Discovering Bug Patterns in JavaScript

Quinn Hanam
University of British Columbia
Vancouver, BC, Canada
qhanam@ece.ubc.ca

Fernando S. de M. Brito[*]
Univ. Federal da Paraíba
João Pessoa, PB, Brazil
fernando@lavid.ufpb.br

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

JavaScript has become the most popular language used by developers for client and server side programming. The language, however, still lacks proper support in the form of warnings about potential bugs in the code. Most bug finding tools in use today cover bug patterns that are discovered by reading best practices or through developer intuition and anecdotal observation. As such, it is still unclear which bugs happen frequently in practice and which are important for developers to be fixed. We propose a novel semi-automatic technique, called BugAID, for discovering the most prevalent and detectable bug patterns. BugAID is based on unsupervised machine learning using language-construct-based changes distilled from AST differencing of bug fixes in the code. We present a large-scale study of common bug patterns by mining 105K commits from 134 server-side JavaScript projects. We discover 219 bug fixing change types and discuss 13 pervasive bug patterns that occur across multiple projects and can likely be prevented with better tool support. Our findings are useful for improving tools and techniques to prevent common bugs in JavaScript, guiding tool integration for IDEs, and making developers aware of common mistakes involved with programming in JavaScript.

CCS Concepts

•Software and its engineering → Software testing and debugging; Empirical software validation;

Keywords

Bug patterns, JavaScript, Node.js, data mining, static analysis

[*]This author was a visiting student at the University of British Columbia, when this work was done.

1. INTRODUCTION

A recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [54]. JavaScript is the language used inside the browser but it is also becoming popular as a desktop and server-side language thanks to the Node.js[1] platform.

Despite this, JavaScript currently lacks IDE support [12] in the form of analysis alerts that warn the developer about potential bugs in their code. Such alerts are common to compiled, strongly typed languages such as Java in the form of compilers and bug finding tools like FindBugs [13]. The lack of tool support for JavaScript is in part due to several challenges that are unique to JavaScript including, but not limited to weak typing [22, 49], dynamic field access and creation [51], higher order functions, dynamic code evaluation and execution [50], and event-driven and asynchronous flow of control [5, 16].

Recent research advances have made the use of static analysis on JavaScript more practical [12, 21, 25, 32, 40, 47, 53], while other techniques mitigate the analysis challenges by using a dynamic or hybrid approach [4, 17, 34]. As toolsmiths begin to develop the first bug finding tools for JavaScript, it is unclear which bugs require the most attention. Unlike the most popular statically typed languages (e.g., [15, 48, 56]), there is little research studying bugs in JavaScript. While DOM API interactions have been shown to dominate bugs in client-side JavaScript [46], to the best of our knowledge, there is no work studying common bug patterns in JavaScript in general. The goals of this paper are twofold, namely, (1) developing a systematic semi-automated approach for discovering pervasive bug patterns; (2) discovering bug patterns in server-side JavaScript that occur frequently across projects. We accomplish these goals in two parts.

First, we propose a novel semi-automatic approach to discover common bug patterns. Our approach, called BugAID, mines version control repositories and discovers bugs that are frequently repaired by developers. Our insight that makes this approach effective is that instances of bug patterns cau-sed by language misuse can be grouped together by changes to *language constructs* in the bug fixing commit. Using this intuition, BugAID creates feature vectors of language construct changes and uses unsupervised machine learning to group them into ranked clusters of bug patterns. By inspecting these clusters, we create natural language descriptions of pervasive bug patterns.

Second, we produce the first ranked list of frequently oc-

---

[1]https://nodejs.org

curring bugs for a programming language by performing a large-scale study of common bug patterns in server-side JavaScript applications. Using BUGAID, we analyze 105,133 commits from 134 JavaScript projects (mostly server-side Node.js applications and packages) and discover 219 common bug fixing change types. We inspect the commits in each change type and discuss the 13 bug patterns that we believe have the highest maintenance costs and are good candidates for automated tool support.

This paper makes the following main contributions:

- A novel technique for automatically learning bug fixing change types based on language construct changes.
- The first comprehensive study of pervasive bug patterns in server-side JavaScript code.
- Our toolset BUGAID and empirical dataset, which are publicly available [3].

2. CROSS-PROJECT BUG PATTERNS

Software bugs can be grouped by their fault (the root cause of the bug), their failure (how they manifest in output), or their fix. A group of defects along one of these dimensions is referred to as a *defect class* [35]. Defect classes are important because they allow us to group defects while designing techniques that mitigate them. For example, static bug finding tools tend to look for defect classes grouped by fault, dynamic bug finding tools target defect classes grouped by failure symptom, and automated repair tools fix defect classes grouped by repair.

Our interest in this work lies in defect classes that are detectable across multiple projects. We call these defect classes *Cross-Project Bug Patterns*.

Definition 1 (Cross-Project Bug Pattern) *A pattern in source code that produces incorrect behaviour, has a consistent fault and repair, and occurs across multiple projects.* □

We include the fault in our defect classes because cross-project tools typically discover defect instances based on their fault. We also include the repair in our defect classes because in order to be effective, cross-project bug finding tools should also produce alerts that are *actionable* [19, 24, 29, 52].

```
1  function(obj, iterator, callback) {
2    callback = callback || noop;
3  + obj = obj || [];
4    var nextKey = _keyIterator(obj);
5    if(limit <= 0) {
6      return callback(null);
7  }
```

Figure 1: A potential `TypeError` in `_keyIterator` at line 4 is repaired by initializing `obj` if it is falsey at line 3.

Consider the bug fix in Fig. 1 from the *async* Node.js project. Prior to the repair, the parameter `obj` could have the value `undefined` at line 4. When fields or methods are accessed on `undefined` objects in JavaScript, a TypeError is thrown. If `obj` is undefined, such an error is thrown inside the method `_keyIterator`. At line 3, the developer repairs the bug by checking if `obj` could be `undefined` and if it is, initializing it to an empty array. We consider this an example of a cross-project bug pattern because it has a specific fault – dereferencing a non-value, a consistent repair – checking if the value of `obj` is `undefined`, and can be seen in many JavaScript projects.

3. APPROACH

The seminal static analysis papers by Engler [8] and Hovermeyer [20] do not provide a systematic approach to discovering and selecting pervasive bug patterns to detect. More than a decade after their publication, many of the popular cross-project bug finding tools in use choose a set of defect classes to detect based on best practices or through developer intuition and observation; this is the case with FindBugs [20] and DLint [17].

