ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries

Jibesh Patra
TU Darmstadt
jibesh.patra@gmail.com

Pooja N. Dixit
TU Darmstadt
poojandixit@gmail.com

Michael Pradel
TU Darmstadt
michael@binaervarianz.de

ABSTRACT

It is a common practice for client-side web applications to build on various third-party JavaScript libraries. Due to the lack of namespaces in JavaScript, these libraries all share the same global namespace. As a result, one library may inadvertently modify or even delete the APIs of another library, causing unexpected behavior of library clients. Given the quickly increasing number of libraries, manually keeping track of such conflicts is practically impossible both for library developers and users. This paper presents ConflictJS, an automated and scalable approach to analyze libraries for conflicts. The key idea is to tackle the huge search space of possible conflicts in two phases. At first, a dynamic analysis of individual libraries identifies pairs of potentially conflicting libraries. Then, targeted test synthesis validates potential conflicts by creating a client application that suffers from a conflict. The overall approach is free of false positives, in the sense that it reports a problem only when such a client exists. We use ConflictJS to analyze and study conflicts among 951 real-world libraries. The results show that one out of four libraries is potentially conflicting and that 166 libraries are involved in at least one certain conflict. The detected conflicts cause crashes and other kinds of unexpected behavior. Our work helps library developers to prevent conflicts, library users to avoid combining conflicting libraries, and provides evidence that designing a language without explicit namespaces has undesirable effects.

ACM Reference Format:
Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180184

1 INTRODUCTION

The popularity of JavaScript has lead to the development of numerous JavaScript libraries. For example, a popular content delivery network that hosts JavaScript libraries provides over 3,000 different libraries.[1]. Libraries are ubiquitous and many applications use multiple libraries. One estimate is that 75% of the top 10 million websites use at least one of the top 18 libraries.[2] A recent study on the top 75,000 Alexa websites [29] reports that the number of externally hosted scripts that a website includes has a median of 9 and a maximum of 202.

```
// Strophe.js
window.Base64 = {
  encode: function
        (b) {
    /* code */
  }
  decode: function
        (b) {
   /* code */
  }
};
```

```
// JSEncrypt.js
window.Base64 = {
  unarmor: function
        (t) {
    /* code */
  }
  decode: function(
        i) {
    /* code */
  }
};
```

```
// Library client
jsEncrypt = new JSEncrypt();
jsEncrypt.setKey(...);

// Returns false instead of
// decrypted data when
// Strophe.js is loaded
// after JSEncrypt.js.
jsEncrypt.encrypt(...);
```

Figure 1: Example of two conflicting libraries and a client that will experience unexpected behavior when loading both libraries.

Unfortunately, using multiple independently developed libraries together may cause unexpected behavior. The reason is that JavaScript does not have namespaces but instead, all libraries share a single global namespace. As a result, a value or a function "exported" by one library may be easily overwritten, modified, deleted, or accidentally used by another library. Moreover, libraries may overwrite built-in APIs, sometimes called "monkey patching", and multiple libraries may try to overwrite the same API in different ways. In practice, the problem is compounded by the loose typing in JavaScript, which allows one library to overwrite another library's API even with a type-incompatible value.

As a real-world example of a library conflict found by our approach, consider Figure 1. The left side of the figure shows an excerpt of *Strophe.js*, a library that implements the XMPP middleware protocol. The center part of the figure shows *JSEncrypt.js*, a library that provides OpenSSL RSA encryption. Both libraries write to the global variable `Base64`.[3] When included together, the library that is included last will overwrite the `Base64` object of the library that was included first. Such overwriting may cause unexpected behavior in a client of either of these libraries. For example, the right side of the figure shows a client that tries to encrypt some data using *JSEncrypt.js*. When executing this client after loading only *JSEncrypt.js*, the last call returns the encrypted data. However, when executing the client after loading *JSEncrypt.js* and then *Strophe.js*, the last call simply returns `false`. The fact that including an apparently unrelated library breaks the core feature of the encryption library will surprise users and is unintended by the developers of both libraries.

Problems caused by conflicting libraries may occur whenever a developer loads two libraries, which is common practice. Even if a

[1] https://cdnjs.com/libraries

[2] https://w3techs.com/technologies/overview/javascript_library/all

[3] The `window` variable is the global object in client-side JavaScript.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180184

developer explicitly loads only one library, other libraries may be implicitly loaded. Due to the highly dynamic nature of JavaScript, where some code may dynamically load other code, an application developer may implicitly load libraries without even noticing it. For example, websites built on top of content management systems often use plugins, each of which implicitly loads some libraries.[4] Other common ways of implicitly loading libraries are third-party ads, social media services, and news feeds. When a conflict between libraries exists, JavaScript often follows a "no crash" philosophy, i.e., misbehavior may not lead to an exception. As a result, conflicts easily remain unnoticed at library load time or even later, until a user triggers the unexpected behavior, as illustrated in the motivating example.

In principle, there is a sane way for libraries to share the global namespace. Ideally, library developers all follow a "single API object" pattern, where the entire API of the library is encapsulated into a single object. The library then writes this object to a single global variable, e.g., named like the library itself, to minimize the potential for conflicts. In practice, not all libraries follow this pattern, and some global variables, such as $ and _, are particularly popular. Our empirical results show that 71% of all libraries do not follow the "single API object" pattern.

Library conflicts are challenging to detect for a program analysis and difficult to avoid for library developers. One reason is that unintended effects of conflicts typically manifest only at runtime. A purely static analysis can either soundly overapproximate potential conflicts and their effects, which is likely to produce a large number of false positives, in particular for JavaScript, or unsoundly underapproximate them, which may miss conflicts. Another challenge for detecting conflicts is the large number of JavaScript libraries. With thousands of libraries available, and new libraries being added and updated every day, analyzing all possible combinations of libraries leads to a combinatorial explosion that is prohibitive in practice. Currently, there exists no technique for library developers to check whether their library conflicts with another and for library clients to check which combinations of libraries to avoid. Furthermore, it is currently unknown to what extent the problem of library conflicts matters in practice.

This paper presents ConflictJS, the first automated and scalable technique that analyzes JavaScript libraries for conflicts. We address the huge search space of possible conflicts and the difficulties of statically analyzing JavaScript through a two-phase approach that combines dynamic analysis and test synthesis. In the first phase, ConflictJS dynamically analyzes individual libraries to detect writes to the global namespace while loading a library. An offline comparison of these global writes yields a set of potential conflicts between libraries. In the second phase, ConflictJS synthesizes and dynamically analyzes library clients to check if potential conflicts indeed lead to unexpected behavior. The second phase, and therefore also the overall approach, is precise in the sense that every validated conflict certainly occurs in the synthesized client and leads to different behavior depending on the loaded libraries.

