does not consider these bug fixes (39 out of 204), since all ingredients for fixing these bugs are given in the buggy code. The analysis is automated and implemented based on the Acorn JavaScript parser[3] and the Difflib.js[4] package, which is ported from Python's difflib library.[5]

We present two variants of the fix ingredients results. Figure 6a includes all tokens except for regular expressions, which are represented as a single token. The figure shows that 25% of the bugs have all of the required ingredients present nearby, and 43% of the bugs have at least 70% of the tokens needed for the fix around the bug. Figure 6b shows the same results, now including regular expressions. Here, 12% of the bugs have all required tokens in the vicinity and 30% of the bugs have at least 70% of their repair ingredients close by. The difference between these two figures is due to the fact that the chance of having the exact regular expression required to fix a bug somewhere in the code is small. Yet, both figures show that for a significant fraction of string-related bugs, all or at least many of the fix ingredients are available in the surrounding code.

Overall, the results of RQ 3 show that many string-related bugs are fixed by modifying very little code, often just a single line, and that many of the fix ingredients are available in the surrounding code. For example, for this bug in the React-boilerplate project, the entry point path is wrong.

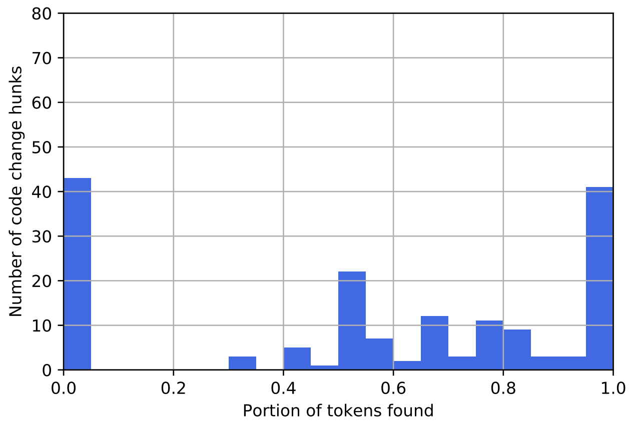
```
console.log(path.join(__dirname, '..', 'app/js/app.js'));
...
  entry = [
-   path.join(__dirname, '..', 'js/app.js') /// Start
     with js/app.js...
+   path.join(__dirname, '..', 'app/js/app.js') /// Start
      with js/app.js...
```

Notice that a few lines above, our script found the correct path used in a log message, which could potentially be used to repair this bug automatically. These findings bring string-related bugs into the reach of automated program repair, motivating future work in this direction.

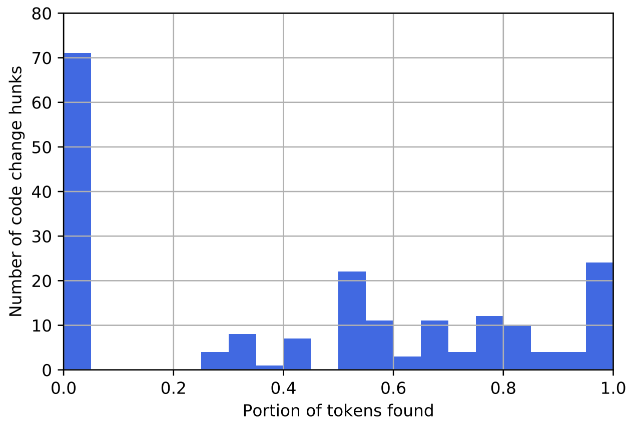
3.4 RQ 4: Program Components

This research question asks what parts of a software project are most affected by string-related bugs. We address this question by studying for each bug the code location where the bug is fixed,

[3]https://github.com/acornjs/acorn/
[4]https://www.npmjs.com/package/difflib
[5]https://docs.python.org/3/library/difflib.html



(a) not counting regular expressions



(b) counting regular expressions

Figure 6: Histogram for the portion of repair ingredients present in +/- 100 lines of the bug.