Because the goal of software companies is to maximize profitability, they should deploy tools that discover defect classes with the highest costs. Monperrus describes the two metrics of defect classes that contribute to costs as the *frequency* with which a defect class appears and the *severity* of a defect class. Frequent defect classes result in high maintenance costs while severe defect classes result in high production loss [35]. The question we are interested in is *how do we systematically discover frequently occurring bug patterns?*

3.1 Reducing the Search Space

One way is to search for bug patterns that are frequently repaired by developers. This can be done by inspecting source code changes in project histories. There is, however, a problem with this method. An inspection of all the bug repairs in one project's history by a human might take several days for a project with a few thousand commits. This means a manual inspection of a sufficient set of projects representative for a language is not feasible. We must therefore reduce the search space in which we look for frequently occurring bug patterns. Because bug patterns have consistent repairs, we can reduce the search space by grouping bug patterns based on their repairs. Repairs can be observed automatically by extracting the source code changes of a bug fixing commit [33]. We focus on commits rather than bug reports because developers often omit links from commits to bug reports or do not create bug reports in the first place [7].

First, we reduce the search space by only considering commits where between one and six statements are modified (i.e., inserted, removed or updated). If zero statements are modified, the commit has not modified any code, while repetitive cross-project repairs have been shown to contain six or fewer modified fragments [39].

The search space should now exclude many repairs that are not repetitive across projects. However, there may still be many bug patterns that do not occur frequently or related bug patterns that are fragmented throughout the search space, making manual inspection challenging. Next, we consider automatically grouping related bug patterns.

3.2 Grouping Cross-Project Bug Patterns

Given a large number of commits with 1-6 modified statements, our goal is to group bug fixing commits with the same bug pattern. Because we do not have a priori knowledge about what bug patterns exist, to achieve this goal we perform cluster analysis using machine learning. The challenge we face is selecting the best feature vector and clustering algorithm such that (1) the number of commits a human must inspect is minimized and (2) the number of

bug patterns recalled by an inspection of the clusters is maximized. Ideally, each cluster would contain all instances of one bug pattern (perfect recall) and only instances of one bug pattern (perfect precision).

So what should our feature vector look like? Our feature vector must capture unknown semantic changes, while ignoring noise, such as variable names and minor differences in control flow. A naive approach is to use a source code differencing approach such as line level differencing or AST differencing. As we discovered early in our work, such an approach does not consider the semantics of the changes and is therefore highly susceptible to noise. A better approach is *change classification*, where semantics are added to the source code changes identified by source code differencing.

Change classification approaches already exist for Java. Fluri and Gall [14] identify 41 basic change types that can be performed on various Java entities, and use those basic changes to discover more complex changes [15]. Kim et. al. [27] enhance this approach by creating a database of basic change facts and using Datalog to specify more complex change types. However, we wish to detect **unknown** change types. From this context, both these approaches suffer, because the lowest level change types which they use, which we call *basic change types* (**BCTs**), must be manually specified (imperatively or declaratively). Such an approach is limited by these pre-defined BCTs and does not scale to capture new patterns with the addition of data.

3.3 Learning Change Types

Our intuition is that we can automatically learn BCTs by combining information from a model of the language with information from source code differencing. We introduce the following abstraction for capturing BCTs. In our abstraction, all BCTs are made up of the following components:

Language construct type: the type of language artifact or concept being modified. For JavaScript, a model of the language may include reserved words, API artifacts (e.g., method and field names), statement types, operators, behaviour (e.g., auto casting behaviour) and even code conventions (e.g., error first protocol).

Language construct context: the context in which the language construct appears. For example, the reserved word `this` could be used in many different contexts such as inside a branch condition or as an argument.

Modification type: how the language construct was modified by the commit. This information is computed by the source code differencing tool.

Name: the name we assign to the language construct.

Assuming the order in which classified changes occur in a commit does not matter much, we can represent more complex change types, i.e., change types made up of one or more BCTs, as a bag of words, where each BCT discovered is a feature, and the number of times a BCT appears in a commit is the value of that feature. For the rest of the paper, when we refer to a *change type*, we mean a set of one or more BCTs.

3.4 Language Construct Selection

We must decide which aspects of the programming language we wish to model. To do this, we distinguish between three types of bug patterns: patterns that are inherent to the programming language, patterns that are inherent to

Table 1: Feature Properties

Identifier	Short Forms
Type	Behaviour(B), Class(CL), Constant(CO), Convention(CV), Error(EX), Event(EV), Field(F), Method(M), Parameter(P), Reserved(RE), Variable(V)
Context	Argument(A), Assignment LHS(AL), Assignment RHS(AR), Class Declaration(CD), Condition(C), Expression(EXP), Error Catch(EC), Event Register(ERG), Event Remove(ERM), Method Call(MC), Method Declaration(MD), Parameter Declaration(PD), Statement(S), Variable Declaration(VD)
Change Type	Inserted(I), Removed(R), Updated(U)
Name	`undefined, equal, return, callback, error,`...

external libraries, and patterns that are inherent to a particular project (e.g., [28, 55]). Our goal in this paper is to discover bug patterns that are inherent to JavaScript. We therefore use JavaScript reserved words, operators and standard methods/fields/events from the ECMAScript5 API [57, 36].

In addition, we include the following JavaScript interpreter behaviours and coding conventions:

falsey: all variables in JavaScript can be cast as booleans in a branch condition. Types `undefined, null` and `NaN`, and values `0` and `''` evaluate to false, while everything else evaluates to true.

typeof: we consider the boolean operators `===` and `!==` with the form `[obj] === {undefined | null | | 0 | ''` `| NaN}` equivalent to inserting a `typeof` keyword.

callback: callback functions are commonly given similar names [16] (i.e., cb, callb, or callback), which we capture in our analysis.

error: we capture errors that are caught or used in the *error-first callback* [1] idiom.

Table 1 shows the concrete values we use for modelling changes in JavaScript and their respective abbreviations used throughout the paper.

3.5 Extracting Basic Change Types

Algorithm 1 shows a summary of our method for extracting BCTs from a list of subject programs ($\mathbb{P}$) into a relational database ($\mathbb{D}$) of {*commit, BCT*} pairs. For each commit (c) in a project's history ($\mathbb{C}$), we obtain the set of all modified source code files in the commit. This gives us a set $\mathbb{F} := \{f_{b1}, f_{r1}\}...\{f_{bn}, f_{rn}\}$ of n {buggy file, repaired file} pairs. For each pair in $\mathbb{F}$, we compute the BCTs that were made to the source code using *abstract syntax tree (AST)* differencing [11]. Because it considers the program structure when computing the changes between f_b and f_r, AST differencing is more accurate than line level differencing. It also computes fine-grained changes by labelling each node in the AST; this fine granularity is useful for learning BCTs.

The product of AST differencing is an AST for f_b (AST_b) and an AST for f_r (AST_r). For each $\{\text{AST}_b, \text{AST}_r\}$ pair, we extract the set of functions that occur in both AST_b and AST_r. We omit functions that were inserted or removed because we find these represent refactorings, not bug fixes. This gives us a set $\mathbb{M} := \{m_{b1}, m_{r1}\}...\{m_{bk}, m_{rk}\}$ of k {buggy function, repaired function} pairs.