We use ConflictJS to analyze and study 951 libraries. The results show that 268 (28%) libraries are potentially conflicting and that 166 (17%) libraries are certainly conflicting with at least one other library. The conflicts may lead to crashes, unexpected behavior, and globally reachable state with unexpected values and types. A manual analysis of conflicting libraries reveals several recurring patterns of root causes for conflicts, which are instructive for library developers, API designers, and language designers. We reported seven of the detected conflicts to the respective library developers, of which four already have been acknowledged and confirmed as problematic. Of the four conflicts, two have been fixed by the developers of the respective libraries.

Compared to existing work on analyzing JavaScript [3], our work is the first to address conflicts among libraries. Existing static analyses focus on type checks of single libraries [11] or assume the presence of library clients [30]. Existing dynamic analyses that target type inconsistencies [38] and other coding problems [17] assume to have inputs to exercise the program, whereas our work synthesizes library clients automatically. JSNose [10] identifies excessive uses of global variables, but focuses on single libraries. Finally, our empirical results relate to existing large-scale studies of JavaScript libraries and their usage [29, 34]. Our work is the first to study library conflicts.

We envision ConflictJS to be useful for developers of libraries and library clients alike. Library developers may use ConflictJS to check whether their library conflicts with others, allowing them to avoid the conflicts by adapting the library. Developers of library clients may use ConflictJS to check which libraries conflict, allowing them to avoid including them together.

In summary, this paper contributes the following:

- We are the first to address the problem of conflicts among libraries in a language without explicit namespaces.
- We present ConflictJS, an automated and scalable technique to precisely detect conflicts in JavaScript libraries through a combination of dynamic analysis and test synthesis.
- We provide empirical evidence that the approach scales to 951 libraries, where it effectively detects and validates 1,840 conflicts among them.
- We provide our implementation as open-source.[5]

2 MOTIVATION AND PROBLEM STATEMENT

This section provides some background, motivates the problem of conflicts among libraries with examples from real-world libraries, and formulates the problem addressed in this paper.

2.1 Background

The JavaScript version that is fully supported by most modern browsers is ECMAScript 5 [7]. It does not provide any kind of namespaces or modules at the language level. Instead, library developers rely on several ad-hoc mechanisms to encapsulate code and to export APIs. First, some libraries follow a "single API object" pattern, where the library initializes itself in a local scope and provides its API as properties of a single global object. The most obvious choice for naming this global API object is the name of the library, which typically is unique. For example, *react.js* follows this pattern by exporting its APIs into the global `React` object. The popular *jQuery* library furthermore enables developers to avoid conflicts

[4] For a real-world example, see http://simple-press.com/documentation/codex/faq/troubleshooting/what-is-this-jquery-conflict/.

[5] https://github.com/sola-da/ConflictJS

by specifying the global variable where to provide the library or to even export the library into an existing, non-global object.[6] Second, some libraries build upon the asynchronous module specification (AMD), a module system targeted at client-side JavaScript and implemented as a library, e.g., RequireJS[7]. Third, some libraries use CommonJS, a module system targeted at non-client-side JavaScript and implemented as the default module system on the Node.js platform. Unfortunately, these options are neither compatible with each other nor available on all JavaScript platforms. ECMAScript 6 [8] unifies ideas from CommonJS and AMD into language-level module support, and popular JavaScript platforms have started to adopt it. However, since widely used libraries cannot rely on recently added language features, they typically ensure backward compatibility by relying on other ways to export their APIs. In summary, the lack of namespace and modules in currently deployed versions of JavaScript creates a non-trivial problem for library developers.

2.2 Motivating Examples and Classification of Conflicts

The following section motivates the problem of conflicts between libraries with real-world examples (Table 1) found using our approach. Furthermore, we use these examples to define four classes of conflicts, based on how the conflicts manifest to a library client. For each example, we show code from two conflicting libraries and a client application that observes different behavior depending on which of the libraries are included and on the order of inclusion.

Inclusion Conflicts. This kind of conflict raises an exception when including multiple libraries, without any further interaction between the client and the libraries. The example in the first column illustrates the problem with the *curl* and *dojo* libraries. Loading the second library after loading the first library causes an exception. For a library user, finding such conflicts is non-trivial because the exception depends on the order of including the libraries: Only if a client loads *dojo* before loading *curl* the exception occurs. The documentation of neither of the libraries provides any reference to the other library, presumably because the respective developers are not aware of each other.

Type Conflicts. Type conflicts occur when multiple libraries write type-incompatible values to the same globally reachable location. Table 1 presents an example of two libraries, *ocanvas.js* and *aframe.js*, that write to `window.logs` an array and a function, respectively. A client using one of these libraries may rely on the type of the conflicting value and will be surprised if including another library or changing the order of library inclusion breaks the type assumption. This and the following kinds of conflict are more subtle than inclusion conflicts because they do not lead to an obvious error when simply including the libraries.

Value Conflicts. Similar to type conflicts, this kind of conflict is caused by multiple libraries writing different values to the same globally reachable location. We classify a conflict as value conflict if the values are type-compatible but different. Table 1 provides an example where two libraries, *pako* and *3Dmol*, write different values to the same variable `pako`. The root cause of this conflict is that *3Dmol* contains an outdated version of *pako*.

Behavior Conflicts. A behavior conflict occurs when multiple libraries store functions at the same globally reachable location, but these functions do not provide the same behavior. Table 1 presents an example where two libraries, *jsface* and *matreshka*, overwrite the same variable `Class`. As illustrated by the client code, the two functions provide different behaviors, which may surprise a client that is not aware of the fact that both libraries provide dissimilar implementations of the same global function.

2.3 Problem Statement

Based on these four types of conflicts, we now formulate the problem addressed in this paper. The input to our approach is a set $\mathcal{L}$ of libraries. We assume that each $l \in \mathcal{L}$ is supposed to be usable without including any other library in $\mathcal{L}$. In particular, this assumption excludes libraries that extend another library, e.g., libraries that extend the popular *jQuery* library with additional features.

Libraries are used by clients that interact with the APIs of a library. Client here means any sequence of statements that is executed after loading one or more libraries. We denote a client c that executes after loading libraries $l1, .., lk$ as $c_{l1,..,lk}$. We call the sequence $l1, .., lk$ of libraries loaded before executing a client the *library configuration*. The "client" row of Table 1 shows examples of clients.

We target conflicts due to libraries that write to the same globally accessible memory location. In JavaScript, such memory locations are properties of an object. Properties are accessed using either dot notation, e.g., `x.p`, or bracket notation, e.g., `x["p"]`. In either case, the name of a property is represented by an identifier of string type. For properties of nested objects, the property accessors consist of multiple identifiers, e.g., `window.foo.bar`. We call all property accessors, using either single or multiple identifiers, *access paths*. If the first segment of an access path is globally reachable, we call it a *global access path*. For example, `window.foo.bar` and `window.baz` are global access paths. Since the `window`-prefix is optional in JavaScript, we omit it in the remainder of the paper, unless needed.

