An Extended Empirical Study of Code Smells In JavaScript Projects

David Johannes, Amir Saboury, Pooya Musavi, Foutse Khomh and Giulio Antoniol
Polytechnique Montreal, Quebec, Canada
{david.johannes, amir.saboury, pooya.musavi, foutse.khomh, giuliano.antoniol}@polymtl.ca

***Abstract*—JavaScript is a powerful scripting programming language that has gained a lot of attention this past decade. Initially used exclusively for client-side web development, it has evolved to become one of the most popular programming languages, with developers now using it for both client-side and server-side application development. Similar to applications written in other programming languages, JavaScript applications contain *code smells*, which are *poor* design choices that can negatively impact the quality of an application. In this paper, we investigate code smells in JavaScript server-side applications with the aim to understand how they impact the fault-proneness and the vulnerability of applications, and how they survive all along the projects. We detect 12 types of code smells in 1807 releases of fifteen popular JavaScript applications (*i.e.,* express, grunt, bower, less.js, request, jquery, vue, ramda, leaflet, hexo, chart, webpack, webtorrent, moment, and riot) and perform survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. We then do the same survival analysis, but with a line grain approach (wich means considering the lines where the code smells and the potential bugs appear), and with a line grain approach including dependencies (which means considering the lines where functions, objects, variables are called). We also perform file grain, line grain, and line grain including dependencies survival analysis, comparing the time until a vulnerability appears. Finally, we perform a survival analysis on code smells to know how long they survive. Results show that (1) on average, files without code smells have hazard rates 76% lower than files with code smells in our file grain analysis, 20% lower in our line grain analysis, and 38% lower in our line grain analysis considering dependencies. (2) Among the studied smells, "Variable Re-assign", "Assignment In Conditional statements", and "Complex Code" smells have the highest fault hazard rates. (3) Files without code smells are not necessarely less vulnerable than files with code smells, but this conclusion needs to be mitigated because of the weaknesses of our vulnerability database. (4) Among the studied smells, "Variable Re-assign" and "This Assign" have the highest vulnerability hazard rates. (5) Code smells, and particularly "Variable Re-assign", tend to be created at the file creation, are not enough removed from the system, and have a high chance of surviving a very long time after their introduction; "Variable Re-assign" is also the most proliferated code smells. Additionally, we conduct a survey with 1,484 JavaScript developers, to understand the perception of developers towards our studied code smells. We found that developers consider "Nested Callbacks", "Variable Re-assign" and "Long Parameter List" code smells to be serious design problems that hinder the maintainability and reliability of applications. This assessment is in line with the findings of our quantitative analysis. Overall, code smells affect negatively the quality of JavaScript applications and developers should consider tracking and removing them early on before the release of applications to the public.**

I. Introduction

"Any application that can be written in JavaScript, will eventually be written in JavaScript."
— Jeff Atwood —

JavaScript is a highly dynamic scripting programming language that is becoming one of the most important programming languages in the world. Recent surveys by Stack Overflow [1] show JavaScript topping the rankings of popular programming languages for four years in a row. Many developers and companies are adopting JavaScript related technologies in production and it is the language with the largest number of active repositories and pushes on Github [2]. JavaScript is dynamic, weakly-typed, and has first-class functions. It is a class-free, object-oriented programming language that uses prototypal inheritance instead of classical inheritance. Objects in JavaScript inherits properties from other objects directly and all these inherited properties can be changed at run-time [3]. This trait can make JavaScript programs hard to maintain. Moreover, JavaScript being an interpreted language, developers are not equipped with a compiler that can help them spot erroneous and unoptimized code. As a consequence of all these characteristics, JavaScript applications often contain code smells [4], *i.e.,* poor solutions to recurring design or implementation problems. However, despite the popularity of JavaScript, very few studies have investigated code smells in JavaScript applications, and to the best of our knowledge, there is no work that examines the impact of code smells on the fault-proneness and on the vulnerability of JavaScript applications. This paper aims to fill this gap in the literature. Specifically, we detect 12 types of code smells in in 1807 releases of fifteen popular JavaScript applications (*i.e.,* express, grunt, bower, less.js, request, jquery, vue, ramda, leaflet, hexo, chart, webpack, webtorrent, moment, and riot) and perform survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. We then do the same survival analysis, but with a line grain approach (wich means considering the lines where the code smells and the potential bugs appear), and with a line grain approach including dependencies (which means considering the lines where functions, objects, variables are called). We also perform file grain, line grain, and line grain including dependencies survival analysis, comparing the time until a vulnerability appears. Finally, we perform a survival analysis on code smells to know how long they survive. We address the following five research questions:

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell? Previous works [5], [6] have found that code smells increase the risk of faults in Java classes. In this research question, we compare the time until a fault occurrence in JavaScript files that contain code smells and files without code smells, computing their respective hazard rates. Results show that on average, across our fifteen studied applications, JavaScript files without code smells have hazard rates 76% lower than files with code smells in our file grain analysis, 20% lower in our line grain analysis, and 38% lower in our line grain analysis considering dependencies.

(RQ2) Are JavaScript files with code smells equally fault-prone? A major concern of developers interested in improving the design of their application is the prioritization of code and design issues that should be fixed, giving their limited resources. This research question examines faults in files affected by different types of code smells, with the aim to identify code smells that developers should refactor in priority. We do this research through our file grain, line grain, and line grain including dependencies analysis. Our findings show that "Variable Re-assign", "Assignment in Conditional Statements", and "Complex Code" smells are consistently associated with high hazard rates across the fifteen studied systems. Developers should consider removing these code smells, in priority since they make the code more prone to faults. We also conducted a survey with 1,484 JavaScript developers, to understand the perception of developers towards the 12 studied code smells. Results show that developers consider "Nested Callbacks", "Variable Re-assign" and "Long Parameter List" code smells to be the most hazardous code smells. Developers reported that these code smells negatively affect the maintainability and reliability of JavaScript applications.

(RQ3) Is the risk of vulnerability higher in files with code smells in comparison with those without code smell? Similarly to RQ1, we compare the time until a vulnerability appears in JavaScript files that contain code smells and files without code smells, computing their respective vulnerability hazard rates. Results show that on average, the risk of vulnerability is not higher in files with code smells in comparison with those without code smells. However, this conclusion has to be mitigated, especially because the vulnerabilities's database is not complete and presents a lack of accuracy.

(RQ4) Are JavaScript files with code smells equally vulnerable? Similarly to RQ2, we examine vulnerabilities in files affected by different types of code smells, with the aim to identify code smells that developers should refactor in priority. Results show that "Variable Re-assign" and "This Assign" code smells tend to make the code more vulnerable than the other code smells, and thus should be removed in priority.

(RQ5) How do the smells survive over time? It is interesting to know how long the smells of a project survive, when they are introduced (at the creation of a file or during a revision), and what type smell are likely to survive the most. Indeed, having a specific knowledge on the smells of a project could help us to determine what smell types are the most dangerous. Results show that smells are created at the file birthdate, and are persistant because a considerable proportion still survive today in the studied systems, and because they have a high chance to survive even a very long time after their introduction into the codebase. Especially, "Variable Re-assign" is the most sizable code smells with one of the highest probability of surviving over time, and thereby we strongly recommend to developers to remove this code smells, at least to reduce their number.

The remainder of this paper is organized as follows. Section II describes the type of code smells we used in our study. Section III describes the design of our case study. Section IV presents and discusses the results of our case study. SectionV presents and discusses the results of our qualitative study. Section VII discusses the limitation of our study. Section VIII discusses related works on code smells and JavaScript systems, while Section IX concludes the paper.

II. Background

To study the impact of code smells on the fault-proneness and the vulnerability of server-side JavaScript applications, and to study the smells's survival, we first need to identify a list of JavaScript bad practices as our set of code smells. Hence, we select the following 12 popular code smells from different JavaScript Style Guides [3], [7]–[11].

1) Lengthy Lines: Too many characters in a single line of code would decrease readability and maintainability of the code. Lengthy lines of code also make the code review process harder. There are different limits indicated in different JavaScript style guides. NPM's coding style [7] and node style guide [8] suggest that 80 characters per line should be the limit. Airbnb's JavaScript style guide [9] which is a popular one with around 42,000 Github stars, suggests a number of characters per line of code less than 100. Wordpress's style guide [12] encourages jQuery's 100-character limit [10]. All the style guides include white spaces and indentations in the limit. As mentioned in jQuery's style guide, there are some cases that should be considered exceptions to this limit: (i) comments containing long URLs and (ii) regular expressions [10].

2) Chained Methods: Method chaining is a common practice in object-oriented programming languages, that consists in using an object returned from one method invocation to make another method invocation. This process can be repeated indefinitely, resulting in a "chain" of method calls. The nature of JavaScript and its dynamic behavior have made creating chaining code structures very easy. jQuery[1] is one of the many libraries utilizing this pattern to avoid overuse of temporary variables and repetition [13]. Chained methods allow developers to write less code. However, overusing chained methods makes the control flow complex and hard to understand [3].

[1]jquery.com

Below is an example of chained methods from a jQuery snippet:

```
$('a').addClass('reg-link')
      .find('span')
      .addClass('inner')
      .end()
      .end()
      .find('div')
      .mouseenter(mouseEnterHandler)
      .mouseleave(mouseLeaveHandler)
      .end()
      .explode();
```

3) Long Parameter List: An ideal function should have no parameters [14]. Long lists of parameters make functions hard to understand [15]. It is also a sign that the function is doing too much. The alternatives are to break functions into simpler and smaller functions that do more specific tasks or to create better data structures to encapsulate the data. To handle a large amount of configurations passing to functions, JavaScript developers tend to use a single argument containing all the configurations. This is a better practice since it eliminates the order of parameters when the function calls, and it is easier to add more parameters later on while maintaining the backward compatibility. Below are examples of this code smell and suggested refactorings.

```
// considered bad
function distance(x1, y1, x2, y2) {
  return Math.sqrt(Math.pow(x1-x2, 2) +
      Math.pow(y1-y2, 2));
}

// alternative
function distance(p1, p2) {
  return Math.sqrt(Math.pow(p1.x-p2.x, 2) +
      Math.pow(p1.y-p2.y, 2));
}
```

```
// considered bad
function send(from, to, subject, body) {
  // ...
}

// alternative
function send(options) {
  // using options.from, options.to
  //      options.subject, options.body
}
```

4) Nested Callbacks: JavaScript I/O operations are asynchronous and non-blocking [16]. Developers use callback functions to execute tasks that depend on the results of other asynchronous tasks. When multiple asynchronous tasks are invoked in sequence (*i.e.*, the result of a previous one is needed to execute the next one), nested callbacks are introduced in the code [17], [18]. This structures could lead to complex pieces of code which is called "callback hell" [3], [17], [19]. There are several alternatives to nesting callback functions like using Promises [17] or the newest ES7 features [20]. Below is an example of Nested Callbacks smell and an alternative implementation that uses Promises.

```
// considered bad
db.getUser({id: 1}, function (user) {
  twitter.getTweets({handle: user.twitter}, function (tweets) {
    sendEmail(tweets, function (done) {
      console.log('Done')
    })
  })
})

// Alternative implementation using Promises
db.getUser({id: 1})
  .then(function (user) {
    return twitter.getTweets({handle: user.twitter});
  })
  .then(function (tweets) {
    return sendEmail(tweets);
  })
  .then(function() {
    console.log('Done')
  })
```

5) Variable Re-assign: JavaScript is dynamic and weakly-typed language. Hence, it allows changing the types of the variables at run-time, based on the assigned values. This allows developers to reuse variables in the same scope for different purposes. This mechanism can decrease the quality and the readability of the code. It is recommended that developers use unique names, based on the purpose of the variables [3]. Below is an example of Variable Re-assign code smell and a suggested refactoring.

```
// considered bad
function parse(url) {
  url = url.split('/'); // bad practice
  var page_id = url.pop();
  var category = url.pop();
  url = url[0]; // bad practice
  return {
    id: page_id,
    category: category,
    url: url
  };
}
parse('example.com/article/12');

// using unique names
function parse(url) {
  const url_parts = url.split('/');
  const page_id = url_parts.pop();
  const category = url_parts.pop();
  const domain = url_parts[0];
  return {
    id: page_id,
    category: category,
    url: domain
  };
}
parse('example.com/article/12');
```

6) Assignment in Conditional Statements:[2] JavaScript has three kinds of operators that use the = character.