For each pair in $\mathbb{M}$, we extract BCTs into set $\mathbb{T}$ by analysing the ASTs, whose nodes are annotated by the differencing

Algorithm 1: Basic Change Type Extraction

Input: $\mathbb{P}$ (subject programs)
Output: $\mathbb{D}$ (database of commits and change-types)
$\mathbb{D} \leftarrow \emptyset$
foreach $p \in \mathbb{P}$ **do**
  $\mathbb{C} \leftarrow \text{Commits}(p)$;
  foreach $c \in \mathbb{C}$ **do**
    $\mathbb{T} \leftarrow \emptyset$;
    $\mathbb{F} \leftarrow \text{ModifiedFiles}(c)$;
    foreach $\{f_b, f_r\} \in \mathbb{F}$ **do**
      $\{AST_b, AST_r\} \leftarrow \text{ASTDiff}(f_b, f_r)$;
      $\mathbb{M} \leftarrow \text{ModifiedFunctions}(AST_b, AST_r)$;
      foreach $\{m_b, m_r\} \in \mathbb{M}$ **do**
        $\mathbb{T} \leftarrow \mathbb{T} \cup \text{ExtractBasicChangeTypes}(m_b, m_r)$;
      end
    end
    $\mathbb{D} \leftarrow \mathbb{D} \cup \langle c, \mathbb{T} \rangle$;
  end
end

Table 2: An example dataset containing feature vectors for three commits. There are three change types in this feature vector. Change-type headers have the form [Type]_[Context]_[Change]_[Name]. The number beside the change-type header is the value for that feature–the number of occurrences of the change type in the commit. See Table 1 for short forms.

Proj.	Commit	Modified Statements	Change-type Header
async	63b	1	B_C_I_falsey [1]
			B_C_R_falsey [0]
			M_MC_I_bind [0]
bower	2d1	4	B_C_I_falsey [1]
			B_C_R_falsey [2]
			M_MC_I_bind [0]
express	5c4	2	B_C_I_falsey [0]
			B_C_R_falsey [0]
			M_MC_I_bind [1]

tool with the modification performed by the commit (i.e., inserted, updated or removed). Each c has a one-to-many relationship between it and the BCTs found within it (one c contains zero or more BCTs) and is stored in a relational database. Each c will be a feature vector in our dataset eventually, but it is convenient to store the dataset in a database which we can query.

Once the database is populated, we filter out noise by selecting candidate commits that (1) have between one and six modified statements, (2) do not contain the text 'merge' in the commit message, (3) contain at least one BCT that does not have language construct context 'S', and (4) has at least one BCT that does not have modification type 'U'. We express this query in Datalog and build a dataset from the results of this query.

Each BCT in the query results is converted to a feature, while each commit in the query results is converted to an instance of a feature vector in the dataset. Table 2 shows an example of three feature vectors. The feature vector includes the the number of modified statements and a bag of words, where the features are the BCTs and the values are the number of times the BCT occurs in the commit. The first commit (from the async project) shows the feature vector of the change type from Fig. 1. It contains one BCT that states that one instance of the behaviour (B) `falsey` used in a branch condition (C) was inserted (I). The other two feature vectors pertain to different change types.

3.6 Clustering and Ranking

We can now obtain a list of change types that occur frequently by clustering the dataset obtained in the previous step (Section 3.5). The clustering algorithm we use is DB-SCAN [9], because (1) it is a density-based algorithm, i.e., it groups feature vectors that are closely related, and (2) unlike other clustering algorithms, it does not require the number of clusters to be provided in advance as an input. We use Manhattan distance as our distance function, because it computes shorter distances between commits with the same BCTs. Change types are ranked by the number of projects that are represented and by the number of commits they contain. We expect change types with more projects to contain cross-project bug patterns and change types with more commits to contain frequently occurring bug patterns.

3.7 Change Type Inspection

For our manual inspection of change types, we give priority to larger change types and only inspect change types that contain five or more commits and are represented by more than one project. We use a systematic inspection process, based on grounded theory, to infer bug patterns from change types, based on their faults and repairs. This approach is described below.

Sampling. From each change type, we randomly select one commit from each project to be part of that change type's sample. For example, if a change type contains 100 commits from 40 projects, the sample size of that change type will be 40. We choose this sampling method because our goal is to find cross-project bug patterns. Selecting multiple items from the same project may bias our results towards projects with more bugs.

Summarizing. In order to help the inspectors understand the change, we open the summary of the commit on GitHub in a web browser. The summary includes a line level diff of the commit, the commit message, and links to the bug report (if any). Using the summary, we record a description of the bug pattern (i.e., the fault and repair for the commit).

Grouping. From these descriptions we group bug pattens. We compare the bug pattern of each commit to existing groups. If the description of a commit's bug pattern closely matches the descriptions of the bug patterns in an existing group, we place the commit in that pattern. If no bug pattern contains a similar fault and repair description, we create a new bug pattern group.

Reviewing. Once all commits in the change type sample have been placed into a group, we iteratively review patterns, merging or splitting them according to their descriptions.

3.8 Implementation

We implement our technique in a tool called BUGAID, which is publicly available [3].

Project histories are explored using JGit [23], a Git client for Java. JavaScript ASTs are created and explored using a fork of Mozilla Rhino [37], a JavaScript parser and runtime environment that we modify to better support AST change classification. AST differencing is performed using a fork of GumTree [11], a fine-grained AST differencing tool that we modify to better support JavaScript AST change

Table 3: Evaluation Subject Systems. KLoC indicates thousands of JavaScript lines of code, excluding comments and empty lines.

Measure	MediacenterJS	PM2	Total
Size KLoC	39	16	55
Stars on GitHub	11,839	1,002	12,841
Commits	1,770	2,549	4,319
Candidate Commits	1,260	1,617	2,877

classification. Clustering is performed using Weka [18].

To assist in the manual inspection of change types, we implement a script that randomly selects commits from a change type and displays the GitHub commit summary in a web browser.

4. CLUSTER EVALUATION AND TUNING

We evaluate our feature vectors and tune the parameters of our clustering algorithm, DBSCAN. We use two popular Node.js projects as evaluation subjects. MediacenterJS is a home media server that we selected randomly from a web search of popular Node.js applications. PM2 is a process manager and load balancer that we selected randomly from the `npm` homepage.

4.1 Gold Standard

To assess the accuracy of our clustering approach, we create a *gold standard* of cross-project bug patterns by manually inspecting all commits of the two subject systems, where [1–6] statements are modified. We classify the bugs repaired by each bug fixing commit into bug pattern categories. We categorize both the fault and repair of each bug and determine if a bug pattern meets our definition of a cross-project bug pattern (see Definition 1).

Our classification yields four-cross project bug patterns with at least three instances:

TypeError Undefined A variable can be undefined but is dereferenced. It is repaired by adding a branch condition that checks if the variable points to the `undefined` object.

TypeError Falsey A variable can evaluate to falsey, but is dereferenced. It is repaired by adding a branch condition that checks if the variable evaluates to false when used as a boolean.

Error Handling A method call may throw an error that is uncaught. It is repaired by surrounding the method call with a try statement.