Based on these definitions, we can now define conflicts between pairs of libraries:

Definition 1 (Conflict). *Let $l_1, l_2 \in \mathcal{L}$ be two libraries that both write to the same global access path p. These libraries are conflicting with each other if there exists a client so that any of the following is true:*
(1) c_{l1} behaves differently than c_{l2}
(2) c_{l1} behaves differently than $c_{l1,l2}$
(3) c_{l1} behaves differently than $c_{l2,l1}$
(4) c_{l2} behaves differently than $c_{l1,l2}$
(5) c_{l2} behaves differently than $c_{l2,l1}$
(6) $c_{l1,l2}$ behaves differently than $c_{l2,l1}$

The first case means that the same client behaves differently depending on which library is loaded. Such a conflict is relevant for the developers of the libraries because these libraries write different data or functions to the same globally accessible memory location. Cases 2 to 5 mean that a client that includes a single library will change its behavior simply because another library is also included.

[6] https://api.jQuery.com/jQuery.noconflict
[7] http://requirejs.org/

Table 1: Examples of conflicts between real-world libraries.

Problem	Inclusion conflict	Type conflict	Value conflict	Behavior conflict

Code Example	`/* curl.js */`
	`window.define`
	`  = function K() { ... };`
	
	`/* dojo.js */`
	`var def = function() { ... };`
	`var req = function() { ... };`
	`if (window.define) {`
	`  ...`
	`} else {`
	`  window.define = def;`
	`  window.require = req;`
	`}`
	`window.require(); // exception`
		`/* ocanvas.js */`
		`(function(a, b, c) {`
		`  a.logs = [];`
		`}(window, document));`
		
		`/* aframe.js */`
		`c = function(e) {`
		`  ...`
		`};`
		`window.logs = c`
			`/* pako */`
			`var pako = {`
			`  Deflate: function() { ... },`
			`  Inflate: function() { ... },`
			`  ...`
			`}`
			
			`/* 3Dmol */`
			`var pako = {`
			`  inflate: function() { ... },`
			`  inflateRaw: function() { ... },`
			`  ...`
			`}`
				`/* jsface */`
				`function O(t, o) {`
				`  ...`
				`}`
				`window.Class = O;`
				
				`/* matreshka */`
				`window.Class`
				`  = function(a, b) { ... }`

Client	`// client that includes first`
	`// curl.js and then dojo.js`
	
	`// exception because require`
	`// is undefined`
		`// try to add to`
		`// the 'logs' array`
		`logs.push("log");`
		
		`// exception because`
		`// logs is a function`
			`Object.keys(pako);`
			
			`// returns 35 with pako`
			`// but 4 with 3Dmol`
				`var v1 = null;`
				`var v2 = "";`
				`v0 = window.Class(v1, v2);`
				
				`// TypeError with matreshka`
				`// but no errors with jsface`

Description	Both libraries write to the global variable `define`. To avoid overwriting an already defined variable, e.g, when the same library is included multiple times, *dojo* checks whether `define` is already defined. Unfortunately, the code incorrectly assumes that `require` is always defined together with `define`, causing an exception when trying to call this function. The problem is triggered by any client that includes first *curl* and then *dojo*.	Both libraries write to a global variable `logs`. The type of `logs` is array in *ocanvas* but function in *aframe*.	Both libraries overwrite the global variable `pako`. The size of the global variable is different in both cases. This overwriting happens because 3Dmol ships a variant of the pako library that misses some features.	Both libraries write to the same global variable `Class`. The implementation of both differ as illustrated by the client.

Such a conflict is relevant for developers of clients who may be surprised that simply including another library causes new behavior. The last case means that a client's behavior changes when swapping the order in which two libraries are included. Again, this case is relevant for client developers because such a change in behavior is surprising.

Based on Definition 1, we say that a library is *conflicting* if there exists another library so that both are conflicting with each other. The problem addressed in this paper is how to find conflicting libraries in a precise way, i.e., without false positives.

2.4 Challenges

Due to the increasing popularity of JavaScript, there exist thousands of libraries. Only few of them come with representative clients that could serve as test cases. Our work aims at detecting conflicts in an automated and scalable way. Automated here means that the approach requires no input except for a set of libraries. Scalable here means that this set may contains thousands of libraries.

To find conflicts in an automated and scalable way, we must address several challenges. First, the sheer number of JavaScript libraries makes it practically impossible for an analysis to compare all combinations or even all pairs of libraries. For example, given 1,000 libraries, there are about 500,000 pairs of libraries. We address this challenge by identifying potential conflicts during an analysis of individual libraries (Section 3.1), which significantly reduces the number of combinations to analyze further. Second, the approach cannot rely on any a-priori available library clients. We address this challenge by synthesizing library clients, guided by the potential conflicts (Section 3.2). Third, to validate whether a potential conflict is indeed a conflict, we need to check whether the behavior of clients differs depending on the library configuration. We address this challenge by comparing the runtime behavior of synthesized clients executed with different library configurations (Section 3.2).

2.5 Scope and Limitations

Some challenges are out of the scope of this work. One of them is detecting all library conflicts. While our approach is precise, it is not sound, i.e., it may miss some conflicts. For most interesting program analysis tasks, providing a sound and precise answer is impossible, and we opt for precision in this work. Another out-of-scope question is how many real-world clients suffer from a detected conflict. Instead of addressing this question, our approach shows the existence of a client by synthesizing the client, so that library developers could anticipate conflicts that any possible client may run into and prevent conflicts before they occur. Finally, we focus on pairwise library conflicts and ignore conflicts that arise only if three or more libraries interact with each other.

3 APPROACH

This section presents ConflictJS, a scalable and automated approach to find conflicts between libraries. Given a set of libraries, the approach consists of two main steps:

(1) *Detection of potential conflicts.* At first, ConflictJS dynamically analyzes individual libraries to identify which globally reachable memory locations they write to. Based on the global writes of each library, the first step then reports a potential conflict for each pair of libraries that write to the same location.
(2) *Validation of conflicts.* This step validates whether two libraries that write to the same globally reachable location can indeed

cause a client to behave differently depending on the library configuration. To this end, ConflictJS synthesizes clients and compares their behavior across different library configurations. If and only if the approach finds a client with diverging behavior, it reports a conflict.

The remainder of this section explains these two steps in more detail.

3.1 Detection of Potential Conflicts

To find potential conflicts between libraries, ConflictJS analyzes the global access paths written to by a library. To this end, we dynamically analyze the loading of each library to keep track of the writes made to the global namespace:

Definition 2 (Global Writes of a Library). *The global writes of a library l is a set $\mathcal{G}_l = \{p_1, .., p_k\}$ of global access paths to which l writes while loading l.*

For example, if the global object is called `window` and a library writes to it using `window.obj = {prop1:1, prop2:2}`, then the set of global writes is `{obj, obj.prop1, obj.prop2}`.