- "=" For assignment.

```
  var pi = 3.14;
```

- "==" For comparing values.

```
  if (username == "admin") {}
```

- "===" For comparing both values and types.

```
  if (input === 5) {}
```

The operator == compares only values and allows different variable types to be equal if their value is the same. On the other hand, the operator === compares both the types and the values of variables and evaluates to false if operands' types are different even if their values are equal.

```
'5' == 5  // true
'5' === 5 // false
```

The operator = not only assigns a value to a variable but also returns the value. This allows multiple assignments in a single statement:

```
var a, b, c;
a = b = c = 5;
```

Which translates into:

```
var a, b, c;
(a = (b = (c = 5)));
```

The = operator also could be used in conditions:

```
function getElement(arr, i) {
  if (i < arr.length) return arr[i];
  return false;
}
var element;
if (element = getElement(arr, 5)){
  console.log(element);
}
```

[2]http://eslint.org/docs/rules/no-cond-assign

Sometimes developers use assignments in conditional statements to write less code. It could also happen by mistyping = instead of ==. IDEs[3] often flag the usage of assignment in conditions with a warning sign. Compilers like `g++` will warn about these patterns if `-Wall` switch is passed to it. It is a common pattern for iterating over an array or any other iterable object and extracting values from them, such as iterating over the result of executing a regular expression on a string. Below is an example of Assignment in Conditions code smell and a suggested refactoring.

```
var str = ’this is a string’;
var rx  = /\w+/g;
var word;
while(word = rx.exec(str)){
    console.log(word[0]); // matched word
    console.log(word.index); // matched index
}

// better approach
var str = ’this is a string’;
var rx  = /\w+/g;
var word;
while(true){
  word = rx.exec(str);
  if (!word) break;
    console.log(word[0]); // matched word
    console.log(word.index); // matched index
}
```

While assignment in conditions could be intentional, it is often the result of a mistake, *i.e.,* = is used instead of == [21].

7) Complex code: The cyclomatic complexity of a code is the number of linearly independent paths through the code [22]. JavaScript files with the Complex code smell are characterized by high cyclomatic complexity values.

8) Extra Bind:[4] The "`this`" keyword in JavaScript functions is contextual and is going to be initialized with the context which the function is being called within.

```
var obj = {
  a: 5,
  f: function () {
    return this.a;
  }
}
obj.f(); // ’this’ in f is ’obj’
```

This design of JavaScript leads to `this` to be bound to a global scope whenever the function is called as a callback if not bound explicitly. So the scope of variable `this` is not lexical. In other words `this` in inner functions is not going to be bound to the `this` of the outer function [3]. Using "`.bind(ctx)`" on a function will change the context of the function and should be used with caution.

The example below shows the usage of `.bind(ctx)` to explicitly bind the context of the callback function to the context of its outer function.

```
function downloader(id) {
  this.path = ’/’ + id;
  this.result = null;
  function callback(data) {
    this.result = data;
    console.log(’done’, this.path);
  }
  download(this.path, callback.bind(this)); // note the usage of ‘this‘
}
```

Sometimes the `this` variable is removed from the body of the inner function in the course of maintenance or refactoring. Keeping `.bind()` in these cases is an unnecessary overhead. In ES6, there is another type of functions called *arrow functions* which solved the problem mentioned above. In *arrow functions* the scoping of `this` is lexical.

The example below shows how *arrow functions* could be used to have lexical `this` inside functions.

```
function downloader(id) {
  this.path = ’/’ + id;
  this.result = null;
  download(this.path, (data) => {
    this.result = data;
    console.log(’done’, this.path);
  });
}
```

9) This Assign:[5] If the context in a callback function is not bound at the definition level, it will be lost. When there are large numbers of inner functions or callbacks in which the context should be preserved, developers often use a hacky solution such as storing `this` in another variable to access to the parent scope's context. If the context of the parent scope is stored in another variable besides `this`, usually named `self` or `that` [23], it would not be overridden and it is going to be bound to the same variable for all the defined functions in the same scope tree.

The example below is an example of storing `this` in another variable to be used in callback functions.

```
function User(id) {
  var self = this;
  self.id = id;
  getPropertiesById(id, function(props) {
    // self is bound to its value on parent scope
    // since there is no self in the current scope
    self.props = props;
  });
}
```

Assigning `this` to other variables could work for small classes, but it decreases the maintainability of code as the size of the project grows. Having a substitute variable for `this` could also break if the substitute variable is overridden by a callback function. It is a bad practice to use this hacky solution since there are other built-in language features to have lexical `this`.

The code below shows how to use built-in language features to achieve lexical `this` in callback functions.

```
function User(id) {
  this.id = id;
  getPropertiesById(id, function(props) {
    this.props = props;
  }.bind(this)); // note the .bind
}

// ES6 feature:
function User(id) {
  this.id = id;
  // arrow functions use lexical ‘this‘
  getPropertiesById(id, props => {
    this.props = props;
  });
}
```

10) Long Methods: Long method is a well-known code smell [3], [15], [24]. Long methods should be broken down into several smaller methods that do more specific tasks.

11) Complex Switch Case: Complex switch-case structures are considered a bad practice and could be a sign of violation of the Open/Close principle [25]. Switch statements also induce code duplication. Often there are similar switch statements through the software code and if the developer needs to add/remove a case to one of them, it has to go through all the statements, modifying them as well [3], [26], [27].

[3]Integrated Development Environment

[4]http://eslint.org/docs/rules/no-extra-bind

[5]https://github.com/amir-s/eslint-plugin-smells

12) Depth:[6] The depth or the level of indentation is the number of nested blocks of code. Higher depth means more nested blocks and more complexity. The following statements are considered as an increment to the number of blocks if nested: `function`, `If`, `Switch`, `Try`, `Do While`, `While`, `With`, `For`, `For in` and `For of`.

These two functions have the same functionality. But the depth of the second implementation is less than the first one.

```
// max depth = 4
function get(array, cb) {
    var result = [];
    for (var i=0;i<array.length;i++) {
        download(array[i], function (data) {
            result.push(data);
            if (result.length == array.length) {
                cb(result);
            }
        })
    }
}

// max depth = 2
function get(array, cb) {
    var result = [];
    function inner_cb(data) {
        result.push(data);
        if (result.length != array.length) return;
        cb(result);
    }
    for (var i=0;i<array.length;i++) {
        download(array[i], inner_cb)
    }
}
```

III. Study Design

The *goal* of our study is to investigate the relation between the occurrence of code smells in JavaScript files and files fault-proneness or vulnerability, as well as the smells survival all along the projects. The *quality focus* is the source code fault-proneness or vulnerability, which, if high, can have a concrete effect on the cost of maintenance and evolution of the system. The *perspective* is that of researchers, interested in the relation between code smells and the quality of JavaScript systems. The results of this study are also of interest for developers performing maintenance and evolution activities on JavaScript systems since they need to take into account and forecast their effort, and to testers, who need to know which files should be tested in priority. Finally, the results of this study can be of interest to managers and quality assurance teams, who could use code smell detection techniques to assess the fault-proneness or vulnerability of in-house or to-be-acquired systems, to better quantify the cost-of-ownership of these systems. The *context* of this study consists of 12 types of code smells identified in fifteen JavaScript systems. In the following, we introduce our research questions, describe the studied systems, and present our data extraction approach. Furthermore, we describe our model construction and model analysis approaches.

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell? Prior works show that code smells increase the fault-proneness of Java classes [5], [6]. Since JavaScript code smells are different from the code smells investigated in these previous studies on Java systems, we are interested in examining the impact that JavaScript code smells can have on the fault-proneness of JavaScript applications.

(RQ2) Are JavaScript files with code smells equally fault-prone? During maintenance and quality assurance activities, developers are interested in identifying parts of the code that should be tested and–or refactored in priority. Hence, we are interested in identifying code smells that have the most negative impact on JavaScript systems, *i.e.,* making JavaScript applications more prone to faults.

(RQ3) Is the risk of vulnerability higher in files with code smells in comparison with those without code smell? Similarly to RQ1, we are interested in examining the impact that JavaScript code smells can have on the vulnerability of JavaScript applications.

(RQ4) Are JavaScript files with code smells equally vulnerable? Similarly to RQ2, we are interested in identifying code smells that have the most negative impact on JavaScript systems, *i.e.,* making JavaScript applications more vulnerable.

(RQ5) How do the smells survive over time? We are interested here in knowing the genealogy of the smells of project, in order to have a better idea of how long those smells survive, if they are persistent, when they are created during the process life of files, and which are the most dangerous.

A. Studied Systems

In order to address our research questions, we perform a case study with the following fifteen open source JavaScript projects. Table I summarizes the characteristics of our subject systems.

Express[7] is a minimalist web framework for Nodejs. It is one of the most popular libraries in NPM [28] and it is used in production by IBM, Uber and many other companies[8]. Its Github repository has over 5,300 commits and more than 200 contributors. It has been forked 5,900 times and starred more than 32,500 times. Express is also one of the most dependent upon libraries on NPM with over 8,800 dependents. There are more than 2,400 closed Github issues on their repository.

Bower.io[9] is a package manager for client-side libraries. It is a command line tool which was originally released as part of Twitter's open source effort[10] in 2012 [29]. Its Github repository has more than 2,600 commits from more than 210 contributors. Bower has been starred over 15,000 times on Github and has over 1,600 closed issues.

LessJs[11] is a CSS[12] pre-processor. It extends CSS and adds dynamic functionalities to it. There are more than 2,600 commits by over 200 contributors on its Github repository. LessJs's repository has more than 2,100 closed issues and it is starred more than 14,500 times and forked over 3,300 times.

Request[13] is a fully-featured library to make HTTP calls. More than 8,300 other libraries are direct dependents of Request. Over 2,100 commits by more than 270 contributors

[6]http://eslint.org/docs/rules/max-depth

[7]https://github.com/expressjs/express

[8]https://expressjs.com/en/resources/companies-using-express.html

[9]https://github.com/bower/bower

[10]https://engineering.twitter.com/opensource

[11]https://github.com/less/less.js

[12]Cascading Style Sheet

[13]https://github.com/request/request

Table I: Descriptive statistics of the studied systems.

Module	Domain	# Commits	# Contributors	# Github stars	# Releases	# Closed issues	# Forks	Project start date
Express	Web framework	5300+	209	32500+	268	2400+	5900+	Jun 26, 2009
Request	HTTP client utility	2100+	272	16000+	130	1200+	1900+	Jan 23, 2011
Less.js	CSS pre-processor	2600+	209	14500+	49	2100+	3300+	Feb 20, 2010
Bower.io	Package manager	2600+	211	15000+	101	1600+	1900+	Sep 7, 2012
Grunt	Task Runner	1400+	66	11000+	11	1000+	1500+	Sep 21, 2011
Jquery	JavaScript library	6200+	265	45500+	146	1300+	13000+	Apr 3, 2009
Vue.js	JavaScript framework	2100+	122	60500+	207	4800+	8500+	Jul 29, 2013
Ramda	JavaScript library	2400+	160	8500+	45	800+	500+	Jun 21, 2013
Leaflet	JavaScript library	6300+	503	18500+	35	3100+	3200+	Sep 22, 2010
Hexo.io	Blog framework	2300+	100	17000+	119	2100+	2500+	Sep 23, 2012
Chart.js	JavaScript charting	2300+	277	31000+	37	3000+	7900+	Mar 17, 2013
Webpack	JavaScript bundler	4300+	327	30000+	244	3300+	3700+	Mar 10, 2012
Webtorrent.io	Streaming torrent client	2000+	89	13500+	257	700+	1200+	Oct 15, 2013
Moment	JavaScript date manager	3400+	413	32000+	62	2400+	4700+	Mar 1, 2011
Riot	Component-based UI library	3000+	159	12000+	96	1600+	900+	Sep 27, 2013

have been made into its Github repository and 16,000+ users starred it. There are more than 1,200 closed issues on its Github repository.