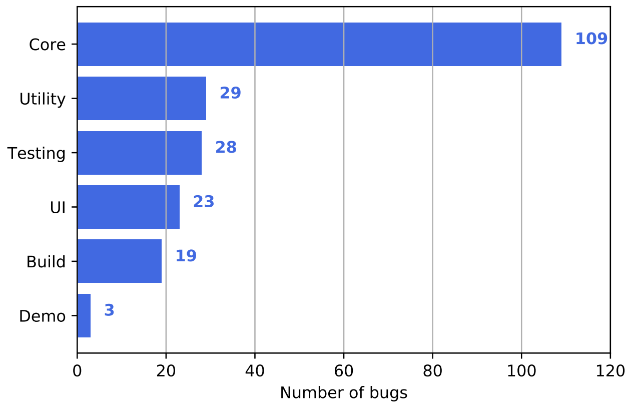


Figure 7: Program components affected by string-related bugs.

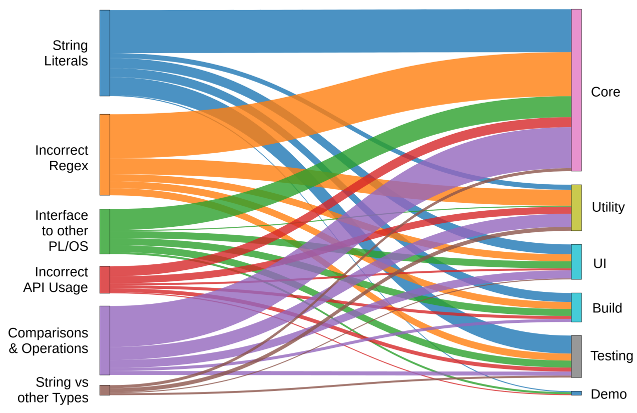


Figure 8: Root causes affecting components.

and by classifying these code locations into six categories: the *core* functionality of the project, *utility* functions, *UI* components, *testing* code, *build* scripts, and *demo* applications or examples for how to use an API.

Figure 7 shows the results of our analysis. The overall finding is that string-related bugs affect all kinds of code components, and are not limited to, e.g., UI or testing code. The most commonly affected parts of the studied projects are the core components, where 109 of the bugs (53%) reside.

To assess whether the components where bugs resides relate in any particular way with the root causes of bugs, Figure 8 shows how many bugs with a specific root cause are in a specific component. For most kinds of root causes and components, we do not see a clear relation between root causes and where a bug resides. The only exceptions are bugs caused by incorrect string literals, which tend to appear relatively often in testing code. The likely reason again is that hard-coded strings are common in testing code. The absence of strong relations between root causes and components implies that bug detection tools that target a specific kind of root cause should be applied across all components of a software.

Overall, we conclude from the results in RQ 4 that string-related bugs are wide-spread, even in the most important parts of code bases, and hence deserve appropriate attention.

3.5 RQ 5: Existing Bug Detection Tools

The following addresses the question of how effective existing bug detection tools are at finding string-related bugs. There are many bug detection tools, e.g., FindBugs [20], Google's Error Prone [1], or Facebook's Infer [8]. For JavaScript, static linters are commonly used in practice [41], and we therefore compare with JSHint[6], one of the most popular static linters for JavaScript. To check which of the bugs studied in this paper are detected by JSHint, we apply the tool to each of the buggy files just before the bug fix was applied. We then compare any warnings reported by JSHint close to the locations that were fixed to address the bug, focusing on warnings in the line directly before, exactly at, or directly after a modified line.

[6]http://jshint.com

In total, there are 218 lines with a JSHint warning close to a bug fix location. A manual inspection shows that almost all of them are unrelated to the string-related bugs, but point to other potential problems at the same code location. The majority of these unrelated warnings are about missing semi-colons (which is legal but discouraged in JavaScript), language features supported only by ECMAScript 6 (which is not yet supported by all browsers), and possibly undefined references to global variables available only in the browser (but not, e.g., in Node.js). Only one out of the 218 warnings points to one of the string-related bugs. It warns about an invalid regular expression, which was fixed by the developer.

Overall, we conclude that a widely used static bug detection tool is not effective at detecting string-related bugs, offering ample of opportunities to improve the state of the art.

4 THREATS TO VALIDITY

Subject projects and bugs. Our study focuses on 204 bugs gathered from 13 JavaScript repositories. While we aim at a diverse set of projects, they may not be representative for other projects. Because the study focuses on JavaScript only, our conclusions may not hold for other languages. The selection process of bugs may have introduced some bias, e.g., toward bugs that are locally fixed by changing a few lines of code. Hence, our findings apply to such bugs only, and more complex string-related bugs may have other properties. Finally, because we study bugs fixed through commits, our dataset misses bugs that never appear in the version control system, either because the developer fixes them even before committing or because the bugs remain unnoticed.