To compute the global writes of a library, ConflictJS generates a trivial client that simply loads the library and dynamically analyzes the execution. The dynamic analysis updates the set $\mathcal{G}$ when specific runtime events occur, as summarized in Table 2. The analysis is guaranteed to observe all global writes that occur while loading the library. In particular, the analysis handles writes to aliases of globally reachable objects, as illustrated by the example involving `window.Array` in Table 2. The access paths of all reachable values, i.e., $paths(v)$ mentioned in Table 2 are computed by recursively traversing the properties of the object v. The information whether a variable is global is provided by Jalangi [46] on top of which we implement the analysis.

After extracting the global writes of each library, ConflictJS compares the global writes of all libraries with each other to check for writes to the same global access path. If two libraries share a global write, we classify them as potentially conflicting:

Definition 3 (Potentially Conflicting Libraries). *Two libraries $l_1, l_2 \in \mathcal{L}$ are potentially conflicting if $\mathcal{G}_{l_1} \cap \mathcal{G}_{l_2} \neq \emptyset$, i.e., if the two libraries share at least one access path in their global writes.*

The first phase of ConflictJS reduces the search space of potential conflicts to be considered by the second phase of the approach. The first phase scales well to a large number of libraries because each library is analyzed in isolation. Comparing the global writes across libraries requires computing pairwise intersections of sets, which easily scales to a large number of sets. As mentioned in Section 2.5, the analysis might miss potential conflicts, e.g., because a library might perform a global write after the library has been loaded. A manual inspection of a subset of libraries suggests this limitation to be negligible in practice, because libraries tend to initialize their APIs at load time.

3.2 Precise Validation of Conflicts

The second step of ConflictJS is to validate potential conflicts identified in the first step. At first, we motivate the need for this second step with an example. Then, we explain the details of the validation.

Table 2: Actions performed by the global-writes analysis.

Runtime event	Action	Example

Variable write $w = v$	If w is a global variable:
	• Add w to $\mathcal{G}$. Let $paths(v)$ be the access paths of all values reachable from v. For each $p_v \in paths(v)$, add p_v to $\mathcal{G}$.
		`(function() {`
		`  var x = {a: 23};`
		`  window.foo = x;`
		`})();`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{foo, foo.a\}$

Property write $x.p = v$	Let $paths(window)$ be the access paths of all globally reachable values. For each $p_w \in paths(window)$:
	• If p_w points to x:
	– Add $concat(p_w, p)$ to $\mathcal{G}$. Let $paths(v)$ be the access paths of all values reachable from v. For each $p_v \in paths(v)$, add $concat(p_w, p_v)$ to $\mathcal{G}$.
		`(function() {`
		`  var x = window.Array;`
		`  x.p = {b: 42};`
		`  var y = {};`
		`  y.q = 5;`
		`})();`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{Array.p, Array.p.b\}$

Declaration of function f	If the global variable f points to the declared function (i.e., the function is globally declared), add f to $\mathcal{G}$.	`(function() {`
		`  function foo() {}`
		`})();`
		`function bar() {}`
		$\mathcal{G} \rightarrow \mathcal{G} \cup \{bar\}$

```
/* JSLite.js */
Array.prototype.remove =
      function(t) {
  var n = this.indexOf(t);
  return n > -1 && this.splice(
        n, 1),
    this
}
```

```
/* ext-core.js */
Array.prototype.remove =
      function(e) {
  var t = this.indexOf(e);
  return -1 != t && this.splice
        (t, 1),
    this
}
```

Figure 2: Example to show the need for validating potential conflicts.

3.2.1 Motivation for Validation. Potentially conflicting libraries write to the same globally accessible memory location. This situation may or may not cause a client to suffer from a conflict as defined in Definition 1. For example, consider Figure 2, which shows code snippets from two potentially conflicting libraries, *JSLite.js* and *ext-core.js*. The global access path to which both libraries write is `Array.prototype.remove`. Both libraries extend the built-in `Array` object by adding a new method `remove`, which can be called with one argument. Even though the two methods are syntactically different, close inspection shows that both pieces of code are functionally equivalent. This example illustrates that reporting all potential conflicts would cause false positives because for some potential conflicts, all clients are guaranteed to observe the same behavior, irrespective of the library configuration.

3.2.2 Synthesizing Clients and Comparing their Behavior. To check whether a potential conflict between two libraries is indeed a conflict, ConflictJS synthesizes library clients and checks whether their runtime behavior differs depending on the library configuration. The basic idea is to consider each of the six scenarios listed in Definition 1 by comparing the behavior of two clients with each other. The two clients contain exactly the same code, except that

Algorithm 1 Validate potential conflicts

Input: Libraries $l1, l2$ that both write to global access path p
Output: Validated conflict between $l1$ and $l2$

1: $c_{empty} \leftarrow$ empty client
2: **if** $conflictingConfigs(c_{empty})$ **then return** "inclusion conflict"
3: $c_{types} \leftarrow$ synthesize client that checks type of p
4: **if** $conflictingConfigs(c_{types})$ **then return** "type conflict"
5: **if** type of p is non-function **then**
6: $\quad c_{values} \leftarrow$ synthesize client that checks value of p
7: $\quad$ **if** $conflictingConfigs(c_{values})$ **then return** "value conflict"
8: **else**
9: $\quad C_{behavior} \leftarrow$ synthesize clients that call function p
10: $\quad$ **for each** $c_{behavior} \in C_{behavior}$ **do**
11: $\quad\quad$ **if** $conflictingConfigs(c_{behavior})$ **then return** "behavior conflict"

12: **function** *conflictingConfigs*(c)
13: $\quad \mathcal{B} \leftarrow \emptyset$ ▷ Set of observed runtime behaviors
14: $\quad$ **for each** $config \in \{l1, l2, l1l2, l2l1\}$ **do**
15: $\quad\quad b_{config} \leftarrow$ execute c_{config}
16: $\quad\quad \mathcal{B} \leftarrow \mathcal{B} \cup \{b_{config}\}$
17: $\quad$ **if** $|\mathcal{B}| > 1$ **then return** true
18: $\quad$ **else return** false

they run with different library configurations. If ConflictJS observes a behavioral difference between the two clients, the potential conflict between the two libraries is indeed a conflict.

For illustration, consider the behavior conflict illustrated in Table 1. Our approach tries to validate this conflict by synthesizing clients, such as the client shown in the table. The approach compares the behavior of this client with different library configurations. For the example, ConflictJS finds that on calling `Class`, there is one library that throws an exception while the other does not. That is, the approach has validated the conflict and reports it, along with the synthesized client that illustrates the conflict.