Grunt[14] is one of the most popular JavaScript task runners. More than 1,600 other libraries on NPM are direct dependents of Grunt. Grunt is being used by many companies such as Adobe, Mozilla, Walmart and Microsoft [30]. The Github repository of Grunt is starred by more than 11,000 users. More than 60 contributors made over 1,400 commits into this project. They also managed to have more than 1,000 closed issues on their github repository. We selected these projects because they are among the most popular NPM libraries, in terms of the number of installs. They have a large size and possess a Github repository with issue tracker and wiki. They are also widely used in production.

Jquery[15] is a famous JavaScript library, created to make easier the writing of client-side scripts in the HTML of web pages. It makes also easier the way to write Ajax (asynchronous JavaScript and XML) code. More than 6,200 commits have been made into its Github repository by over 260 contributors, and 45,500+ users starred it. Plus, it is forked more than 13,000 times, and there are more than 1,300 closed issues. Jquery is likely one of the most popular and biggest project of JavaScript ones.

VueJs[16] is a performant and progressive JavaScript framework for building user interfaces. It has the big advantage (in comparison with other JavaScript frameworks) to be incrementally adoptable. Over 120 contributors made over 2,100 commits into its Github repository, and they closed more than 4,800 issues. It is forked more than 8,500 times and starred more than 60,500 times, which makes it so popular.

Ramda[17] is a functional library, which makes easier the creation of functional pipelines and functions (as sequences for example), and doesn't mutate user data. It is starred more than 8,500 times, and 160 contributors made over 2,400 commits into its Github repository.

Leaflet[18] is used for mobile-friendly interactive maps, and is designed in order to be simple, efficient, easily extended (with plugins), easy to use, and usable across desktop and mobile platforms. Its Github repository is starred by more than 18,500 users and forked by over 3,200 users. More than 500 people contribute to over 6,300 commits, and managed to have more than 3,100 closed issues on their github repository.

Hexo.io[19] is a very fast, powerful, and simple framework designed for blog's creation. It has 100 contributors, who made more than 2,300 commits, and closed over 2,100 issues. Its Github repository is forked over 2,500 times and starred over 17,000 times.

ChartJs[20] is a flexible and very simple HTML5 charting that offers to designers and developers the chance to see their data in 8 different ways, possibly scalable, customisable and animated. Its Github repository joins over 270 contributors, who closed more than 3,000 issues in over 2,300 commits. Plus, more thant 7,900 users fork it and over 31,000 users star it.

Webpack[21] is a module blunder designed for modern applications. It allows the browser to load only a few number of bundles as small as possible. Those bundles correspond to the packaged modules that the application needs. Webpack is easy to configure and to take in hand. Its Github repository has over 4,300 commits and more than 320 contributors, who closed more than 3,300 issues. It has been forked 3,700 times and starred more than 30,000 times.

Webtorrent.io[22] is a streaming torrent client especially designed for the desktop and the web browser. Almost 90 contributors made over 2,000 commits and help to solve and close more than 700 issues on its Github repository. It is starred over 13,500 times.

Moment[23] allows users to do whatever they want with dates and times in JavaScript (which means manipulate, parse, validate, display, etc.) in a very easy way. Its Github repository has more than 3,400 commits, over 400 contributors, and more than 2,400 closed issues. It is forked over 4,700 times and starred more than 32,000 times.

Riot[24] is a simple, minimalistic, and elegant component-based

[14]https://github.com/gruntjs/grunt
[15]https://github.com/jquery/jquery
[16]https://github.com/vuejs/vue
[17]https://github.com/ramda/ramda
[18]https://github.com/Leaflet/Leaflet
[19]https://github.com/hexojs/hexo
[20]https://github.com/chartjs/Chart.js
[21]https://github.com/webpack/webpack
[22]https://github.com/webtorrent/webtorrent
[23]https://github.com/moment/moment
[24]https://github.com/riot/riot

UI library that offers to users the necessary building blocks for modern client-side applications, some custom tags, and an elegant syntax and API. Almost 160 people contribute to its Github repository, made more than 3,000 commits, and close over 1,600 issues. It is starred more than 12,000 times.

B. Data Extraction

To answer our research questions, we need to mine the repositories of our fifteen selected systems to extract information about the *smelliness* of each file at commit level, identifying whether the file contains a code smell or not. In addition, we need to know for each commit, if the commit introduces a bug, fixes a bug, or introduces a vulnerability, or just modifies the file in a way that a code smell is removed or added. Figure 1 provides an overview of our approach to answer RQ1 and RQ2, Figure 2 to answer RQ3 and RQ4, and Figure 3 to answer RQ5. We describe each step in our data extraction approach below. We have implemented all the steps of our approach into a framework available on Github[25].

Snapshot Generation: Since all the five studied systems are hosted on Github, at the first step, the framework performs a `git clone` to get a copy of a system's repository locally. It then generates the list of all the commits and uses it to create snapshots of the system that would be used to perform analysis at commits level.

Identification of Fault-Inducing Changes: Our studied systems use Github as their issue tracker and we use Github APIs to get the list of all the resolved issues on the systems. We leverage the SZZ algorithm [31] to detect changes that introduced faults. We first identify fault-fixing commits using the heuristic proposed by Fischer et al. [32], which consists in using regular expressions to detect bug IDs from the studied commit messages. Next, we extract the modified files of each fault-fixing commit through the following Git command:

```
git log [commit-id] -n 1 --name-status
```

We only take modified JavaScript files into account. Given each file F in a commit C, we extract C's parent commit C'. Then, we use Git's `diff` command to extract F's deleted lines. We apply Git's `blame` command to identify commits that introduced these deleted lines, noted as the "candidate faulty changes". We eliminate the commits that only changed blank and comment lines. Then, we filter the commits that were submitted after their corresponding bugs' creation date. Considering the file F in a fault-fixing commit and its commit that introduced faults, we use again Git's `diff` command to extract F's changes between both commits, in order to retrieve the "candidate fault lines" (useful for our line grain analysis). For the next step, we use UglifyJS[26] to get an F's Abstract Syntax Tree (AST) that gives the dependencies of all F's variables, objects and functions (which means their declaration and use lines). We then match F's dependencies with the "candidate fault lines" to extend them: given an F's element (variable, object, or function), if one of its declaration or use lines is found into the "candidate fault lines", then we add these declaration and use lines to the "candidate fault lines". We finally obtain the "extended candidate fault lines" (usefull for our line grain analysis including dependencies).

Identification of Vulnerability-Inducing Changes: We use Snyk[27] as our vulnerability tracker. We passe each commit of each project through Snyk which gives us the presence and characteristics (vulnerable module, vulnerability's title, how it is introduced) of commit's vulnerabilities, if they exist or if they are listed by Snyk. Given two vulnerabilities, we consider they are the same if they have the same characteristics, which means the same title, vulnerable modules, and modules through which they were introduced. Given a vulnerability V, we only keep the first commit C that introduced for the first time V. Then, similarly to the identification of fault-inducing changes, we only take C's modified JavaScript files into account (our "candidate vulnerability changes"), and for each of those files, we take the last commit before C that modified significantly those files. Given C, let's name F one of the C's modified JavaScript file, and C' the last commit before C that modified F. The next step is to use Git's `diff` command to extract F's changes between both commit C and C', in order to retrieve the "candidate vulnerability lines" (useful for our line grain analysis), and then to use UglifyJS to keep dependencies in account and get the "extended candidate vulnerability lines" (usefull for our line grain analysis including dependencies).

AST Generation and Metric Extraction: To automatically detect code smells in the source code, we first extract the Abstract Syntax Tree from the code. AST are being used to parse a source code and generate a tree structure that can be traversed and analyzed programmatically. ASTs are widely used by researchers to analyze the structure of the source code [33]–[35]. We used ESLint[28] which is a popular and open source lint utility for JavaScript as the core of our framework. Linting tools are widely used in programming to flag the potential non-portable parts of the code by statically analyzing them. ESLint is being used in production in many companies like Facebook, Paypal, Airbnb, etc. ESLint uses espree[29] internally to parse JavaScript source codes and extracts Abstract Source Trees based on the specs[30]. ESLint itself provides an extensible environment for developers to develop their own plugins to extract custom information from the source code. We developed our own plugins and modified ESLint built-in plugins to traverse the source tree generated by ESLint to extract and store the information related to our set of code smells described in section II. Table II summarizes all the metrics our framework reports for each type of code smell.

Smells Genealogy: Thanks to our previous extraction methods, we easily get, for each Javascript file of a project, the

[25]https://github.com/DavidJohannesWall/smells_project
[26]https://github.com/mishoo/UglifyJS
[27]https://snyk.io/test
[28]http://eslint.org/
[29]https://github.com/eslint/espree
[30]https://github.com/estree/estree

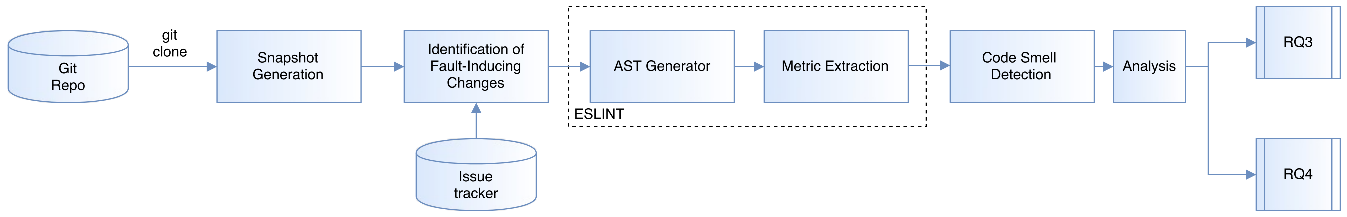


Figure 1: Overview of our approach to answer RQ1 and RQ2.

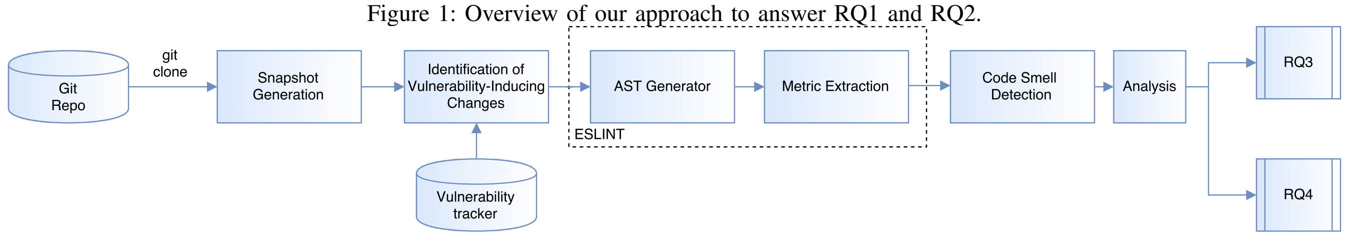


Figure 2: Overview of our approach to answer RQ3 and RQ4.

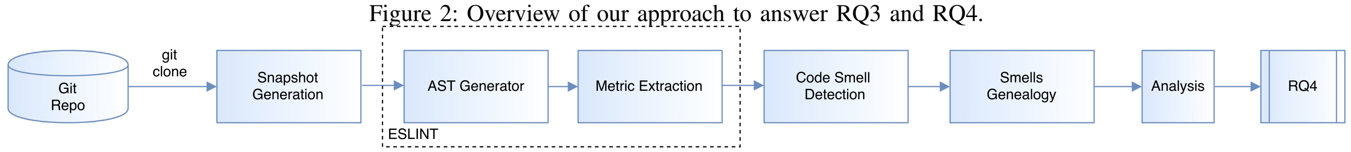


Figure 3: Overview of our approach to answer RQ5.

history of the commits that modified those files. Given the history H of a JavaScript file F, we identifiy and track F's smells through each commit of H. Given two consecutive commits $C1$ and $C2$ of H: if one smell appears in $C2$ (and not in $C1$), we consider it as a new smell and keep its date of creation ($C2$'s date); if one smell disappears in $C2$ (and was present in $C1$), we consider it was killed, and keep its date of destruction ($C2$'s date). If a smell if never killed (present in the last commit of H), we consider its presence until the last project's commit. To measure the similarity degree between two smells, they first need to be from the same smell type, and then we use SequenceMatcher[31] from difflib (a Python library) that gives us a number between 0 and 1 as a similarity degree (1: both smells are the same; 0: they are totally different). We consider two smells as the same if they are from the same smell type (among the 12 studied), and if their similarity degree is greater than 0.7. If one smell of $C1$ gets a similarity degree greater than 0.7 with two smells of $C2$, we keep the maximum in account. We tried our survival analysis of smells with different thresholds of similarity degree (0.8 and 0.9), but we observe no significant difference with the use of 0.7 threshold.