Manual classification. Our answers for RQ 1, 2, and 4 rely on a manual inspection and classification of the 204 bugs. While such a manual process is necessary to identify non-trivial patterns, it may introduce mistakes. To mitigate this potential threat, the classification results were checked by two authors and iteratively refined until reaching agreement.

5 DISCUSSION

The fact that string-related bugs often affect the core components of a project and lead to hard-to-detect misbehavior, such as incorrect output or corrupted files, raises the question of how developers can find such bugs quicker. Unfortunately, developers currently have access to little or no effective tool support for dealing with string-related bugs. Most program analyses do not reason about the values of strings, but simply abstract these values away. How to develop effective tools for finding string-related bugs needs more research. NLP-based reasoning about the content of strings may be a promising, yet mostly unexplored, direction. The high number of regular expression bugs, combined with the wide use of regular expressions [10], suggests that mining-based or learning-based approaches to find regular expression bugs could be a fruitful research direction. Our findings about recurring root cause patterns (RQ 1) will help guide such efforts toward the most promising kinds of string-related bugs.

While string-related bugs are non-trivial to detect, fixing them often involves only small and local code changes. This finding motivates work on automated repair of such bugs. A common challenge in automated repair is how to find repair ingredients, i.e., code fragments needed in the corrected code. Our results for a show that for many string-related bugs, the repair ingredients are available, waiting to be exploited by appropriate repair tools.

6 RELATED WORK

6.1 String-related Bugs

Even though there is no comprehensive study of string-related bugs yet, prior work has looked into specific kinds of string-related code. Regular expressions are shown to be regularly used by programmers [10], and at the same time, tested less well than other code [47]. Our finding that many string-related bugs are caused by incorrect regular expressions show the natural consequence of the two previous results. Recently, Wang et al.[46] have done an empirical study on regular expression bugs, similar to Section 3.1.2 in our study. The results in [46] are close to what we found about regular expression bugs, although the repositories and programming languages in that study are different. To avoid such bugs, visualization [6] and static type checking [38] of regular expressions have been proposed. String-related bugs sometimes cause security vulnerabilities, e.g., format string vulnerabilities [16, 37], cross site scripting vulnerabilities [5], or regular expression denial of service vulnerabilities [17, 39]. Compared to other string-related bugs, such vulnerabilities have been studied intensively, and there are techniques to detect and fix them. Our study shows that there are many string-related bugs beyond vulnerabilities, for which very little tool support exists.

6.2 Program Analysis of String-related Code

There are several techniques for reasoning about strings in a program analysis. Christensen et al. [15] propose an analysis of string expressions that tries to create a regular language describing the values a string may have. String solvers, such as Hampi [23] and Rex [43], help expressing and solving constraints gathered during the analysis of string-manipulating programs, which can be used, e.g., to create injection attacks [24]. These techniques help reasoning about the question of what values a string expression may have, which is an important part of better tools to detect string-related bugs.

6.3 Natural Language Information in Code

Strings often contain some form of natural language information. Some program analyses use such natural language information, e.g., to infer specifications from API documentation [7, 49], to find inconsistencies between comments and code [40], to warn about potentially incorrect combinations of identifier names [34], and to predict types [29]. However, none of these techniques reasons about natural language information within strings, which could be interesting future work to find some of the bugs studied in this paper.

6.4 Studies of Bugs

Other studies investigate bugs in specific application domains, e.g., in operating systems [13], in deep learning systems [21, 48], and in blockchain code [44], or specific classes of bugs, e.g., concurrency bugs [28], bugs in test code [42], long-lasting bugs [11], and performance bugs [19, 22, 27]. Prior studies of bugs in JavaScript focus on client-side JavaScript bugs [31], performance bugs in JavaScript [36], and bugs that occur on the Node.js platform [45]. Our work contributes an in-depth analysis of the important yet currently understudied class of string-related bugs. A study by Habib and Pradel [18] shows that state-of-the-art static bug detection tools miss the majority of bugs that occur in the wild. Our findings about lint-like tools for JavaScript reinforces this finding for string-related bugs.