Algorithm 1 summarizes our approach for validating potential conflicts by synthesizing and dynamically executing library clients. The main idea is to compare the execution of a client c with different library configurations, i.e., c_{l1}, c_{l2}, $c_{l1,l2}$, and $c_{l2,l1}$, as summarized in function *conflictingConfigs*. If there are multiple different behaviors, then the algorithm has validated a conflict. The following describes how ConflictJS creates clients to detect the four kinds of conflicts presented in Section 2.2.

3.2.3 Checking for Inclusion Conflicts. At first, ConflictJS checks for inclusion conflicts (lines 1 to 2). An inclusion conflict is triggered by simply including libraries, i.e., the client is an empty client that does not contain any statements. To compare library configurations, the behavior b_{config} indicates whether including libraries causes the client to throw an exception. If one library configuration causes an exception, whereas another configuration does not, then ConflictJS reports an inclusion conflict.

For the inclusion conflict example of Table 1, ConflictJS reports a conflict because trying to execute the empty client after loading *curl.js* and *dojo.js* causes an exception, whereas executing the empty client after loading only one of these libraries does not throw any exception.

3.2.4 Checking for Type Conflicts. For any pair of libraries $l1, l2$ and shared global access path p for which the approach has not validated an inclusion conflict, the next step is to check for type conflicts. To this end, ConflictJS synthesizes a client that reads the value at the access path p and then checks its type (lines 3 to 4). The approach again executes this client with all possible library configurations and summarizes the behavior of each configuration as the type of the access path p. If one library configuration causes the client to see type t_1, whereas another library configuration causes the client to see type $t_2 \neq t_1$, then ConflictJS reports a type conflict.

An example of a library pair with a type conflict is given in the second column of Table 1. The approach reports this conflict because `push` is an array when loading one library but a function when loading the other library. The "client" cell of the table shows a client that suffers from this type conflict because the conflict causes the client to crash when it tries to call a function that turns out to be an array.

3.2.5 Checking for Value Conflicts. While checking for type conflicts, the analysis gathers information about the types of values stored at a global access path. For potential conflicts that are neither validated to be an inclusion conflict nor to be a type conflict, both libraries write values of the same type to the access path. Based on this type, ConflictJS checks for the remaining two kinds of conflicts. If the type is function, the approach compares the behavior of clients that call this function, as described below. If the type is a non-function, then the approach synthesizes a client that reads the value at the access path p (lines 6 to 7). To compare the behavior of this client across library configurations, ConflictJS compares the value read at p. The analysis directly compares primitive values and deeply compares objects. If different library configurations cause the client to read different values, then ConflictJS reports a value conflict.

The "value conflict" column of Table 1 gives an example of a type conflict on the `pako` access path. ConflictJS synthesizes a client that extracts the number of properties of the value stored at `pako` and the then recursively extracts the values of these properties. The approach reports a conflict because the number of `pako`'s properties depends on whether the *pako* library or the *3Dmol* library is loaded.

3.2.6 Checking for Behavior Conflicts. The most challenging kind of conflict are behavior conflicts. These conflicts occur when different libraries write functions to the same global access path but the behaviors of these functions differ. In general, deciding whether the behavior of two functions differs is undecidable. ConflictJS approaches this problem by trying to synthesize clients that expose a difference in behavior. If the analysis succeeds in generating such a client within a fixed time budget, it reports a behavior conflict.

To synthesize clients we use a simple test generator inspired by Randoop's feedback-directed, random test generation [36]. Other test generation techniques, such as symbolic or concolic execution [5, 15, 28] or search-based test generation [12], could also be used for this step. Given a function-typed access path p defined by two libraries, the test generator starts by estimating the number

n of arguments that the function expects. To this end, we use the `length` property of the function object at *p*, which in JavaScript yields the number of declared function parameters. This number is an estimate because a function body may also access additional arguments using the built-in `arguments` value. Next, to generate a call to the function, the test generator randomly decides on a random number ranging between 0 and *n* of arguments to pass. For each argument, the test generator decides on the type of argument to create by randomly choosing between the following types: *boolean*, *string*, *number*, *array*, *object*, *undefined* and *null*. To create a boolean, string, or number, the generator picks from a pre-defined pool of values. For arrays, the generator randomly picks a length ranging between 0 and 10 and fills it with random strings and numbers. Finally, to create an object, the generator creates up to 10 properties and assigns randomly generated values to them.

Once the arguments are generated, the function is called using the generated arguments. If and only if the call succeeds, without raising an exception, for at least one library configuration, the generator synthesizes a client that contains this call.

To compare the behavior of synthesized clients across library configurations, ConflictJS summarizes the behavior of the client execution based on the return value of the function and based on whether the function raises an exception. The approach reports a behavior conflict in two cases: (i) if one library configuration causes the client to crash whereas another library configuration does not cause a crash, or (ii) if both configurations do not crash but the return value of the function at *p* differs.

For example, consider the last column of Table 1. ConflictJS synthesizes clients that call the function stored at the conflicting access path `Class`. The client shown in the table throws an exception for one of the two libraries but not for the other, which is why ConflictJS reports a behavior conflict.

4 IMPLEMENTATION

We implement ConflictJS as a client-server-based tool that analyzes JavaScript libraries. The client component synthesizes, executes, and analyzes clients in a browser, and sends a summary of the runtime behavior to the server. The server detects potential conflicts and validates them based on execution behavior gathered in the first and second phase, respectively. Our dynamic analyses to find global writes is build on top of Jalangi [46]. When synthesizing clients to detect behavior conflicts, we set the testing budget to 50 tests per access path. In this paper, we implement the approach only for client side JavaScript libraries and it would be straightforward to adapt for server-side npm libraries but the problem is less severe for Node.js because there is a commonly accepted module system.

5 RESULTS AND DISCUSSION

We apply ConflictJS to 951 popular JavaScript libraries to evaluate the effectiveness of the approach in detecting library conflicts. We focus on the following research questions:

- How effective is ConflictJS in finding library conflicts and what kinds of conflicts occur in practice? (Section 5.2)
- What are the root causes of conflicts between libraries? (Section 5.3.1)

Table 3: JavaScript libraries used for the evaluation.