Code Smell Detection: Among of 12 metric values reported by our framework, 4 are boolean. The boolean metrics concern *This Assign*, *Extra Bind*, *Assignment in Conditional Statements*, and *Variable Re-assign* smells. The 8 remaining metrics are integers. To identify code smells using the metric values provided by our framework, we follow the same approach as previous works [36], [37], defining threshold values above which files should be considered as having the code smell. We define the thresholds relative to the systems using Box-plot analysis. We chose to define threshold values relative to the projects because design rules and programming styles can vary from one project to another, and hence it is important to compare the characteristics of files in the context of the project. For each system, we obtain the threshold values as follows. We examined the distribution of the metrics and observed a big gap around the first 70% of the data and the top 10%. Hence, we decided to consider files with metric values in the top 10% as containing the code smell. For files that contain multiple functions, we aggregated the metric values reported for each functions using the maximum to obtain a single value characterizing the file.

[31]https://docs.python.org/2/library/difflib.html

C. Analysis

To assess the impact of code smells on the fault-proneness or vulnerability of JavaScript files, or to assess the smells survival over project lifetime, we perform survival analysis, comparing the time until either a fault occurrence, or a vulnerability occurrence, in files containing code smells and files without code smells, or comparing the time until a type smell occurrence in files containing code smells, for each of the 12 type smell.

Survival analysis is used to model the time until the occurrence of a well-defined event [38]. One of the most popular models for survival analysis is the Cox Proportional Hazards (Cox) model. A Cox hazard model is able to model the instantaneous hazard of the occurrence of an event as a function of a number of independent variables [39] [40]. Particularly, Cox models aim to model how long subjects under observation can *survive* before the occurrence of an event of interest (a fault occurrence in our case) [40] [41].

Survival models were first introduced in demography and actuarial sciences [42]. Recently, researchers have started applying them to problems in the domain of Software Engineering. For example, Selim et al. [41] used the Cox model to investigate characteristics of cloned code that are related to the occurrence of faults. Koru et al. [43] also used Cox

Table II: Metrics computed for each type of code smell.

Smell Type	Type	Metric
Lengthy Lines	Number	The number of characters per line considering the exceptions described in section II.
Chained Methods	Number	The number chained methods in each chaining pattern.
Long Parameter List	Number	The number of parameters of each function in source code.
Nested Callbacks	Number	The number of nested functions present in the implementation of each function.
Variable Re-assign	Boolean	The uniqueness of variables in same scope.
Assignment in Conditional Statements	Boolean	The presence of assignment operator in conditional statements.
Complex code	Number	The cylcomatic complexity value of each function defined in the source code.
Extra Bind	Boolean	Whether a function is explicitly bound to a context while not using the context.
This Assign	Boolean	Whether `this` is assigned to another variable in a function.
Long Methods	Number	The number of statements in each function.
Complex Switch Case	Number	The number of case statements in each switch-case block in the source code.
Depth	Number	The maximum number of nested blocks in each function.

models to analyze faults in software systems. In Cox models, the hazard of a fault occurrence at a time t is modeled by the following function:

$$\lambda_i(t) = \lambda_0(t) * e^{\beta * F_i(t)} \tag{1}$$

If we take log from both sides, we obtain:

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_1 * f_{i1}(t) + ... + \beta_n * f_{in}(t) \tag{2}$$

Where:

- $F_i(t)$ is the time-dependent covariates of observation i at the time t.
- β is the coefficient of covariates in the function $F_i(t)$.
- λ_0 is the baseline hazard.
- n is the number of covariates.

When all the covariates have no effect on the hazard, the baseline hazard can be considered as the hazard of occurrence of the event (*i.e.,* a fault). The baseline hazard would be omitted when formulating the relative hazard between two files (in our case) at a specific time, as shown in the following Equation 3.

$$\lambda_i(t)/\lambda_j(t) = e^{\beta * (f_i(t) - f_j(t))} \tag{3}$$

The proportional hazard model assumes that changing each covariate has the effect of multiplying the hazard rate by a constant.

Link function. As Equation 2 shows, the log of the hazard is a linear function of the log of the baseline hazard and all the other covariates. In order to build a Cox proportional model, a linear relationship should be available between the log hazard and the covariates [44]. Link functions are used to transform the covariates to a new scale if such relationship does not exist. Determining an appropriate link function for covariates is necessary because it allows changes in the original value of a covariate to influence the log hazard equally. This allows the proportionality assumption to be valid and applicable [44].

Stratification. In addition to applying a link function, a stratification is sometimes necessary to preserve the proportionality in Cox hazard models [39]. For example, if there is a covariate that needs to be controlled because it is of no interest or secondary, stratification can be used to split the data set so that the influence of more important covariates can be monitored better [39].

Model validation. Since Cox proportional hazard models assume that all covariates are consistent over time and the effect of a covariate does not fluctuate with time, hence, to validate our model, we apply a non-proportionality test to ensure that the assumption is satisfied [44] [41].

In this paper, we perform our analysis at commit level. For each file, we use Cox proportional hazard models to calculate the risk of a fault occurrence over time, considering a number of independent covariates. We chose Cox proportional hazard model for the following reasons:
(1) In general, not all files in a commit experience a fault. Cox hazard models allow files to remain in the model for the entire observation period, even if they don't experience the event (*i.e.,* fault occurrence). (2) In Cox hazard models, subjects can be grouped according to a covariate (*e.g.,* smelly or non-smelly). (3) The characteristics of the subjects might change during the observation period (*e.g.,* size of code), and (4) Cox hazard models are adapted for events that are recurrent [44], which is important because software modules evolve over time and a file can have multiple faults during its life cycle.

IV. Case Study Results

In this section, we report and discuss the results for each research question. For each research question, we collect information about smell, fault hazard, and vulnerability codes of JavaScript files of the studied systems, and more specifically those which end with the *.js* extension. Also, we don't take in account the JavaScript files with *.min.js* extension, because they are a minified version of *.js* files that we already keep in our study. In this way, we avoid redundancy in our analyzes.

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell?

Approach. We use our framework described in Section III-B (Figure 1) to collect information about the occurrence of the 12 studied code smells in our fifteen subject systems. For each file and for each revision r (*i.e.,* corresponding to a commit), we also compute the following metrics:

- **Time:** the number of hours between the previous revision of the file and the revision r. We set the time of the first revision to zero.
- **Smelly:** this is our covariate of interest. It takes the value 1 if the revision r of the file contains a code smell and 0 if it doesn't contain any of the 12 studied code smells.

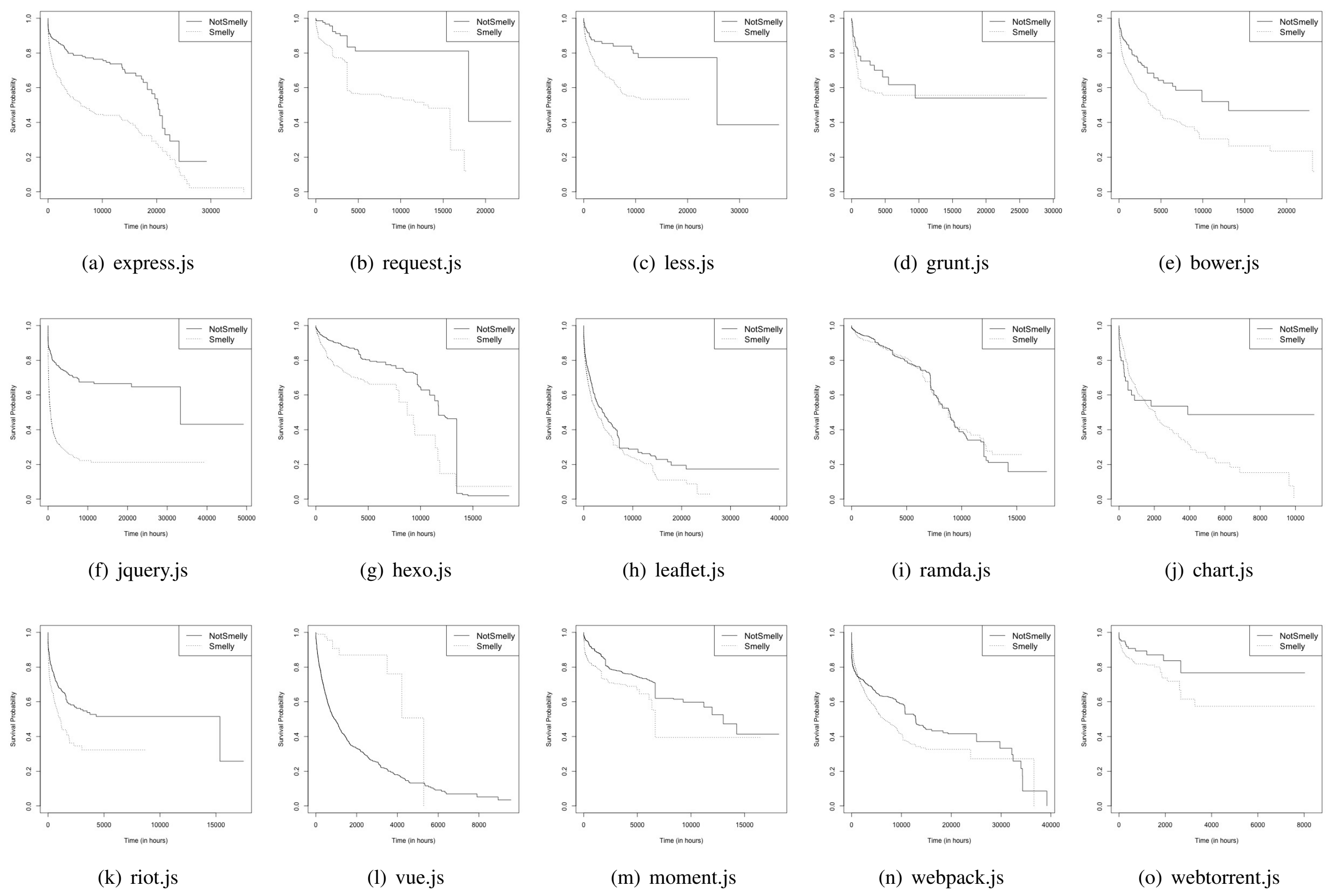


Figure 4: Survival probability trends of smelly codes vs. non-smelly codes in our fifteen JavaScript projects with the file grain approach (hazard study).

- **Event:** for the file grain approach, this metric takes the value 1 if the revision r is a fault-fixing change and 0 otherwise. We use the SZZ algorithm to insure that the file contained a code smell when the fault was introduced. For the line grain and line grain including dependencies approaches, this metric takes the value 1 if the revision r is a fault-fixing change and if there is at least one match between the fault lines and the smell lines, and 0 otherwise. Indeed, if there is no matching, we consider that the fault-fixing change doesn't fix any code smells.

Using the smelly metric, we divide our dataset in two groups: one group containing files with code smells (*i.e.,* smelly = 1) and another group containing files without any of the 12 studied code smells (*i.e.,* smelly = 0). For each group we create an individual Cox hazard model. In each group, the covariate of interest (*i.e.,* smelly) is a constant function (with value either 1 or 0), hence, there is no need for a link function to establish a linear relationship between this covariate and our event of interest, *i.e.,* the occurrence of a fault. We use the *survfit* and *coxph* functions from R [45] to analyze our Cox hazard models.

In addition to building Cox hazard models, we test the following null hypothesis: H_0^1: *There is no difference between the probability of a fault occurrence in a file containing code smells and a file without code smells*. We use the *log-rank* test (which compares the survival distributions of two samples), to accept or refute this null hypothesis.