Incorrect Comparison A compare-by-value operator (i.e. `==` or `!=`) is too permissive. It is repaired by replacing the compare-by-value operator with a compare-by-type-and-value operator (i.e. `===` or `!==`).

4.2 Comparison

As discussed in Section 3.2, change type classification tools already exist for Java. While we believe our approach is uniquely suited to identifying unknown bug fixing changes and could outperform existing change classification approaches for this specialized task, existing tools for Java cannot easily be converted to JavaScript. Therefore, a direct comparison is not possible. Instead, we compare our feature vector of BCTs learned with language constructs to two alternative approaches based on creating a feature vector of

more naive BCTs built from AST node types. This gives us three data sets to evaluate (one dataset for each feature vector):

**Dataset 1: Language Constructs** BCTs are the same as those described in Section 3.3.

Dataset 2: Statements BCTs are statement types (e.g., for loops, expression statements, function declarations, etc.) and how they are modified (i.e., inserted, removed, updated or moved).

Dataset 3: Nodes BCTs are AST node types (e.g., simple names, literals, expressions, etc.) and how they are modified (i.e., inserted, removed, updated or moved).

4.3 Evaluation Results

We cluster each of our three evaluation datasets into change types using various values of DBSCAN's density parameter epsilon, where the distance between data points in the same cluster must be less than epsilon. Smaller values of epsilon result in denser clusters. Because we assign the modified statements feature a weight of 0.2 and BCT features a weight of 1, values of epsilon that are less than 1 mean that commits in the same cluster have the same set of BCTs, but may differ in the number of modified statements. The first value of epsilon, 0.1, means that commits in the same cluster have the same feature vector.

Each set of change types for a dataset/epsilon pair is evaluated against our gold standard. We consider a cross-project bug pattern captured by a change type if there are two or more instances of the pattern in the change type. This would allow a human to identify the pattern if they manually inspected the change type. The results of the evaluation are shown in Fig. 2.
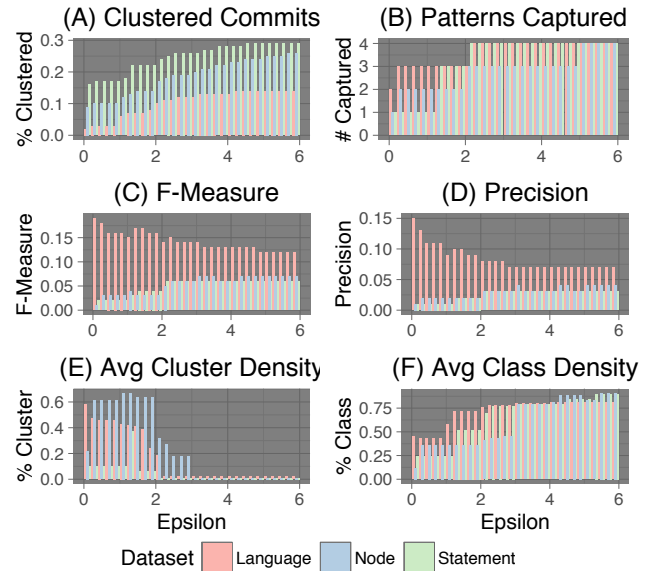
Figure 2: Evaluation and tuning results. In each chart, the x-axis shows the values of epsilon − the DBSCAN density parameter.

(A) Shows the percentage of commits that were clustered relative to the number of commits in the dataset that

148

meet the modified statement requirements. Lower is better in this case because most commits are unrelated and should not be clustered. A low value also means fewer commits need to be inspected by a human.

(B) Shows the number of cross project bug patterns in the gold standard that would be captured if a human manually inspected all change types. A y-value of 4 means all patterns would be captured.

(C) and (D) Show precision and recall. An instance is recalled if it is found with at least one other instance in a cluster. In all cases higher is better.

(E) Shows the average percent of cross-project bug patterns that make up their respective change type. A high value means the pattern is more likely to be captured by a random sample of the change type.

(F) Shows the average percent of cross-project bug patterns that are included in a change type. A high value means an estimate of the pattern frequency is more likely to be correct.

When picking a feature vector and epsilon value, we are interested in how much we can reduce the number of commits we have to inspect while still capturing all frequently occurring bug patterns. Charts (A), (B) and (E) are most relevant for this purpose. Our dataset of language constructs significantly outperforms the other datasets with respect to the number of commits that need to be inspected (A), while being competitive with the dataset of AST-node changes with respect to patterns captured (B) and cluster density (E). An epsilon value of between 0.3 and 0.9 seems to give us the largest search space reduction while still maintaining good recall.

It is worth mentioning that while precision and recall is significantly better for the dataset of language constructs, it is low relative to what one might expect from some other clustering task. Precision seems low because there are many bug patterns that were not identified in our manual inspection. Because clustering discovers unknown relationships, this is not surprising. These patterns will be discusses in the next section. A better measure of precision with respect to cross-project patterns is the average cluster density in chart (E). Recall is also low for small values of epsilon. This indicates that our approach is susceptible to noise. For example, the *Incorrect Comparison* pattern has only three instances. Two of the instances fix more than one bug in the same commit, which causes the distance between the instances to increase. We rely on a large dataset to mitigate the effects of low recall and sensitivity to noise.

Overall, our proposed approach based on language constructs outperforms the two alternatives. Based on our evaluation results, we choose an epsilon value of 0.3 as the basis of our feature vector. This value of epsilon yields the lowest number of commits to inspect and the highest F-Measure and cluster density while still capturing 3/4 patterns.

4.4 Clustering Example

Fig. 3 shows a breakdown of the commits for our evaluation dataset using the language construct feature vector and epsilon=0.3. The treemap on the left represents all 4,319 commits in PM2 and MediacenterJS. The box labelled *Merge* represents the 637 commits removed because they merge two branches. The box labelled *Outside Modified Statement Bounds* represents the 2,803 commits removed
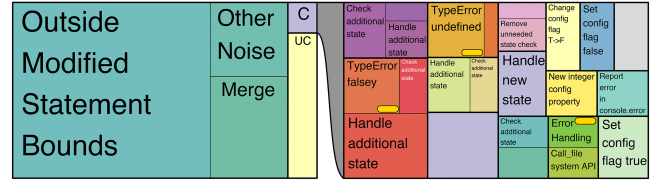
Figure 3: A treemap of the search space reduction (left) and the clusters and their patterns (right). Patterns from the gold standard are marked with a gold bar.

because they do not meet the modified statement requirements of repetitive changes. The box labelled *Other Noise* represents the 465 commits removed because they do not meet other requirements from the query filter. The box labelled *UC (unclustered)* represents the 342 commits removed because they were not present in a change type with size >= 3. The smallest box labelled *C (clustered)* represents the 72 commits which were clustered into change types.