7 CONCLUSION

This paper presents the first in-depth empirical study of string-related software bugs. These programming mistakes are a class of simple yet prevalent bugs that we find to affect all kinds of components of software projects, including the core functionality. Our study investigates 204 string-related bugs gathered from 13 popular JavaScript projects. We find that many bugs are instances of recurring root cause patterns, with incorrect string literals (42%) and incorrect regular expressions (37%) being the most common root causes. While string-related bugs tend to require only small fixes (61% are fixed by modifying just one line), they can have severe consequences (30% lead to incorrect output) and are non-obvious to detect (only 11% lead to an error message and 18% manifest only in specific environments). A widely used static checker for JavaScript misses all but one of the 204 studied bugs. In general, string-related bugs are difficult to find with program analyses because most analyses do not reason about the values of strings.

Our dataset is made available as a reference for future work on best practices for developers, bug detection tools, automated repair tools, and the design of string-related APIs and programming language features. In particular, we hope to inspire work on developer tools to find and fix string-related bugs. The current state of the art leaves a huge untapped potential for techniques that detect such bugs, and the root cause patterns described in this paper will guide such efforts toward the most promising kinds of bugs. Moreover, we find string-related bugs to be in reach for automated program repair: Not only are many fixes simple, but for many bugs, the fix ingredients, i.e., tokens added in a fix, are available in the vicinity of the fix location.

REFERENCES

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012.* 14–23.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 4–16.

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* (2017).

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.

[5] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti (Eds.). ACM, 91–100. https://doi.org/10.1145/1772690.1772701

[6] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: visual debugging of regular expressions. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 504–507. https://doi.org/10.1145/2591062.2591111

[7] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[8] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Vol. 9058. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[9] Michael Carraz, Konstaninos Korakitis, Peter Crocker, Richard Muir, and Christina Voskoglou. 2020. Developer Economics: State of the Developer Nation. SlashData, Ltd..

[10] Carl Chapman and Kathryn T. Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 282–293. https://doi.org/10.1145/2931037.2931073

[11] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. 2014. An empirical study of dormant bugs. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, 82–91. https://doi.org/10.1145/2597073.2597108

[12] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong âĂę.

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. 2001. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 73–88. https://doi.org/10.1145/502034.502042

[14] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated identification of cross-browser issues in web applications. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania.* IEEE Computer Society, 1–10. https://doi.org/10.1109/ICSM.2010.5609723

[15] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 2694. Springer, 1–18. https://doi.org/10.1007/3-540-44898-5_1

[16] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, Dan S. Wallach (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/sec01/cowanbarringer.html

[17] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12.* USENIX.

[18] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *ASE.*

[19] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *ESEM.*

[20] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, 132–136.

[21] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *FSE.*

[22] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Conference on Programming Language Design and Implementation (PLDI).* ACM, 77–88.

[23] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, Gregg Rothermel and Laura K. Dillon (Eds.). ACM, 105–116. https://doi.org/10.1145/1572272.1572286

[24] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. In *ICSE.* 199–209.

[25] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[26] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 306–315.

[27] Yepang Liu, Chang Xu, and S.C. Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*. 1013–1024.

[28] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 329–339.

[29] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

[30] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[31] Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*. 55–64.

[32] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *International Conference on Automated Software Engineering (ASE)*. 371–382.

[33] Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

[34] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[35] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[36] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[37] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David A. Wagner. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, Dan S. Wallach (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/sec01/shankar.html

[38] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, Wei-Ngan Chin and Aquinas Hobor (Eds.). ACM, 20–26. https://doi.org/10.1145/2318202.2318207

[39] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium*. 361–376.

[40] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 145–158. https://doi.org/10.1145/1294261.1294276

[41] Kristín Fjóla Tómasdóttir, Mauricio Finavaro Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 578–589. https://doi.org/10.1109/ASE.2017.8115668

[42] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 101–110. https://doi.org/10.1109/ICSM.2015.7332456

[43] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic Regular Expression Explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 498–507. https://doi.org/10.1109/ICST.2010.15

[44] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 413–424. https://doi.org/10.1109/MSR.2017.59

[45] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 520–531.

[46] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. [n.d.]. An Empirical Study on Regular Expression Bugs. ([n. d.]).

[47] Peipei Wang and Kathryn T. Stolee. 2018. How well are regular expressions tested in the wild?. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 668–678. https://doi.org/10.1145/3236024.3236072

[48] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. https://doi.org/10.1145/3213846.3213866

[49] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming (ECOOP)*. 318–343.