Findings. File grain results presented in Figure 4 show that files containing code smells experience faults faster than files without code smells. Table III (line grain results) and Figure 5 (line grain including dependencies results) show the same minimized but still acceptable observation, wich means files containing code smells experience faults faster than files without code smells. The Y-axis in Figure 4 and Figure 5 represents the probability of a file *surviving* a fault occurrence. Hence a low value on the Y-axis means a low *survival* rate (*i.e.,* a high hazard or high risk of fault occurrence). For all fifteen projects, and for each approach (file grain, line grain, and line grain including dependencies), we calculated relative hazard rates (using Equation 3 from Section III-C) between files containing code smells and files without code smells. Results show that, on average, files without code smells have hazard rates 76% lower than files with code smells in our file grain analysis, 20% lower in our line grain analysis, and

Table III: Fault hazard ratios for each project with the line grain approach. $exp(coef)$ values means higher hazard rates.

module	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	1.341	0.002	0.192
request	2.538	0.028e-2	0.602
less	1.791	0.003	0.419
bower	1.321	0.027	0.982
grunt	0.594	0.005	0.019
jquery	3.436	0	1.197e-8
vue	0.062	0.011e-9	0.843
ramda	0.460	0.01e-8	3.691e-5
leaflet	0.725	0.088e-4	0.739
hexo	1.199	0.077	0.945
chart	0.711	0.136	1.46e-9
webpack	0.603	0	0
webtorrent	1.222	0.047e-9	0.045
moment	0.941	0.401	5.046e-5
riot	1.047	0.586	0.797

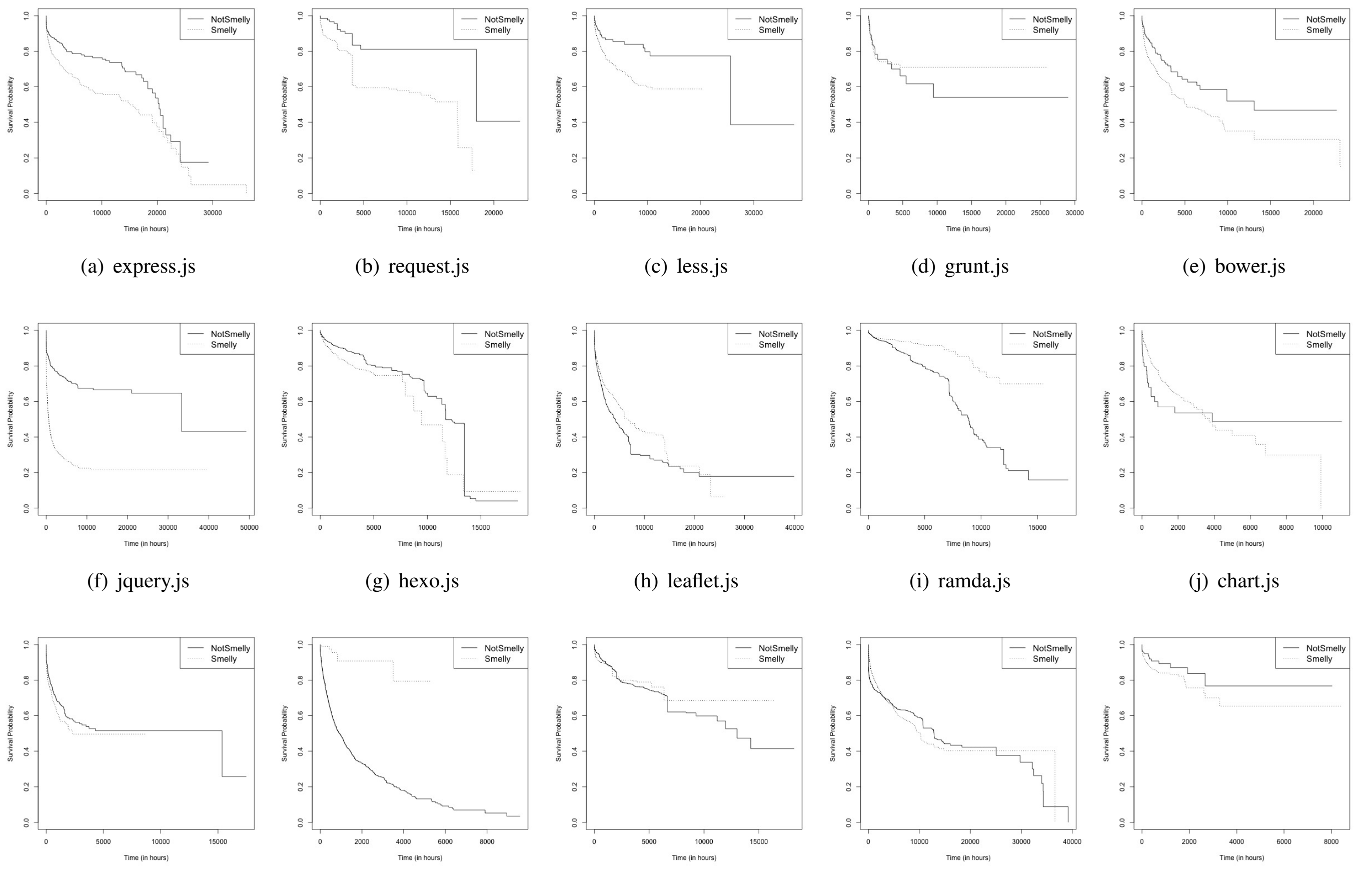
38% lower in our line grain analysis including dependencies. It is normal to see this pourcentage decreasing in line grain approach, because we add an additional matching condition to set the *event* to 1. Between the line grain and the line grain including dependencies analyzes, this pourcentage increases due to the increasing of the fault lines number considering during the compute of *event*. We performed a *log-rank* test comparing the survival distributions of files containing code smells and files without any of the studied code smells and obtained p-values lower than 0.05 for most of the fifteen studied systems. Hence, we reject H_0^1. Since our detection of code smells depends on our selected threshold value (*i.e.,* the top 10% value chosen in Section III-B), we conducted a sensitivity analysis to assess the potential impact of this threshold selection on our result. More specifically, we rerun all our analysis with threshold values at top 20% and top 30%. We observed no significant differences in the results. Hence, we conclude that:

JavaScript files without code smells have hazard rates 76% lower than JavaScript files with code smells in the file grain approach, and this difference is statistically significant. Plus, this difference still remains significant in the line grain including dependencies approach, because hazard rates reach 38%.

(RQ2) Are JavaScript files with code smells equally fault-prone?

Approach. Similar to **RQ1**, we use our framework from Section III-B ((Figure 1)) to collect information about the occurrence of the 12 studied code smells in our fifteen subject systems. For each file and for each revision r (*i.e.,* corresponding to a commit), we also compute the **Time** and **Event** metrics defined in **RQ1**. For each type of code smell i we define the metric **Smelly**$_i$**:** which takes the value 1 if the revision r of the file contains the code smell i and 0 if it doesn't contain any of the 12 studied code smells. Also, for the line grain and the line grain including dependencies approaches, we define the metric **Event**$_i$**:** which takes the value 1 if the revision r is a fault-fixing change and if the code smell i is in the intersection between the fault lines and the smell lines, and 0 otherwise. When computing the **Event** and **Event**$_i$ metrics, we used the SZZ algorithm to ensure that the file contained the code smell i when the fault was introduced. Because size, code churn, and the number of past occurrence of faults are known to be related to fault-proneness, we add the following metrics to our models, to control for the effect of these covariates : (i) LOC: the number of lines of code in the file at revision r; (ii) Code Churn: the sum of added, removed and modified lines in the file prior to revision r; (iii) No. of Previous-Bugs: the number of fault-fixing changes experienced by the file prior to revision r. We perform a stratification considering the covariates mentioned above, in order to monitor their effect on our event of interest, *i.e.,* a fault occurrence. Next, we create a Cox hazard model for each of our fifteen studied systems. In order to build an appropriate link function for the new covariates considered in this research question (*i.e.,* LOC, Code churn, and No. of Previous-Bugs), we follow the same methodology as [39] [41] and plot the log relative risk vs. each type of code smell, the No. of Previous-Bugs, LOC and Code Churn in each of our fifteen datasets (corresponding to the fifteen subject systems). For all types of code smells and No. of Previous-Bugs covariates, we observed that a linear relationship exists. Since the plots for LOC and Code Churn covariates were similar to each other, for all of the fifteen systems, and because of space limitation, in this paper, we present only the plot of LOC (obtained on *express.js*) and Code Churn (obtained on *grunt.js*) covariates (see Figure 6). Figure 6 shows that for LOC, we do not have a linear relationship, hence we visually identified a suitable function (*i.e.,* a logarithmic function) to establish a linear relationship. In the case of Code Churn, we identified that a negative linear function should be applied. We generated summaries of all our Cox models and removed insignificant covariates, *i.e.,* those with p-values greater than 0.05. Finally, for each system, we performed a non-proportional test to verify if the proportional hazards assumption holds.

Findings. Tables IV, V and VI summarizes the fault hazard ratios for the 12 studied code smells for respectively the file grain, line grain, and line grain including dependencies approach. The value in the column exp(coef) shows the amount of increase in hazard rate that one should expect for each



(a) express.js (b) request.js (c) less.js (d) grunt.js (e) bower.js

(f) jquery.js (g) hexo.js (h) leaflet.js (i) ramda.js (j) chart.js

(k) riot.js (l) vue.js (m) moment.js (n) webpack.js (o) webtorrent.js

Figure 5: Survival probability trends of smelly codes vs. non-smelly codes in our fifteen JavaScript projects with the line grain including dependencies approach.

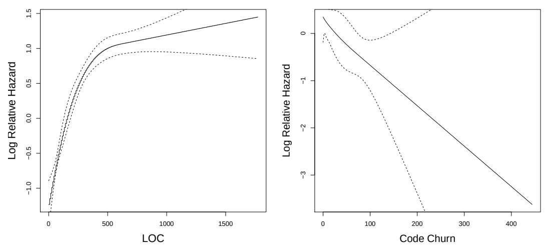


Figure 6: Determining a link function for express.js (left figure) and grunt.js (right figure) modules for two covariates: LOC and Code Churn respectively.

unit increase in the value of the corresponding covariate. The last column of Tables IV, V and VI show that the p-values obtained for the non-proportionality tests are above 0.05 for all the fifteen systems; meaning that the proportional hazards assumption is satisfied for all the fifteen studied systems. Actually, we removed from the tables the insignificant covariates, which means those with non-proportionality test p-values less than 0.05, and we will only consider the covariates with *exp(coef)* greater than 1 (for those the corresponding files are more fault-proneness when they are smelly).