	Min	Median	Max	Total
Libraries	-	-	-	951
Lines of code	9	574	275,0852	2,750,852
Size (bytes)	148	14,645	2,517,510	68,412,720

- Do library developers make an effort to avoid conflicting scenarios by following the "single API object" pattern? (Section 5.3.2)
- What are the popular access paths that developers tend to choose? (Section 5.3.3)
- Is there a correlation between conflicts and the popularity of a library? (Section 5.3.4)
- How are the global writes and conflicts distributed across libraries and access paths, respectively? (Sections 5.3.5 and 5.3.6)

5.1 Experimental Setup

Our evaluation uses 951 real-world JavaScript libraries with a total of 2,750,852 lines of JavaScript code (Table 3). The libraries include the popular *jQuery*, *Underscore*, and *Dojo* projects, as well as various other highly popular libraries. We obtain these libraries by downloading them from the CDNJS content delivery network.[8] At the time of starting our experiments, the content delivery network offered a total of 2,095 libraries. We remove libraries that cannot be used in isolation in a standard desktop browser, e.g., because they rely on another library or because they target mobile devices. We heuristically check for such libraries by loading each library in isolation and filtering away all libraries that throw an exception. After filtering, 951 libraries remain, which is our benchmark set for the evaluation. To run our experiments, we use an Intel Core i7–4790 CPU machine clocked at 3.60GHz with 32 GB of memory, running Chrome 55, Node.js 6.9.1 on Ubuntu 16.04.

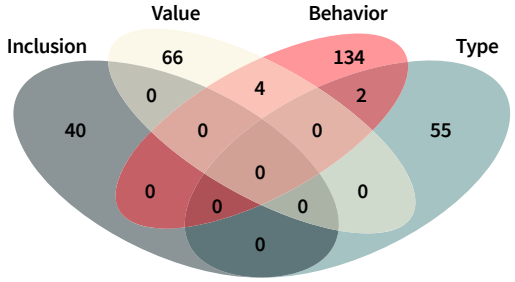
5.2 Effectiveness in Finding Library Conflicts

5.2.1 Potential Conflicts. When analyzing the global writes of individual libraries, ConflictJS records writes to a total of 130,714 different access paths across the 951 libraries. Intersecting the global writes of libraries reveals that 4,121 of the access paths cause a potential conflict, i.e., at least two libraries write to each of these access paths. These conflicting writes are performed by 268 of the 951 libraries, i.e., roughly one out of four libraries is involved in a potential conflict.

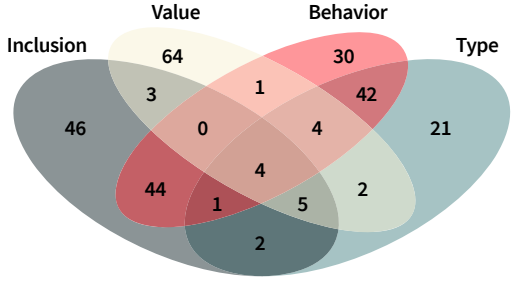
5.2.2 Validated Conflicts. Out of the 268 potentially conflicting libraries, ConflictJS validates 166 as certainly conflicting by synthesizing a client whose behavior depends on the library configuration. The validated conflicts are due to 1,840 distinct access paths. In other words, ConflictJS successfully validates 62% of the potentially conflicting libraries (i.e., of 268 libraries) as certainly conflicting and finds a validated conflict in 17% of all libraries (i.e., of 951 libraries).

5.2.3 Kinds of Validated Conflicts. Figure 3 summarizes how prevalent the four kinds of conflicts are among all validated conflicts. The two sides of the figure provide different views on the same data. Figure 3a focuses on pairs of conflicting libraries and shows how many of these pairs are caused by the four kinds of conflicts. If a pair of libraries is involved in multiple kinds of conflicts, then this pair

[8] https://cdnjs.com/



(a) Library pairs classified by the type of conflict.



(b) Libraries classified by the type of conflict.

Figure 3: Prevalence of the four kinds of validated conflicts. Note that the surface is not proportional to the numbers.

is shown at the set intersection. For example, there are four pairs of libraries that have a value conflict for a global access path and a behavior conflict for another global access path. Figure 3b shows the distribution among the four kinds of conflicts for individual libraries. Since a single library may be involved in conflicts with different libraries, these sets overlap. For example, there are seven libraries that are involved in at least one inclusion conflict, value conflict, and behavior conflict.

There are two main take-aways of these results. First, all four kinds of conflicts are prevalent in practice, which confirms our decisions to consider all four kinds in ConflictJS. Second, the majority of conflicts are non-inclusion conflicts, i.e., they do not cause an exception just after loading the conflicting libraries. Finding such conflicts and reasoning about them is challenging for both library developers and users alike.

5.3 Empirical Study of Library Conflicts

The large number of libraries considered and conflicts detected in our evaluation, enables us to learn more about how and why conflicts occur in JavaScript libraries. We discuss these findings in the following and discuss what impact they have on library developers, library users, and language designers.

5.3.1 Root Causes of Conflicts. To understand the root causes of conflicts between libraries we manually inspect a random sample of 25 conflicting libraries. During the manual inspection, we identified seven recurring patterns. Table 4 describes each pattern and and illustrates it with an example.

Five of the seven patterns, which account for 18 out of the 25 inspected conflicts, are unintended by the developers and likely to cause surprising behavior for library users. These patterns are shown in the upper part of Table 4. The patterns cover conflicts caused by independently developed variants of the same functionality, copied third-party code, poor API usage, repeated use of convenient global identifier name, and incorrect attempts to patch built-in JavaScript APIs. To double-check our intuition about whether conflicts are intended by the library developers, we reported seven conflicts to the developers of conflicting libraries. At the time of writing, four of our reports have been acknowledged and confirmed as worth fixing by the respective developers. Of the four acknowledged libraries, two have been fixed by the developers. Apart from this, based on our bug report, the developer of a library has reported a bug to another library with which it was conflicting. Subsequently, this bug report also got fixed.

For all of these five patterns, the root cause boils down to suboptimal decisions by library developers, such as programming errors or copy-and-paste of existing code. However, at least for some of them, the design of the JavaScript language and APIs may also be partially to blame. For example, instances of the "Poor API usage" pattern are caused by the fact that the JavaScript web APIs provide two orthogonal ways to attach event handlers: Setting the handler, e.g., `onmessage = ..`, which overwrites any already attached handler, and adding a handler via `addEventListener("message", ..)`, which preserves already attached handlers. The conflicts detected by ConflictJS are the result of libraries overwriting each other's event handlers by directly setting the handler. Another example is the "Incorrect monkey patching" pattern. The term "monkey patching" refers to extending built-in APIs of the JavaScript language, which is possible but non-trivial to implement without removing existing functionality.

The remaining two patterns, shown in the lower part of Table 4, both occur in a situation where library users are unlikely to be surprised by the conflict. One reason is that libraries depend on each other and document these dependencies clearly, so that library users know in which order to load them. Ideally, our experimental setup would filter such libraries, as we assume each library is supposed to be used independently. Another reason is that libraries provide the same or very similar overall functionality, so that library users would never include both together.

Overall, we draw two conclusion from our manual inspection. First, most conflicts reported by ConflictJS are programming errors that should be fixed by library developers to prevent clients from surprising behavior. Second, the root causes of conflicts are diverse but can be classified into a set of recurring patterns. Knowing these patterns may become the basis of guidelines for library developers what mistakes to avoid. Furthermore, the patterns can guide the design of future program repair techniques that fix conflicting code.