The 72 clustered commits are expanded into the treemap on the right, which shows a detailed breakdown of the change types. Each colour shows one change type which is broken down into bug patterns discovered within the change type. In addition to the bug patterns identified in our gold standard, we inspect all change types and label additional bug patterns which we didn't identify in our manual inspection. For example, the largest change type in red at the bottom left of the treemap shows all instances that contain the feature B_C_I_falsey[1-2] and 1 or 2 modified statements. An inspection of this change type yields three bug patterns: one identified in our manual inspection (TypeError falsey), and two where the fault is that a program state is unchecked or unhandled. The patterns not identified in our manual inspection are arguably less relevant to cross-project tools, but still interesting.

5. EMPIRICAL STUDY

In this section, we present our empirical study in which we address the following research questions:

RQ1 What basic change types exist in JavaScript?
RQ2 What change types exist in JavaScript?
RQ3 What pervasive bug patterns exist in JavaScript?

Recall that BCTs are features in our feature vector, each cluster represents a change type and bug patterns are inferred from a manual inspection of change types.

5.1 Subject Systems

Our subject set contains 134 projects (70 packages and 64 applications) and 121,296 commits to analyse.

A JavaScript project is either a package (also called a module) used in other projects, or an application, which is normally executed as standalone software. We use both in our study, but search for them in different ways.

For packages, we use `npm`,[2] a popular package manager for Node.js. `npm` has become the largest software package repository with over 250,000 packages, now surpassing Maven Central and RubyGems. The `npm` website provides lists of

---

[2] https://www.npmjs.com
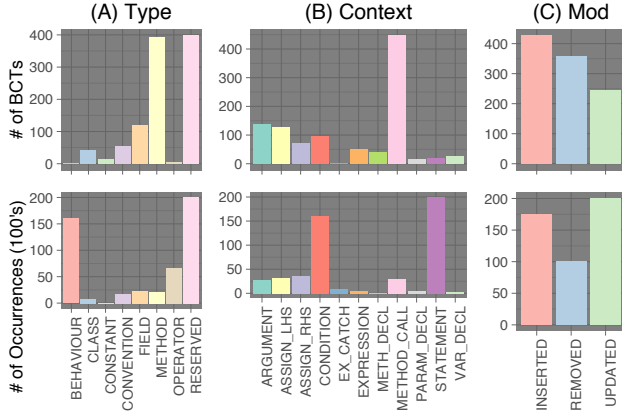
Figure 4: The distribution of the 1,031 BCTs discovered by our analysis over the feature properties from Table 1. Column (A) shows the distribution over language construct types. Column (B) shows the distribution over language construct contexts. Column (C) shows the distribution over modification types.

Table 4: Subject Systems. KLoC indicates thousands of JavaScript lines of code, excluding comments and empty lines. Downloads are over the last month, extracted from npm official website (only available for modules).

Measure	Average	Median	Total
Size KLoC	18	4	2,459
Stars on GitHub	3,296	524	441,631
Downloads x10^3	1,958	120	262,503
Commits	905	257	121,296
Feature Vectors	–	–	105,133
Analysis Time	6m10s	4s	4h32m

the most depended-upon packages [43] and most starred-by-users packages [44]. From each of the two lists above, we take the top 50 packages and merge them into a single list. We remove 27 duplicates which occur in both lists and three projects which are written in CoffeeScript, resulting in a final set of 70 modules. All these use Git and are hosted on GitHub. In total, these 70 modules have been downloaded more than 262 million times over the last month and have around 425,000 stargazers (number of users that starred the GitHub repository) on GitHub.

For finding popular Node.js applications, we rely on lists curated by users [42, 45] that collect popular Node.js applications. After we remove duplicates and projects where commit messages are not written in English, we end up with a list of 64 Node.js applications. These applications have more than 17,000 stargazers on GitHub. Table 4 summarizes some of the characteristics of the 134 JavaScript projects we use in our study.

5.2 Feature Extraction (RQ1)

We use BugAID to extract BCTs from each commit and build the database of {*commit, BCT*} relations. BugAID is run on a server with two Intel® Xeon® E5-2640 CPUs and 64GB of RAM. Table 4 shows the time to extract the BCTs
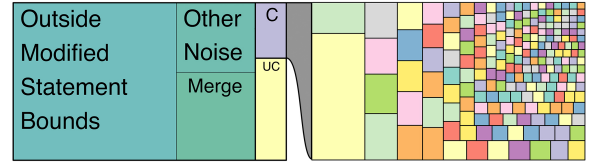
Figure 5: Of the original 105,133 commits, filtering and clustering produces 219 clusters containing 4,179 commits.

from the commits of the subject projects. Four projects did not complete after 24 hours because GumTree AST differencing did not complete. These projects are omitted from the analysis time statistics. The number of feature vectors are less than the total number of commits because the analysis for each project is done in a single thread, so the analysis missed some commits from projects where GumTree did not complete. A few large projects take up most of the analysis time, with an average analysis time of 6m10s and a median analysis time of 4s. The total analysis time is 4h32m.

BugAID discovered 1,031 BCTs in the 105,133 commits we analyzed. Fig. 4 shows the distribution of the BCTs over the feature properties described in Section 3.3 and listed in Table 1. The first row shows the distribution over the BCTs themselves, while the second row shows the distribution over the BCT occurrences in the dataset. We do not show the distribution over each language construct name (e.g., var, falsey, etc.) because there are too many to fit within the paper.

The number and diversity of BCTs supports our claim that manually encoding all BCTs is impractical and that modelling the language in order to automatically learn BCTs is a more scalable approach. Because previous work required BCTs be manually defined, the number of change types, or the granularity of the results in studies using these tools may be limited.

The full list of BCTs is available in our dataset [3].

5.3 Clustering (RQ2)

We use BugAID to (1) build a dataset of commits based on the query in Section 3.5 and (2) cluster the commits in to change types.

Fig. 5 shows a breakdown of the commits. The tree map on the left shows that after filtering commits with our Datalog query, we are left with 11,928 commits to cluster (from a total of 105,133 commits). Clustering removes another 7,731 commits, leaving 4,197 commits in 219 clusters with size >= 5. The tree map on the right shows the size of each of the 219 clusters. The smallest clusters contain five commits.

Fig. 6 shows the 219 change types in more detail. change types are shown in two facets: the number of BCTs in the change type and the average number of modified statements (MS) in the change type. There are 85 change types with one BCT, 101 with two, 29 with three and 4 with four. Commits with fewer modified statements are more repetitive, so there are more change types with fewer modified statements.

Table 5 gives descriptive statistics of the change types in Fig. 6. For each change type, it includes the number of commits, the number of projects represented, the number of modified statements, and the number of BCTs. There are three change types with only one project represented, which

Table 5: Change types descriptive statistics

Measure	Min	Average	Median	Max
Commits	5	19	8	655
Projects	1	9	6	67
Modified Statements	1	1.7	1	6
Basic Change Types	1	1.8	2	4
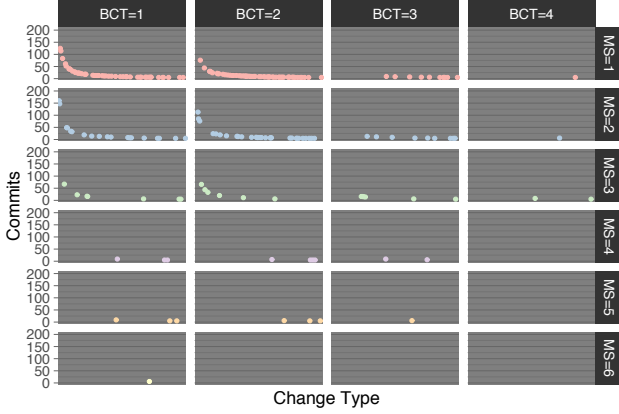
Figure 6: Clustering produced 219 change types. The BCT facet stores the number of BCTs in the change type. The MS facet stores the average number of modified statements in the change type.

are excluded from our inspection.