Overall, the hazard ratios of the studied code smells vary across the systems and accross the approaches (file grain, line grain and line grain including dependencies). With the file grain approach, *Variable Re-assign* has one of the highest hazard ratio in six out of fifteen systems (40%); *This Assign*, and *Complex Switch Case* have one of the highest hazard rate in four out of fifteen systems (27%); *Nested Callbacks*, *Assignment in Conditional Statements*, *Complex Code*, and *Long Methods* are the most hazard code smell in three out of fifteen systems (20%); and the other smells are the most hazard code smell in at least one system. With our line grain approach, the results are a little different. *Variable Re-assign* still has one of the highest hazard ratio in most systems, that is to say in four out of fifteen systems (27%); *Assignment in Conditional Statements* has one of the highest hazard rate in two out of fifteen systems (13%); *Lenghty Lines*, and *Complex Code* are the most hazard code smell in only one out of fifteen systems (7%); the other smells don't appear in any of the studied systems as having a high hazard ratio. With the last approach (line grain including dependencies), *Variable Re-assign* still is one of the most hazard code smell in most systems, in seven out of fifteen systems (47%); *Complex Code* has one

Table IV: Hazard ratios for each type of code smells with file grain approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	No.Previous-Bugs	1.031	0	0.076
	Long Methods	3.363	0.012e-11	0.14
	Complex Switch Case	2.528	0.043e-8	0.322
	Variable Re-assign	1.927	0.056e-14	0.725
grunt	No.Previous-Bugs	1.051	0.003	0.925
	Nested Callbacks	3.609	0.086e-3	0.65
	Assign. in Cond. State.	2.008	0.004	0.707
	Long Methods	1.8	0.019	0.098
bower	No.Previous-Bugs	1.047	0	0.596
	LOC	1.001	0.023e-13	0.379
	Complex Code	5.778	0	0.432
	Long Methods	3.55	0	0.525
	Extra Bind	3.03	0.097e-5	0.378
less	No.Previous-Bugs	1.024	0	0.313
	Depth	2.36	0.013e-2	0.95
	Assign. in Cond. State.	2.25	0.012e-10	0.169
	Lengthy Lines	2.07	0.027e-4	0.709
request	No.Previous-Bugs	1.053	0.097e-13	0.23
	LOC	1.001	0.022e-14	0.718
	This Assign	3.094	0.027e-12	0.4
	Variable Re-assign	2.614	0.063e-4	0.78
	Complex Switch Case	2.608	0.002	0.632
jquery	LOC	1.0001	0	0.164
	Lengthy Lines	2.025	0	0.393
	Assign. in Cond. State.	1.881	0.014e-6	0.123
	Complex Code	1.706	0	0.169
hexo	No.Previous-Bugs	1.255	0	0.355
	LOC	1.001	0	0.236
	Chaine Methods	2.511	0.044e-8	0.805
	Variable Re-assign	1.916	0.027e-11	0.523
	Nested Callbacks	1.915	0.02e-3	0.942
leaflet	LOC	1.0001	0.016e-10	0.14
	Nested Callbacks	1.494	0.009	0.373
	Variable Re-assign	1.24	0.082e-2	0.321
ramda	LOC	1.0001	0.033	0.122
	Complex Switch Case	3.64	0.034e-7	0.952
	Long Parameter List	3.487	0.011e-7	0.616
	Complex Code	2.859	0.076e-5	0.291
riot	Variable Re-assign	1.782	0.024e-12	0.09
	This Assign	1.445	0.002	0.458
vue	Assign. in Cond. State.	0.161	0.01	0.856
	Variable Re-assign	0.092	0.018e-19	0.149
moment	No.Previous-Bugs	1.014	0	0.326
	Complex Switch Case	9.495	0	0.914
	Chained Methods	4.148	0	0.119
	This Assign	2.925	0	0.327
webtorrent	No.Previous-Bugs	1.063	0.014e-9	0.053
	LOC	1.001	0.014e-6	0.053
	This Assign	1.967	0.019e-2	0.28
	Variable Re-assign	1.883	0.01	0.399

Table V: Hazard ratios for each type of code smells with line grain approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	No.Previous-Bugs	1.034	0	0.126
	Variable Re-assign	1.22	0.024	0.814
grunt	No.Previous-Bugs	1.064	0.009	0.693
	Assign. in Cond. State.	0.315	0.047	0.628
	Chained Methods	0.206	0.002	0.94
	Lengthy Lines	0.154	0.036e-3	0.053
bower	No.Previous-Bugs	1.051	0	0.679
	LOC	1.001	0.064e-11	0.480
	Variable Re-assign	1.335	0.02	0.832
	This Assign	0.394	0.012e-5	0.998
less	No.Previous-Bugs	1.024	0	0.566
	Variable Re-assign	1.446	0.033	0.652
	Assign. in Cond. State.	1.342	0.036	0.052
request	No.Previous-Bugs	1.061	0.013e-13	0.185
	LOC	1.001	0	0.517
	Variable Re-assign	1.913	0.003	0.455
jquery	LOC	1.0001	0	0.164
	Lengthy Lines	2.002	0	0.385
	Assign. in Cond. State.	1.881	0.014e-6	0.123
	Complex Code	1.684	0	0.164
hexo	No.Previous-Bugs	1.25	0	0.296
	LOC	1.001	0.015e-8	0.251
leaflet	No.Previous-Bugs	1.016	0	0.08
	LOC	1.0001	0.01e-2	0.192
	Variable Re-assign	0.712	0.025e-4	0.578
	Complex Code	0.485	0.003	0.179
	Chained Methods	0.405	0.034e-4	0.201
ramda	LOC	1.0001	0.037	0.225
	Nested Callbacks	0.399	0.014e-2	0.271
	Complex Code	0.242	0.046	0.322
	Chained Methods	0.231	0.038	0.996
chart	No.Previous-Bugs	1.151	0	0.113
	Nested Callbacks	0.321	0.034e-7	0.747
	This Assign	0.265	0	0.198
	Long Parameter List	0.191	0.036e-4	0.306
riot	This Assign	0.125	0.047e-6	0.8
	Long Parameter List	0.105	0.069e-4	0.277
vue	Variable Re-assign	0.069	0.065e-9	0.678
	This Assign	0.017	0.049e-3	0.156
moment	No.Previous-Bugs	1.015	0	0.496
	Long Methods	0.261	0.003	0.425
webpack	Extra Bind	0.295	0.035	0.969
webtorrent	No.Previous-Bugs	1.054	0.075e-5	0.058
	LOC	1.001	0.018e-2	0.095
	Nested Callbacks	0.17	0.013	0.405

of the highest hazard rate in three out of fifteen systems (20%); *Lenghty Lines*, *Assignment in Conditional Statements*, and *Long Methods* are the most hazard code smells in only one out of fifteen systems (7%); the other smells don't have an enough high hazard rate in any of the studied systems. Furthemore, the most hazard types of code smell seem not to vary accross the approaches, and this observation particularly affects *Variable Re-assign*, *Assignment in Conditional Statements*, and *Complex Code* smells.

As we expected, in our three approaches, the covariates No.Previous-Bugs is significantly related to fault occurrence, because it appears in at least eight out of fifteen systems with an *exp(coef)* greater than 1 and good p-values. However, its hazard rate is lower than those of many of the studied code smells. LOC is significantly related to fault occurrence in seven systems in our three approaches (less than half of the studied systems) with a very low hazard rates, meaning that JavaScript developers cannot simply control for size and monitor files with previous fault occurrences, if they want to track fault-prone files effectively. Since *Variable Re-assign*, *Assignment in Conditional Statements*, and *Complex Code* are related to high hazard ratios in respectively 38%, 13% and 16% of the cases (which means fifteen studied systems and three approches, that is to say 45 cases), we strongly recommend that developers prioritize files containing these three types of code smells during testing and maintenance activities.

Table VI: Hazard ratios for each type of code smells with line grain including dependencies approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	No.Previous-Bugs	1.034	0	0.124
	Complex Code	1.955	0.03e-2	0.152
	Long Methods	1.674	0.024	0.227
	Variable Re-assign	1.354	0.042e-2	0.742
grunt	No.Previous-Bugs	1.072	0.035e-2	0.495
	Lengthy Lines	0.402	0.001	0.288
bower	No.Previous-Bugs	1.05	0	0.855
	LOC	1.001	0.07e-11	0.652
	Complex Code	2.314	0.006	0.734
	Variable Re-assign	1.579	0.019e-2	0.601
less	No.Previous-Bugs	1.023	0	0.522
	Variable Re-assign	1.616	0.005	0.463
request	No.Previous-Bugs	1.056	0.038e-13	0.188
	LOC	1.001	0.022e-14	0.664
	Variable Re-assign	2.316	0.089e-3	0.66
jquery	LOC	1.0001	0	0.161
	Lengthy Lines	2.002	0	0.385
	Assign. in Cond. State.	1.881	0.014e-6	0.123
	Complex Code	1.684	0	0.164
hexo	No.Previous-Bugs	1.254	0	0.325
	LOC	1.001	0.019e-11	0.269
	Variable Re-assign	1.321	0.004	0.896
leaflet	LOC	1.0001	0.081e-10	0.101
	Variable Re-assign	0.755	0.047e-3	0.435
	Complex Code	0.485	0.003	0.179
	Chained Methods	0.434	0.091e-4	0.222
ramda	LOC	1.0001	0.037	0.228
	Nested Callbacks	0.579	0.007	0.978
	Complex Code	0.242	0.046	0.322
chart	No.Previous-Bugs	1.15	0	0.122
	Nested Callbacks	0.321	0.034e-7	0.747
	This Assign	0.281	0	0.173
	Long Parameter List	0.191	0.036e-4	0.306
riot	Variable Re-assign	1.26	0.004	0.258
	Chained Methods	0.22	0.067e-3	0.602
	This Assign	0.125	0.046e-6	0.93
vue	Variable Re-assign	0.092	0.018e-9	0.149
	Depth	0.033	0.066e-2	0.187
moment	No.Previous-Bugs	1.015	0	0.493
	This Assign	0.384	0.003	0.565
webpack	Nested Callbacks	0.381	0.002	0.439
webtorrent	LOC	1.001	0.08e-6	0.054
	Variable Re-assign	1.654	0.043	0.647
	Nested Callbacks	0.255	0.019	0.491

JavaScript files containing different types of code smells are not equally fault-prone. Developers should consider refactoring files containing either Variable Re-assign code smell, or Assignment in Conditional Statements code smell, or Complex Code smell in priority since they seem to increase the risk of faults in the system.

Similar to **RQ1**, we conducted a sensitivity analysis to assess the potential impact of our threshold selection (performed during the detection of code smells) on the results; rerunning the analysis using threshold values at top 20% and top 30%. We did not observed any significant change in the results.

(RQ3) Is the risk of vulnerability higher in files with code smells in comparison with those without code smell?

Approach. We use here our framework described in Section III-B (Figure 2) to collect information about the occurrence of the 12 studied code smells in our fifteen subject systems, as well as the vulnerable codes and commits. For each file and for each revision r (*i.e.,* corresponding to a commit), we also compute the **Time** and **Smelly** metrics defined in **RQ1**. We also compute the **Event**, but differently to **RQ1**: for the file grain approach, this metric takes the value 1 if the revision r introduces a vulnerability v for the first time through its changes and 0 otherwise. Indeed, We only take in account the revision r in which the vulnerability v appears for the first time. We use the SZZ algorithm to insure that the file contained a code smell when the vulnerability was introduced for the first time. For the line grain and line grain including dependencies approaches, this metric takes the value 1 if the revision r introduces the vulnerability v for the first time and if there is at least one match between the vulnerable lines and the smell lines, and 0 otherwise. Actually, if there is no matching, we say that there is no link between the changes that inroduce the vulnerability for the first time and the code smells.

To collect information about vulnerabilities of revisions, we question a specific vulnerability database, Snyk[32], which gives us the vulnerabilities's characteristics of each revision and of each studied systems. However, this database is not complete, which poses two problems to keep in mind during our study:

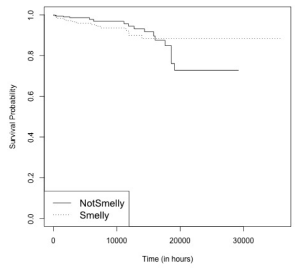
- Given a studied system, some of its revisions are not identified in the database. We often meet a situation in which three successive revisions should introduce a same vulnerability, but the intermediate one indexes no vulnerability because it is not recognized by the database.
- Some of our studied systems (more precisely seven, which means almost half) have no vulnerability. However, we are aware that this kind of system (which means large system) are unlikely to have no vulnerability. Thereby, we will focus our analyzes on the following eight systems: *bower*, *express*, *grunt*, *hexo*, *request*, *riot*, *webpack*, and *webtorrent*.

Also, this database presents another drawback, because of its low accuracy. Indeed, when a vulnerability is found by the database, it does not specify any file of the revision in which the vulnerability has been found. Thus, we can only suppose that the vulnerability affects all the files changed by the revision, which is not accurate.

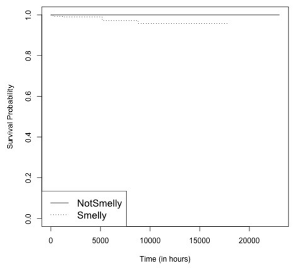
Using the smelly metric, in a similar way than **RQ1**, we divide our dataset in two groups: one group containing files with code smells (*i.e.,* smelly = 1) and another group containing files without any of the 12 studied code smells (*i.e.,* smelly = 0). For each group we create an individual Cox hazard model. In each group, the covariate of interest (*i.e.,* smelly) is a constant function (with value either 1 or 0), hence, there is no need for a link function to establish a linear relationship between this covariate and our event of interest, *i.e.,* the first appearance of a vulnerability. We use the *survfit* and *coxph* functions from R [45] to analyze our Cox hazard models.