5.3.2 The "Single API Object" Pattern. The "single API object" pattern (Section 2.1) allows developers to avoid conflicts by storing all globally accessible data into a single object named like the library. If all libraries follow this pattern, no conflicts occur. To understand whether libraries follow this pattern, we check for each library whether for all writes to a global access path, the path begins with a segment that matches the name of the library, as listed in the CDNJS content delivery network. When matching an access path and a library name, we omit the *.js* suffix that some libraries use.

We find that 273 out of the 951 libraries follow the "single API object" pattern. While promising, this means that 71% of all libraries do

Table 4: Recurring patterns among the root causes of conflicts.

Pattern	Description	Nb.	Example(s)
Independent implementations	Two libraries implement similar functionality and use the same global access path to store the function, but the behavior slightly differs.	5/25	*polymer* and *trix* both define `wrap` and `unwrap` functions. Other examples: Figure 1 and issue #434 of *es6-shim*.
Copied third-party code	Two libraries both copy code from a third party, e.g., another library. At least one version of the code is outdated.	5/25	*qooxdoo* includes an outdated copy of *sinon*. See issue #9277 of *qooxdoo*. Another example: Issue #1068 of *d3fc*.
Poor API usage	A library adds an event handler in a way that removes all other handlers for this kind of event, instead of adding to the existing event handlers.	4/25	*rxjs* and *gifshot* both write to `onmessage` to handle postMessage communication. Instead, they should use `addEventListener`, which allows multiple event handlers.
Convenient identifier	Two libraries use a convenient, global identifier for different purposes.	3/25	*mermaid*, a library for generating diagrams, writes to `_`, which is also used by *score-js* and others. See issue #512 of *mermaid*.
Incorrect monkey patching	A library tries to extend a built-in API but accidentally removes existing functionality.	1/25	*PreloadJS* and *zingchart* both overwrite the built-in `JSON` in a way that destroys existing functionality. See issue #226 of *PreloadJS*.
Documented dependency	One library depends on another and documents this dependency.	4/25	*alloy-ui* is a framework built on top of *yui*. Clients should not be surprised by "conflicts" between them.
Fork	One library is derived from another library and modifies or extends the functionality of the original library.	3/25	*wysihtml* is an extended version of *wysihtml5*. Clients should never use both together.

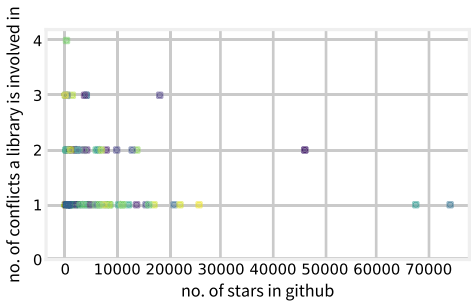
Table 5: Popularity of global access paths (measured in the number of libraries that write to an access path).

Libs.	Global access paths
13	$
12	localStorage.debug
10	requestAnimationFrame
9	_, jQuery, onload, require
8	clearImmediate, Promise, __core-js_shared__, __core-js_shared__.wks, setImmediate, __core-js_shared__.wks.iterator, __core-js_shared__.wks.toStringTag

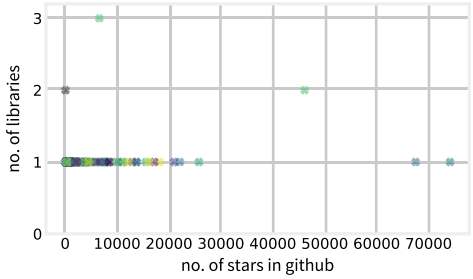
not follow the pattern, but instead use the shared global namespace in a possibly conflicting way. We conclude that relying on developer discipline in an open environment, such as the JavaScript library ecosystem, is insufficient to enforce a conflict-avoiding policy.

5.3.3 Popular Global Access Paths. The large number of potential conflicts detected by ConflictJS raises the question what global access paths are particularly popular among library developers. Table 5 lists the most popular global access paths along with the number of libraries that write to it. Perhaps unsurprisingly, the most popular access path is the dollar sign, $, which is a legal identifier name in JavaScript and used by several libraries, including *jQuery* to export their API. Another popular choice is the underscore sign, _, which is shared, e.g., by the *Underscore* and *Lodash* libraries. Choosing a short identifier name to export an API is tempting for library developers and potentially convenient for library users. However, the downside is that multiple libraries may (either knowingly or not) pick the same short identifier name, which likely causes surprises if these libraries are used together.

5.3.4 Conflicts Versus Library Popularity. To better understand to what extent library conflicts depend on a library's popularity, Figure 4a shows for each library how many stars it has and in how many conflicts it is involved. Each data point corresponds to one library. For example, one library that has 45,901 starts is involved in two conflicts. Overall, the figure shows that most conflicts are due to libraries with less than 10,000 stars. The main reason is that only few libraries have more than 10,000 stars, as illustrated in Figure 4b. This figure shows how the libraries validated to be conflicting from



(a) Number of conflicts each library is involved in.



(b) Popularity measure of libraries validated to be conflicting

Figure 4: Influence of popularity on number of conflicts and number of libraries. Each data point represents one library. (twelve out of 166 libraries do not have a Github repository and hence are not included here)

our benchmark are distributed across the popularity measure. Both figures look similar, which explains the distribution of conflicts across popularity. At the same time, it is interesting to note that even some highly popular libraries are involved in conflicts, as indicated by the data points on the right end of Figure 4a.

5.3.5 Distribution of Global Writes Across Libraries. To better understand the large number of 130,714 global writes performed by the 951 libraries, we analyze how these writes are distributed across the libraries. The results show a highly skewed distribution, with a few libraries writing to many global access paths but with a median

of only one global write. The libraries that write to most global access paths are large and popular libraries, such as Amazon's AWS JDK (36,049 access paths) and Microsoft's implementation of TypeScript (5,678 global writes). Their high number of global writes does not imply bad coding practice. For example, the many access paths written by the AWS SDK library almost all start with `AWS.`, i.e., they follow the "single API object" pattern. We conclude that judging libraries based on their total number of global writes, which might have been a simple alternative to ConflictJS, is not an effective way to find conflict-triggering libraries.

5.3.6 Distribution of Conflicts Across Access Paths. A reader may wonder how many libraries write to the same global access path. Investigating this question yields a long-tail distribution: Most global access paths (3,836) are contended for by only two libraries, but a large number of highly popular global access paths is written to by up to 13 libraries. We conclude that preventing library developers from using a few highly contended access paths, such as `$` and `_`, is insufficient to solve the problem of library conflicts, because there are many other access paths that cause conflicts.