The change type with the most commits has the signature {B_C_-falsey_1}, which inserts one falsey check into a branch condition and has one modified statement on average. This is typically to repair a `TypeException` or to handle a missing edge case. As another example, the most frequent change type with four BCTs has the signature {R_S_I_var_1, B_C_I_falsey_1, R_AR_I_true_1, R_AR_I_false_1}, which declares a new variable, adds a falsey branch condition and adds one `true` and one `false` reserved word to the right hand side of assignments. When we inspect the commits with this change type, we see that this change type repairs a bug where a method should only be executed once.

The full list of change types is available in our dataset [3].

5.4 Pervasive Bug Patterns (RQ3)

Our manual inspection took two authors approximately 18 man hours to inspect the change types described in Section 5.3. Compared to an earlier inspection, where it took one author over five days to inspect approximately 500 commits from only two Node.js projects, our approach is able to greatly reduce the inspection time. We found that in most cases, the change types produced by BUGAID have very similar bug patterns, i.e., usually between one and three patterns per change type

We now provide a qualitative analysis of common bug patterns from our inspection. We discuss the bug patterns at a high level and omit discussing each change type individually. For example, a bug pattern where `null` or `undefined` could be dereferenced is present in multiple change types because it can be repaired in multiple ways, such as putting the dereference inside a conditional statement, exiting the function before the dereference occurs or initializing the variable. For

each group of bug patterns, their frequency, the associated change type (CT) ID, and a link to a representative commit are listed in our online dataset and tech report [3] for the interested reader.

The following are the inferred patterns that we believe are the most relevant to the development of tools and IDE support for detecting and preventing bugs in JavaScript code.

5.4.1 Dereferenced Non-Values

In JavaScript, there are two types, `null` and `undefined`, which indicate that a variable does not have a value. Dereferencing these objects produces a `TypeError`. A variable which is not declared or initialized points to the property `undefined`, while it is convention to assign the value `null` to a variable that has no value. The following repairs frequently occur for this bug pattern:

Protect with falsey check (9 CTs). In JavaScript, several types and values are coerced to the boolean value `false` when used as a boolean in a condition. These include `false`, `0`, `""`, `null`, `undefined` and `NaN` and are known as falsey values. This repair prevents the dereference of `null` or `undefined` values by checking if they are coerced to false.

```
1 + if (!target.parts) return;
2     part = target.parts[playingPart];
```

Protect with no value check (2 CTs). There are two different equivalence comparison operators in JavaScript. The first is the value comparison operator (== and !=), which checks if two variables have the same value. `null` and `undefined` are value-equivalent, because both are considered to have no value. This repair prevents the dereference of `null` or `undefined` values by checking if they do not have a value.

```
1 - if ( val <= 0 ) {
2 + if ( val <= 0 || val == null ) {
3     val = curCSS( elem, name );
```

Protect with type check (4 CTs). The second equivalence comparison operator is the value and type comparison operator (=== and !==), which checks if two variables have the same value and type. Types can also be checked with the `typeof` keyword. This repair prevents the dereference of `null` or `undefined` values by performing a type check with either a value-and-type comparison operator or the use of a `typeof` operator.

```
1 + if (typeof torLink === 'string') {
2     if (torLink.toLowerCase().match(...) != null) {
```

5.4.2 Incorrect Comparison

Because there are multiple ways (falsey, check-by-value or check-by-value-and-type) of checking equivalence in JavaScript, it is possible to use a comparison operator that permits too many types, or too few types to pass a branch condition.

Compare fewer types (3 CTs). A variable can be compared for multiple types by coercing it to a boolean or by using a check-by-value operator. In some cases, the omission of type equivalence causes an incorrect result for a branch condition. For example, if the variable `port` may be `undefined`, it is incorrect to prevent its use in a URL by checking if it is falsey. Such a comparison would also filter out port number zero, which is a valid port. The use of compare-by-value operators are discouraged for this reason, and linting tools typically generate a warning when it encounters one.

In most cases, the repairs we see address bad practice, however, we also see instances of bugs that result from the use of compare-by-value operators.

```
1 - if (typeof opt.default != 'undefined')
2 + if (typeof opt.default !== 'undefined')
3     self.default(key, opt.default);
```

Compare more types (1 CT). In the opposite scenario, a value may be encoded in different types (e.g., no-value can be expressed as `null` or `undefined`), but the comparison permits only one of those values.

```
1 - if (encoded_test === "TEST")
2 + if (encoded_test == "TEST")
```

5.4.3   Missing argument (2 CTs)

JavaScript functions are variadic, i.e., they can accept a variable number of arguments. This means JavaScript does not require that the arguments in a function call match the parameters in a function definition. Because of this, arguments may be accidentally omitted, resulting in an incorrect argument order. In the callee, this means the parameters will be out of order. This bug is often repaired by passing `null` in place of the missing argument.

```
1 if (completed === arr.length) {
2 -   callback();
3 +   callback(null);
```

5.4.4   Incorrect API configs (6 CTs)

One of the more common change types is the repair of API configurations. In this bug pattern, an API configuration option is not specified or not correct.

There are several factors that may contribute to this pattern. First, APIs can be configured in many different ways (e.g., through constructors, object literals, JSON, etc.), but very rarely are developers forced to specify a configuration option. In a statically typed language such as Java, the developer might be forced to assign values to fields through the constructor, which would be enforced at compile time. Second, many JavaScript IDEs do not support code completion, forcing a developer to manually look up expected options. Third, Node.js applications rely heavily on a large number of APIs maintained by different developers. Such variation in developers may contribute to a wide variety of APIs that may be non-trivial to configure.

```
1   grunt.initConfig({
2     clean: {
3 +     force: true,
4       build: ['dist']
```

5.4.5   `this` not correctly bound

In JavaScript functions, the reserved word `this` is bound to the object which first executes the function. Because of JavaScript's closure property, developers may incorrectly assume that `this` is bound to the object in which the function is defined.