In addition to building Cox hazard models, we test the following null hypothesis: H_0^2: *There is no difference between*

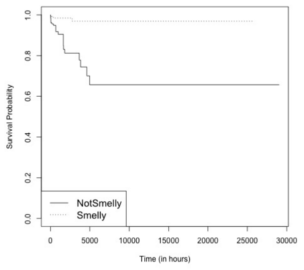
[32]https://snyk.io/test



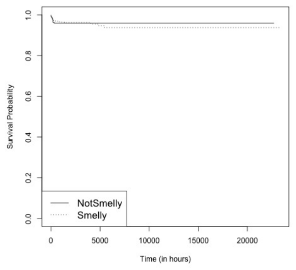
(a) express.js



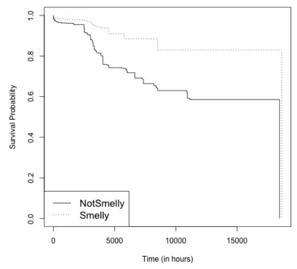
(b) request.js



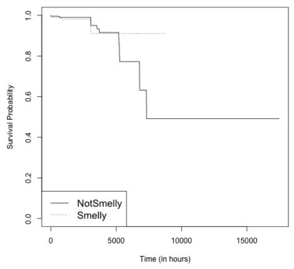
(c) grunt.js



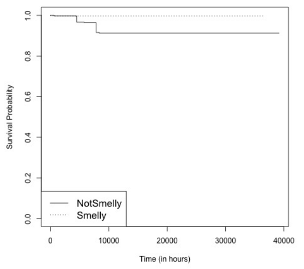
(d) bower.js



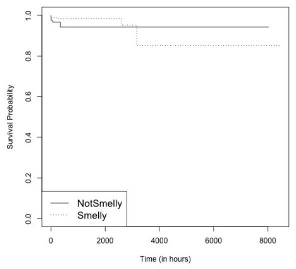
(e) hexo.js



(f) riot.js



(g) webpack.js



(h) webtorrent.js

Figure 7: Survival probability trends of smelly codes vs. non-smelly codes in our fifteen JavaScript projects with the file grain approach (vulnerability study).

Table VII: Hazard ratios (vulnerability study) for each project with the line grain approach. $exp(coef)$ values means higher hazard rates.

module	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	0.288	0.082e-3	0.113
request	0.189	0.014	0.441
bower	0.229	0.031e-7	0.021
grunt	0.043	0.023e-7	0.994
hexo	0.287	0.056e-14	0.902
webpack	0.142	0.047e-3	0.089
webtorrent	0.175	0.097e-3	0.217
riot	0.774	0.445	0.505

Table VIII: Hazard ratios (vulnerability study) for each project with the line grain including dependencies approach. $exp(coef)$ values means higher hazard rates.

module	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	0.489	0.013	0.066
request	0.831	0.818	0.667
bower	0.243	0.013e-6	0.009
grunt	0.096	0.03e-7	0.58
hexo	0.322	0.04e-12	0.896
webpack	0.176	0.09e-3	0.059
webtorrent	0.31	0.009	0.109
riot	0.774	0.445	0.505

the probability of a first vulnerability occurrence in a file containing code smells and a file without code smells. We use the *log-rank* test (which compares the survival distributions of two samples), to accept or refute this null hypothesis.

Findings. File grain results presented in Figure 7 show that files containing code smells don't necessarily experience vulnerability faster than files without code smells (only *request* and *bower* show that files containing smells have a trend to be more vulnerable than files without smells). Table VII (line grain results) and VIII (line grain including dependencies results) show the same accentuated observation, because none of the *exp(coef)* is greater than 1. The Y-axis in Figure 7 represents the probability of a file *surviving* a vulnerability occurrence. Hence a low value on the Y-axis means a low *survival* rate (*i.e.,* a high vulnerability or high risk of vulnerability occurrence). For all fifteen projects, and for each approach (file grain, line grain, and line grain including dependencies), we calculated relative fault hazard rates (using Equation 3 from Section III-C) between files containing code smells and files without code smells. Results show that, on average, files without code smells have hazard rates 34% upper than files with code smells in our file grain analysis,

Table IX: Hazard ratios (vulnerability study) for each type of code smells with file grain approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
grunt	Variable Re-assign	0.178	0.057e-5	0.169
request	LOC	1.002	0.001	0.935
	This Assign	4.707	0.03	0.52
hexo	This Assign	0.41	0.02	0.275
	Variable Re-assign	0.407	0.074e-8	0.668
riot	This Assign	2.76	0.026	0.974
webpack	LOC	1.001	0.009	0.465

Table X: Hazard ratios (vulnerability study) for each type of code smells with line grain approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
express	Variable Re-assign	0.342	0.001	0.08
grunt	Variable Re-assign	0.047	0.059e-7	0.996
bower	This Assign	0.283	0.015	0.756
request	LOC	1.002	0.001	0.935
	Variable Re-assign	0.228	0.028	0.426
hexo	This Assign	0.291	0.006	0.377
	Chained Methods	0.2	0.023	0.593
	Variable Re-assign	0.297	0.014e-12	0.868
webpack	LOC	1.001	0.009	0.465
	Variable Re-assign	0.146	0.061e-3	0.088
webtorrent	Variable Re-assign	0.24	0.002	0.183

and this pourcentage increases with the others analyzes (line grain and line grain including dependencies). Nevertheless, we need to be careful with our conclusion because of the lack of completeness and accuracy of our vulnerability database, as said previously. Hence, we can not reject H_0^2 because of our *exp(coef)* results, but we can not validate it because of the weaknesses of the vulnerability database we used.

JavaScript files with code smells are not necessarily more vulnerable than JavaScript files without code smells, and we need a better vulnerability database to confirm or refute more precisely our conclusion.

(RQ4) Are JavaScript files with code smells equally vulnerable?

Approach. Similar to **RQ3**, we use our framework from Section III-B (Figure 2) to collect information about the occurrence of the 12 studied code smells, as well as the vulnerable codes and commits, in our fifteen studied systems. For each file and for each revision r (*i.e.,* corresponding to a commit), we also compute the **Time** and **Event** metrics defined in **RQ3**. For each type of code smell i we define the metric **Smelly$_i$:** which takes the value 1 if the revision r of the file contains the code smell i and 0 if it doesn't contain any of the 12 studied code smells. Also, for the line grain and the line grain including dependencies approaches, we define the metric **Event$_i$:** which takes the value 1 if the revision r introduces a vulnerability v for the first time through its changes and if the code smell i is in the intersection between the vulnerability lines and the smell lines, and 0 otherwise. When computing the **Event** and **Event$_i$** metrics, we used the SZZ algorithm to ensure that the file contained the code smell i when the vulnerability was introduced. Because size and code churn could be related to vulnerability, we add the following metrics to our models, to control for the effect of these covariates : (i) LOC: the number of lines of code in the file at revision r; (ii) Code Churn: the sum of added, removed and modified lines in the file prior to revision r.

We perform a stratification considering the covariates mentioned above, in order to monitor their effect on our event of interest, *i.e.,* a vulnerability occurrence. Next, we create a Cox hazard model for each of our fifteen studied systems. We then generated summaries of all our Cox hazard models and removed insignificant covariates, *i.e.,* those with p-values greater than 0.05. Finally, for each system, we performed a non-proportional test to verify if the proportional hazards assumption holds.

Table XI: Hazard ratios (vulnerability study) for each type of code smells with line grain including dependencies approach. Higher $exp(coef)$ values means higher hazard rates.

module	covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
grunt	Variable Re-assign	0.088	0.057e-7	0.581
bower	This Assign	0.283	0.015	0.756
request	LOC	1.002	0.001	0.935
hexo	This Assign	0.291	0.006	0.377
	Variable Re-assign	0.327	0.035e-11	0.972
webpack	LOC	1.001	0.009	0.465
	Variable Re-assign	0.181	0.012e-2	0.058

Findings. Tables IX, X and XI summarize the vulnerability hazard ratios for the 12 studied code smells for respectively the file grain, line grain, and line grain including dependencies approach. The value in the column *exp(coef)* shows the amount of increase in vulnerability hazard rate that one should expect for each unit increase in the value of the corresponding covariate. The last column of Tables IX, X and XI show that the p-values obtained for the non-proportionality tests are above 0.05 for all the fifteen systems; meaning that the proportional hazards assumption is satisfied for all the fifteen studied systems. Actually, we removed from the tables the insignificant covariates, which means those with non-proportionality test p-values less than 0.05.

Overall, the vulnerability hazard ratios of the studied code smells vary across the systems and accross the approaches (file grain, line grain and line grain including dependencies), and almost all of them have an *exp(coef)* less than 1. It confirms our last conclusion in **RQ3**, that is to say that JavaScript files with code smells are not more vulnerable than those without code smells (and we still have to mitigate this conclusion because of our vulnerability database). As said in **RQ3**, we collected information only on eight systems (out of fifteen) because of the lack of completeness of the vulnerability database. Plus, because of our p-values restriction, only a few covariates are reported in our tables, and some of the eight studied systems don't appear. However, it is interesting to notice that with the file grain approach, *This Assign* has the highest vulnerability hazard ratio in three out of eight systems (37.5%), and is the only covariate with a hazard ratio greater than 1 in two systems (*request* and *riot*); *Variable Re-assign* has one of the highest hazard rate in two out of eight systems (25%); *This Assign* and *Variable Re-assign* are the only smell covariates reported. With our line grain approach, *Variable Re-assign* has still one of the highest hazard ratio in most systems, that is to say in six out of eight systems (75%); *This Assign* has one of the highest hazard rate in two out of eight systems (25%); *Chained Methods* is the most hazard code smell in only one out of eight systems (12.5%); the other smells don't appear in any of the studied systems because of the p-values limitations. With the line grain including dependencies approach, *Variable Re-assign* is still one of the most hazard code smell in most systems, in three out of fifteen systems (37.5%); *This Assign* has one of the highest hazard rate in two out of eight systems (25%); and only *Variable Re-assign* and *This Assign* are reported. Furthemore, the most vulnerability hazard types of code smell seem not to vary accross the approaches, and this observation particularly affects *Variable Re-assign* and *This Assign* code smells.

In our three approaches, the covariate LOC is significantly related to fault occurrence in two systems (*request* and *webpack*) with a very low hazard rate, meaning that JavaScript developers cannot simply control for size if they want to track vulnerable files effectively. Since *Variable Re-assign* and *This Assign* have the highest hazard ratios in respectively 46% and 29% of the cases (which means eight studied systems and three approches, that is to say 24 cases), we can recommend that developers prioritize files containing these two types of code smells in order to make their system less vulnerable. *Variable Re-assign* seems, as in **RQ2**, an unavoidable type smell that developers have to strongly consider when they maintain, test, and fix their system for reducing fault-proneness and potentially vulnerability.

JavaScript files containing different types of code smells are not equally vulnerable. Developers should consider refactoring files containing Variable Re-assign *code smell or* This Assign *code smell in priority since they seem to increase the risk of vulnerability in the system. Finally,* Variable Re-assign *code smell is essential to consider for reducing the risk of vulnerability and fault in the system.*

(RQ5) How do the smells survive over time?

Approach. We use now the framework described in Section III-B, Figure 3, to collect information about the appearance of the 12 studied code smells in our fifteen subject systems, as well as their line localization, their content and their genealogy (which means their evolution over time from their creation to either their destruction, or the last revision of the studied system). For each studied system and for each smell type, we compute the following metrics:

- The number of created smells.
- The number of killed smells (over the system lifetime).
- The number of survived smells, which means the number of smell that presently appear in the system.
- The number of smells created at the file birthdate.
- The median days of survival of the smells.
- The average days of survival of the smells.

For each smell created (which means never encountered before), we also compute the **Time** and **Event** metrics thus defined:

- **Time:** the time in days since the smell creation.
- **Event:** this metric takes the value 1 if the studied smell is present at this time (which means not killed), and 0 otherwise.