6 RELATED WORK

Lint-like Checkers. Lint-like checkers search for bad coding practices through lightweight static analysis[9], dynamic analysis [17], and combinations of both [10]. Some of them, e.g., ESLint and JSNose [10], warn about excessive use of global variables within a single file, but they do not analyze conflicts across files or libraries.

Analysis of Libraries. Existing analyses of JavaScript libraries check that a library implementation matches its interface specification [11], statically analyze library clients to understand types and other properties of a library [30], and search for code injection vulnerabilities [49]. Our work synthesizes library clients instead of analyzing existing clients. Beyond JavaScript, Pollux [27] determines the effects of upgrading a library. Our work differs from all the above by analyzing multiple libraries and their potential interactions, instead of a single library.

Dynamic Analysis for JavaScript. A survey [3] summarizes dynamic analyses for JavaScript. Existing analyses include determinacy analysis [43], dynamic data race detectors [23, 33, 37, 41], dynamic model checkers [24], profilers to detect performance problems [16, 26, 44], taint- and information-flow analyses [4, 6, 21], and analyses to understand code changes [2] and the root cause of a crash [31]. All these techniques are orthogonal to ConflictJS, which is the first to focuses on library conflicts.

Test Generation. The test synthesis part of ConflictJS relates to generating test cases, such as feedback-directed, random test generation [36], symbolic and concolic execution [5, 15, 47], and search-based testing [12]. JSeft [32] exploits fixtures extracted from executions to create tests. These techniques could help the second phase of ConflictJS to further increase the percentage of validated behavior conflicts.

Type Checking and Type Inference. Type conflicts relate to type errors and type inconsistencies [38]. Several approaches infer and check types through static [18, 22, 25, 50], dynamic [38], or hybrid [19] analysis. None of these has been applied to multiple libraries. Another difference is that most type checkers focus on soundness and therefore suffer from false positives, whereas ConflictJS validates potential conflicts.

Studies of JavaScript Code. Studies show that JavaScript libraries are widely used and often combined with each other. Nikiforakis et al. [34] report that 88% of the websites include at least one remote library, and that libraries are loaded from over 300.000 unique URLs. Another study [29] shows that a web site includes a median of 9 and a maximum of 202 externally hosted scripts. These numbers illustrate the risk of accidental conflicts between libraries. Beyond JavaScript, Eshkevari et al. report conflict-like problems in PHP applications [9]. Other studies investigate recurring performance bottlenecks [45], dynamic code loading [40, 42], insecure coding practices [48, 51], type coercions [39], type-related errors [14], recurring bug patterns [20], the use of trivial software packages [1], the root causes of failures [35], and the use of callbacks [13].

7 CONCLUSION

JavaScript code, including independently developed libraries, shares the same global namespace. Because the most widely used versions of the language lack features designed for encapsulating exported APIs, library developers risk to accidentally share the same globally accessible memory locations and write different data and functions to them. This paper defines and classifies such library conflicts, presents an automatic and scalable approach to detect them, and studies conflicts in a large set of libraries. We deal with the huge search space of possible conflicts through a two-phase approach that dynamically analyzes libraries in isolation to detect potential conflicts and then synthesizes library clients to validate conflicts. Among 951 real-world libraries, the approach finds 166 (17%) certainly conflicting libraries. Furthermore, we empirically study how and why conflicts occur, showing that a diverse set of programming errors in libraries are the primary root cause.

Our work not only provides a practical tool for library developers to detect conflicts and for library users to avoid conflicting libraries, but also highlights the importance of language features for encapsulating independently developed code. We believe that our work provides ample opportunities for future work. One direction is to complement our precise but unsound analysis with a sound (and likely imprecise) checker for library conflicts. To help developers avoid conflicts, another line of future work are repair tools that either address the coding errors that cause conflicts. Finally, future work could develop automatic code transformations to help libraries use encapsulation mechanisms provided in recent and future versions of JavaScript.

Acknowledgments
This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case

[9] Popular tools include http://eslint.org/, http://jshint.com/, and http://www.jslint.com/.

Study on npm. In *FSE*.
[2] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *ECOOP*. 321–345.
[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] Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis.. In *PLAS*.
[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 209–224.
[6] Ravai Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged Information Flow for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 50–62.
[7] ECMA. 2011. Standard ECMA-262, ECMAScript Language Specification, 5.1 Edition. (June 2011).
[8] ECMA. 2015. Standard ECMA-262, ECMAScript Language Specification, 6th Edition. (June 2015).
[9] Laleh Eshkevari, Giuliano Antoniol, James R. Cordy, and Massimiliano Di Penta. 2014. Identifying and Locating Interference Issues in PHP Applications: The Case of WordPress. In *Proceedings of the 22Nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 157–167. DOI:http://dx.doi.org/10.1145/2597008.2597153
[10] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 116–125.
[11] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 1–16.
[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 416–419.
[13] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. 247–256.
[14] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 758–769.
[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223.
[16] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 357–368.
[17] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.
[18] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*. 256–275.
[19] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 239–250.
[20] Quinn Hanam, Fernando Santos De Mattos Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 144–156.
[21] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*. 1663–1671.
[22] Phillip Heidegger and Peter Thiemann. 2010. Recency Types for Analyzing Scripting Languages.. In *European Conference on Object-Oriented Programming (ECOOP)*. 200–224.
[23] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications.. In *ICST*. 61–70.
[24] Casper Svenning Jensen, Anders Moller, Veselin Raychev, and Martin Vechev. 2015. Stateless Model Checking of Event-Driven Applications. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.
[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Symposium on Static Analysis (SAS)*. Springer, 238–255.
[26] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 345–356.
[27] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. 2016. POLLUX: safely upgrading dependent application libraries. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 290–300.
[28] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
[29] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *NDSS*.
[30] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/SIGSOFT FSE*. 499–509.
[31] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Feedback-directed instrumentation for deployed JavaScript applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 899–910.
[32] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSEFT: Automated Javascript Unit Test Generation. In *ICST*.
[33] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.
[34] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 736–747. DOI: http://dx.doi.org/10.1145/2382196.2382274
[35] 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.
[36] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. IEEE, 75–84.
[37] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Conference on Programming Language Design and Implementation (PLDI)*.
[38] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*.
[39] 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)*.
[40] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*.
[41] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[42] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming (ECOOP)*. 52–78.
[43] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *PLDI*. 165–174.
[44] Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An Actionable Performance Profiler for Optimizing the Order of Evaluations. In *International Symposium on Software Testing and Analysis (ISSTA)*. 170–180.
[45] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.
[46] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 488–498.
[47] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 263–272.
[48] Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites.. In *NDSS*.
[49] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*.
[50] Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *European Symposium on Programming (ESOP)*. 408–422.
[51] Chuan Yue and Haining Wang. 2009. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*. 961–970.