Access `this` through closure (2 CTs). Because of JavaScript's closure property, a function declared inside another function can access the local variables of the parent function. This property can be used to eliminate the use of `this` inside a callback function, and avoid binding issues.

```
1 + var self = this;
2   self.serverConfig.connect(function(err, result) {
3     self._state = 'connected';
```

Bind `this` (1 CT). JavaScript contains three functions that explicitly bind `this` to an object before a function is called. These functions are `call`, `bind` and `apply`. This repair binds `this` using one of these three functions.

```
1 - _kiwi.global.panels = this.panels;
2 + _kiwi.global.panels = _.bind(this.panels, this);
```

5.4.6   Unhandled exception

Error handling in JavaScript can be messy and error-prone. Because JavaScript is often asynchronous and event-based, errors are propagated in many different ways.

Catch thrown exception (1 CT). In sequential JavaScript, where errors are propagated with a `throw` statement and caught with a `catch` statement, error handling is not enforced, which can lead to uhandled errors.

```
1 + try {
2     html = kiwi.jade.compile(str)(...);
3 + } catch (e) {
4 +   response.statusCode = 500;
```

Error not propagated to callback (1 CT). Callbacks and promises complicate the propagation of errors in JavaScript because in this asynchronous model, errors cannot be propagated up the stack [16]. Errors that must be explicitly propagated through callbacks or promises are often forgotten about.

```
1   Server.prototype.handle = function(outerNext) {
2     function next(err) {
3 -     outerNext();
4 +     outerNext(err);
```

Check callback exception (1 CT). Similar to the previous repair, errors given to callback functions are often left unchecked. This repair handles an error that has been passed to the callback function.

```
1   return fs.readFile(path, function(err, buffer) {
2 +   if (err) {
3 +     throw err;
```

6.   DISCUSSION

Due to space limitations, we have only presented a few of the more common bug patterns that exist in our dataset of change types. Many more exist, although building a generic tool to look for some of them might be prohibitively difficult because of their project-specific characteristics.

One interesting question is, of the bug patterns we identify, which are currently handled by existing tools? Type checking tools for JavaScript [10, 22, 49] have been a research focus recently; once adopted and integrated into IDEs they may help to reduce bug patterns such as those in *dereferenced non-values*. Linting tools such as JSHint[3] can also help prevent some, but not all of the bugs in *incorrect comparison*. Simple tools like argument-validator [6] may help prevent the *missing arguments* bug pattern. Still, techniques for preventing these bugs are far from mature. New ideas, implementations and integrations are needed to further reduce their prevalence.

For other bug patterns like *this not correctly bound*, *incorrect API configurations* and *unhandled exception*, there is a lack of tool support, to the best of our knowledge.

---

[3]http://jshint.com

We must also consider bugs we expected but did not find. For example, we found that prototype handling was often done very well, with changes to prototypes often reflecting changes to a module's API. This was often the best documented and well-formatted code. We did not encounter any repairs involving the use of `eval`, considered by many [21, 50] as a problematic language construct. While `eval` did appear in our list of BCTs, it did not appear in any change type. This likely indicates that `eval` is either used sparingly in Node.js JavaScript or it is not as error-prone as people might have expected.

Our findings may be used to (1) direct researchers towards creating and improving tools and techniques to prevent common bugs in JavaScript, (2) act as a guide to which tool support to implement in a JavaScript IDE, and (3) make developers aware of the common mistakes involved with programming in JavaScript so that they can adopt their programming style accordingly.

Threats To Validity. Threats to internal validity are: (1) the language model we use is not complete. There are language constructs which we did not include in our language model (e.g., the `<` operator). A better language model may discover more bug patterns, however, we believe that our language model includes the most important JavaScript constructs and likely captures many of the most pervasive bug patterns; (2) the subjects and change types used in tuning our clustering approach may have caused over-fitting. We attempted to mitigate this by keeping our language model simple and by using a large data set to mitigate the effects of a small epsilon; (3) we only inspect repairs which appear in a commit. Bugs that are repaired before a commit is merged into the main branch are not captured by our method.

Threats to external validity are: (1) the results may not extend to client-side JavaScript. Almost all of our subjects are server side Node.js projects under heavy use and development; (2) the results may not generalize to all Node.js projects. While this is unlikely for the most frequently occurring patterns, some results may be specific to our subject systems; (3) we discuss a subset of bug patterns based on our knowledge of program analysis and testing. There may be bug patterns that are more interesting or relevant to others, which is why we make the data set of bug patterns publicly available.

7. RELATED WORK

Mining change types. Fluri et al. [15] use hierarchical clustering to discover unknown change types in three Java applications. This approach is similar to ours, however, the basic change types they use are limited to the 41 BCTs identified in [14] and do not use language constructs. Unlike BUGAID, this work does not identify pervasive bug patterns. Negara et al. [38] use an IDE plugin to track fine-grained changes from 23 developers and infer general code change patterns. Livshits and Zimmerman [31] discover application-specific bug patterns (methods that should be paired but are not) by using association rule mining on two Java projects. Unlike BUGAID, this work is limited to method pairs only. Pan et al. [48] use line level differencing to extract and reason about repair patterns for automated program repair. The repair patterns they identify are coarse grained and do not identify the root cause of the bugs. Kim et al. [26] discover six common repair patterns in Java by grouping changes

with groums [41]. They suggest possible root causes for three of these patterns. However, the focus of their work is on discovering repair patterns for automated repair. They do not provide metrics or discussion on the root causes of the six repair patterns. It is unclear whether using groums for discovering change types is more effective than any of the approaches previously discussed. An implementation of the groum mining tool is not available (and may not be feasible) for analyzing JavaScript.

In our experience, these techniques have not been able to group repair patterns tightly enough to easily infer the root cause of each bug. This limitation from our perspective may not be relevant to automated repair research, however, it is worth considering if a more fine grained grouping of repair patterns will enhance automated repair approaches.

Mining bug patterns. Li et. al. group bugs by classifying the natural language content of bug reports [30]. This is useful for inferring what kinds of bugs are being repaired at a high level, but does not give us the exact source code changes that are taking place. Further, because many bugs are repaired without records in a bug report, this technique may miss important bug patterns.

JavaScript bug patterns. There have been very few studies which investigate bug patterns in dynamic languages. Ocariza et al. [46] manually inspect bug reports pertaining only to *client-side* JavaScript code. They find that the majority of reported JavaScript bugs on the client-side are DOM-related, meaning the fault involves a DOM API call in JavaScript code. We apply our technique to JavaScript applications and server-side JavaScript code (in which the DOM is non-existent), to detect and analyze bug patterns that are inherent to the JavaScript language. In addition, our study includes 134 JavaScript applications, making it the largest empirical study to find common JavaScript bug patterns.

8. CONCLUSION AND FUTURE WORK

We proposed BUGAID, a data mining technique for discovering common unknown bug patterns. We showed that language construct changes are closely related to bug patterns and can be used to group bug fixing commits with similar faults and repairs. Using BUGAID, we performed an analysis of 105,133 commits from 134 server-side JavaScript projects. We discovered 1,031 BCTs and 219 change types. From our inspection of change types, we discussed 13 groups of pervasive bug patterns that we believe are among the most pressing issues in JavaScript.

With a better language model, weighing of features and enough data, it is feasible in future work that the density of the clusters could be increased so that almost all clusters include only one bug pattern. If such a result could be achieved, our approach to discovering pervasive bug patterns will be completely automated, with human intervention only required to create natural language descriptions of the bug patterns. This could have implications for a number of research areas including automated repair and defect prediction.

9. ARTIFACT DESCRIPTION

In this section we describe the artifact (dataset and tool) that accompanies this paper. The purpose of the artifact is to enable replication of the evaluation and empirical study,

and to allow users to explore some of the bug patterns that exist in JavaScript in greater detail than could be presented in this paper. It has multiple components including: an executable for reproducing or expanding the dataset, the list of subjects, the raw data, the list of basic change types (BCTs) and the list of change types. In addition to the raw data and executable, for each component we provide a graphical web interface for exploring the data.

9.1 Executable and Subjects

We provide an executable and a list of git repositories for reproducing or expanding the results of our empirical study. The executable has two main classes: the first builds the raw dataset of $\langle commit,BCT \rangle$ relations by mining the repository histories; the second filters the data and clusters the commits into change types. Installation and useage instructions are available in the executable's README [2].

The list of git repositories is the same as what we used in our empirical study. Detailed information about these repositories is available on the artifact's web page [3].

9.2 Raw Data

Our commit mining process created $\langle commit,BCT \rangle$ relations for 105,133 commits. The dataset is available as a CSV file which is downloadable from the artifact's web page. Each row contains the commit ID, the number of modified statements in the commit and a list of BCTs that are present in the commit. It can be used in lieu of regenerating the dataset with the executable or to perform alternate data mining tasks.

9.3 Basic Change Types

A BCT is the smallest unit of change in our method. The list of 582 unique BCTs in the raw data and the number of occurrences of each BCT is available through a searchable interface on the artifact's web page. The dataset is useful for looking up the relative frequency of BCTs. For example, the most frequently inserted call to a JavaScript API method is *replace*, which was inserted 269 times in our subjects. By contrast, the builtin method *eval* was only inserted 11 times.

9.4 Change Types

Change types are groups of commits which share a similar set of BCTs and number of modified statements. The list of 219 change types discovered in our empirical study is available as an `.arff` file and through a searchable interface on the artifact's web page.

The dataset is useful for exploring bug patterns that exist for JavaScript. In our empirical study, we highlighted 12 bug patterns that we believe are good candidates for detection using static analysis tools. There are many more patterns that either occurred less frequently in our dataset or that require project-specific or API-specific knowledge to diagnose. Because there are alternate applications for which these patterns may be relevant, we provide an interface for exploring change types in the same manner that we used in our empirical study.

We suggest the following method for investigating a particular class of bug:

1. Identify which BCTs might be included with the class of bug under investigation.
2. Use the interface on the artifact's web page to find change types that contain these BCTs.

3. For each change type, use the interface to inspect the commits inside the change type.

For example, assume we are interested in bugs associated with using callbacks. First, we identify relevant BCTs. In this case they are BCTs that change a `callback` convention token, such as the BCT *CV_MC_L_callback*. Next, we search for clusters that contain this BCT. Cluster #3 includes this BCT. Finally, we inspect the commits in cluster #3 by opening the commit summaries on GitHub using the provided links. We observe that in general, the commits in cluster #3 repair an error caused by unchecked state by returning control through a callback function.

9.5 Evaluation Data

The raw data used in the evaluation – both $\langle commit,BCT \rangle$ relations and change types – is available as a download from the artifact's web page. It can be used to replicate the evaluation.

10. REFERENCES

[1] Error Handling in Node.js. https://www.joyent.com/developers/node/design/errors, 2014.

[2] BugAID: Source Code. https://github.com/saltlab/bugaid, 2016.

[3] BugAID: Toolset and Dataset. http://salt.ece.ubc.ca/software/bugaid/, 2016.

[4] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 321–345, 2015.

[5] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[6] argument-validator. argument-validator: Simple argument validator for javascript. https://www.npmjs.com/package/argument-validator, 2016.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130. ACM, 2009.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72. ACM, 2001.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[10] Facebook Inc. flow: a static type checker for JavaScript. http://flowtype.org/, 2015.

[11] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324. ACM, 2014.

[12] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of the International Conference on Software Engineering*, ICSE'13, pages 752–761. IEEE Press, 2013.

[13] FindBugs. FindBugs - find bugs in Java programs. http://findbugs.sourceforge.net/, 2015.

[14] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[15] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 463–466. IEEE Computer Society, 2008.

[16] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.

[17] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105. ACM, 2015.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[19] Q. Hanam, L. Tan, R. Holmes, and P. Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 152–161. ACM, 2014.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[21] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 34–44. ACM, 2012.

[22] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[23] JGit. JGit. https://eclipse.org/jgit/, 2015.

[24] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 672–681. IEEE Press, 2013.

[25] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132. ACM, 2014.

[26] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811. IEEE Press, 2013.

[27] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[28] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45. ACM, 2006.

[29] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a Google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381. IEEE Press, 2013.

[30] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33. ACM, 2006.

[31] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of ESEC/FSE*, pages 296–305. ACM, 2005.

[32] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[33] N. Meng, M. Kim, and K. S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the International Conference on Software Engineering*, ICSE'13, pages 502–511. IEEE Computer Society, 2013.

[34] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proceedings of the International Working Conference onSource Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[35] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. ACM.

[36] Mozilla. MDN JavaScript pages. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Index, 2015.

[37] Mozilla Developer Network. Rhino. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino, 2015.

[38] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813, New York, NY, USA, 2014. ACM.

[39] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 180–190, Nov 2013.

[40] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 518–529. ACM, 2014.

[41] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of ESEC/FSE*, pages 383–392. ACM, 2009.

[42] Nodejitsu blog. Blog post: npmawesome: Ten open source Node.js apps. http://blog.nodejitsu.com/ten-open-source-node-js-apps/, 2015.

[43] npm. Most depended-upon packages. https://www.npmjs.com/browse/depended, 2015.

[44] npm. Most starred packages. https://www.npmjs.com/browse/star, 2015.

[45] nw.js. List of apps and companies using nw.js. https://github.com/nwjs/nw.js/wiki/List-of-apps-and-companies-using-nw.js, 2015.

[46] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering (TSE)*, page 17 pages, 2016.

[47] F. Ocariza, K. Pattabiraman, and A. Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015.

[48] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[49] M. Pradel, P. Schuh, and K. Sen. Typedevil: Dynamic type inconsistency analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*, 2015.

[50] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78. Springer, 2011.

[51] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1–12. ACM, 2010.

[52] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, 2015.

[53] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458. Springer-Verlag, 2012.

[54] Stack Overflow. 2015 Developer Survey. http://stackoverflow.com/research/developer-survey-2015, 2015.

[55] B. Sun, G. Shu, A. Podgurski, and B. Robinson. Extending static analysis by mining project-specific rules. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1054–1063, 2012.

[56] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 2013.

[57] W3schools. JavaScript reserved words. http://www.w3schools.com/js/js_reserved.asp, 2015.