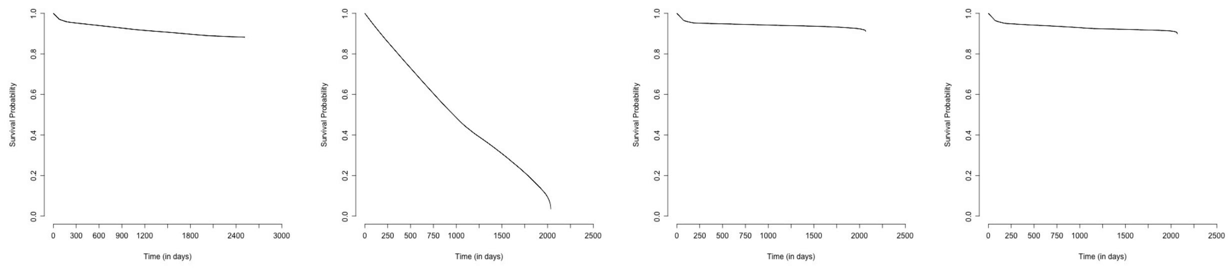
In this way, if a particular smell s is killed x days after its introduction, we will have the corresponding event metric equal to 1 from 0 to $x-1$, and equal to 0 at the time x and after. When we report those information for a studied system, the maximum time that we take in account, for a particular smell type, corresponds to the maximum lifetime of the smells of this type. Thereby, for each smell type of the system and for each time, we will know the proportion of smells alive relatively to the number of smells created. This will particularly help us in the Cox survival model design.

Then, for each of the twelve studied smells, and for each of the fifteen studied systems, we create an individual Cox survival model using the **Time** and **Event** metrics previously defined. We use the *survfit* and *coxph* functions from R [45] to analyze our Cox survival models.

Findings. Our results are presented in the Table XII for a density analysis, and in the Figures 8 to 22 for a survival analysis. For the Table XII, for each system and each smell, the third column corresponds to the number of killed smells, and the fourth to the number of survived smells. The sum of both columns gives us the number of created smells. The fifth column reports, in pourcentage, the proportion of smells created at the files birthdate, relatively to the number of created smells. In order to not overload the presentation of our results, we only report the descriptive statistics for the four most relevant smells, which means those for wich the number created is the most considerable. Finally, the Table reports, for each studied system, general statistics (*SUM* lines), computed by summing the statistics of the twelve studied smells. For the Figures 8 to 22, we plot the survival analysis for each studied systems, and for each smells reported in the Table XII, still in order to not overload the presentation of our results. The Y-axis corresponds to the chance of surviving of a given smell type, x days after its introduction into the codebase. The results presented in Table XII show that smells are not often introduced during files evolution and changes, but rather at the creation of files. Indeed, when we look at the *SUM* lines, from 34.5% (for *leaflet*) to 91.4% (for *less*) of the smells are introduced at the file birthdate, meaning that developers

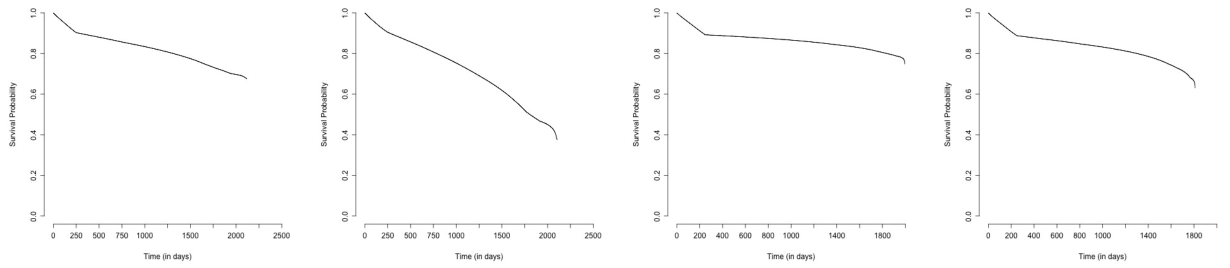
Table XII: Descriptive statistics on survival over time of the largest smells of studied systems.

System	Smell	Not Survived	Survived	Number created at file birth	Median days or survival	Average days of survival
express	Variable Re-assign	6743	425	5783 (80.7%)	74	209
	Nested Callbacks	314	728	417 (40%)	1101	1152
	Complex Code	374	5	353 (93.1%)	74	122
	Long Methods	283	9	260 (89%)	74	143
	SUM	8430	1238	7348 (76%)		
grunt	Variable Re-assign	2210	317	1636 (64.7%)	248	411
	Lengthy Lines	243	108	172 (49%)	248	681
	Complex Code	91	5	83 (85.4%)	248	292
	Long Methods	55	5	41 (68.3%)	248	334
	SUM	2732	448	2017 (63.4%)		
bower	Variable Re-assign	1427	1801	1235 (38.3%)	797	777
	Chained Methods	82	96	35 (19.7%)	1231	819
	Lengthy Lines	32	50	28 (34.1%)	644	719
	Nested Callbacks	38	32	27 (38.6%)	163	656
	SUM	1647	2087	1368 (36.6%)		
less	Variable Re-assign	36979	3779	37303 (91.5%)	56	281
	Assign. in Cond. State.	1349	4	1261 (93.2%)	405	477
	Lengthy Lines	547	26	509 (88.8%)	36	139
	This Assign	392	33	387 (91.1%)	129	314
	SUM	40241	3927	40391 (91.4%)		
request	Variable Re-assign	1362	667	1140 (56.2%)	365	534
	This Assign	28	50	34 (43.6%)	874	743
	Chained Methods	32	22	38 (70.4%)	256	557
	Nested Callbacks	1	28	1 (3.4%)	774	592
	SUM	1455	777	1232 (55.2%)		
jquery	Variable Re-assign	13076	5156	12122 (66.5%)	528	694
	Complex Switch Case	146	15	130 (80.7%)	179	435
	This Assign	118	37	120 (77.4%)	539	716
	Chained Methods	59	58	35 (29.9%)	657	785
	SUM	13743	5356	12675 (66.4%)		
hexo	Variable Re-assign	19023	823	17626 (88.8%)	2	86
	Lengthy Lines	768	12	675 (86.5%)	10	138
	Long Parameter List	755	3	728 (96%)	2	20
	Complex Code	599	19	576 (93.2%)	2	51
	SUM	22522	980	20584 (87.6%)		
leaflet	Variable Re-assign	5856	498	2241 (35.3%)	789.5	734
	Lengthy Lines	733	270	358 (35.7%)	203	354
	Nested Callbacks	123	489	114 (18.6%)	986	1029
	Long Methods	77	5	32 (39%)	911	752
	SUM	6997	1278	2855 (34.5%)		
ramda	Variable Re-assign	4720	1078	4656 (80.3%)	375	391
	Chained Methods	365	101	344 (73.8%)	241	372
	Long Parameter List	176	90	208 (78.2%)	206	396
	Complex Code	194	28	200 (90.1%)	375	364
	SUM	5668	1340	5599 (79.9%)		
chart	Variable Re-assign	5297	5696	4538 (41.3%)	406	365
	This Assign	199	388	86 (14.7%)	406	339
	Lengthy Lines	119	14	34 (25.6%)	12	154
	Complex Switch Case	53	30	31 (37.3%)	169	258
	SUM	5740	6207	4746 (39.7%)		
riot	Variable Re-assign	8331	2625	7866 (71.9%)	52	188
	Lengthy Lines	193	33	206 (91.2%)	7	150
	This Assign	63	92	61 (39.4%)	43	110
	Assign. in Cond. State.	107	47	80 (51.9%)	100.5	196
	SUM	9119	2937	8637 (71.6%)		
vue	Variable Re-assign	4199	5833	6587 (65.7%)	139	175
	Lengthy Lines	2947	4208	3518 (49.2%)	125	143
	Complex Code	414	675	679 (62.4%)	139	171
	Long Methods	259	417	421 (62.3%)	139	171
	SUM	8612	11712	12129 (59.7%)		
moment	Variable Re-assign	5642	11063	6154 (36.8%)	450	486
	Nested Callbacks	19	335	12 (3.4%)	492	464
	Complex Switch Case	117	69	114 (61.3%)	299	634
	Chained Methods	113	42	74 (47.7%)	243	389
	SUM	6135	11590	6561 (37%)		
webpack	Variable Re-assign	4643	627	3192 (60.6%)	352	593
	Nested Callbacks	379	54	276 (63.7%)	104	359
	Chained Methods	371	16	174 (45%)	492	589
	Lengthy Lines	182	38	86 (39.1%)	236	518
	SUM	5970	813	3987 (58.8%)		
webtorrent	Variable Re-assign	709	424	471 (41.6%)	335	370
	This Assign	108	53	88 (54.7%)	335	427
	Nested Callbacks	28	55	23 (27.7%)	453	406
	Chained Methods	12	2	9 (64.3%)	19.5	122
	SUM	869	535	591 (42.1%)		



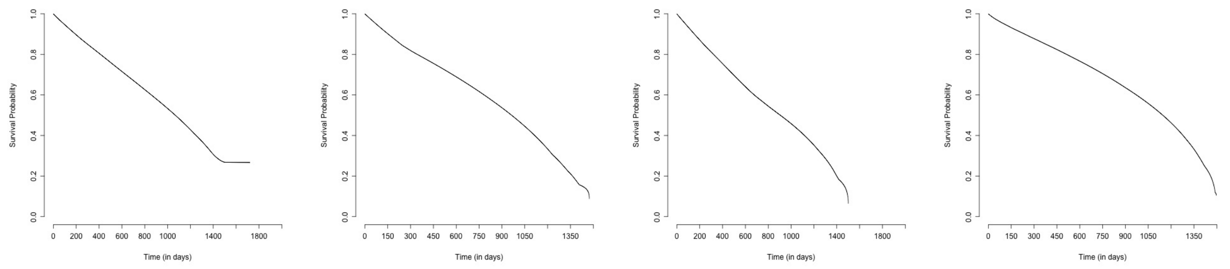
(a) Variable Re-Assign (b) Nested Callbacks (c) Complex Code (d) Long Methods

Figure 8: Survival analyzes of the largest smells of express.js.



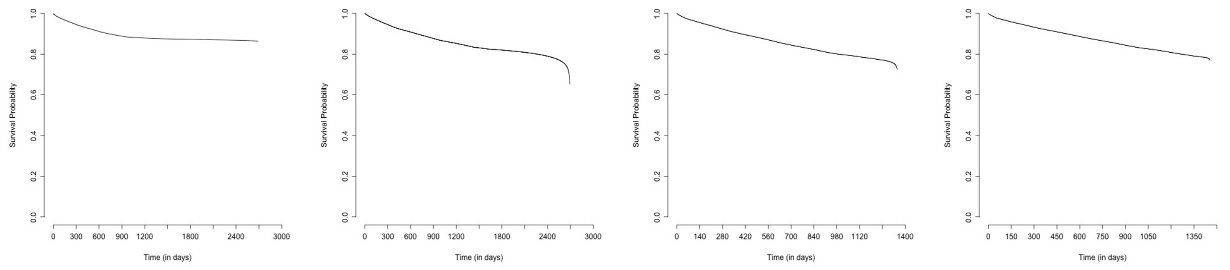
(a) Variable Re-Assign (b) Lengthy Lines (c) Complex Code (d) Long Methods

Figure 9: Survival analyzes of the largest smells of grunt.js.



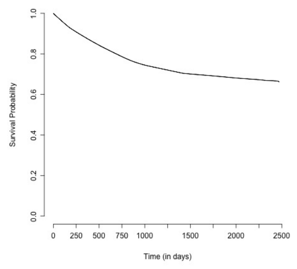
(a) Variable Re-Assign (b) Chained Methods (c) Lengthy Lines (d) Nested Callbacks

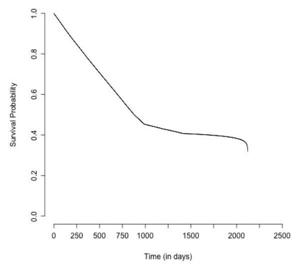
Figure 10: Survival analyzes of the largest smells of bower.js.

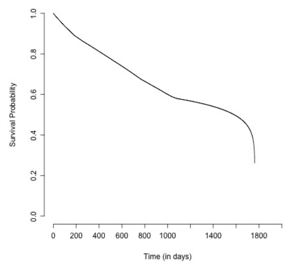


(a) Variable Re-Assign (b) Nested Callbacks (c) Complex Code (d) Long Methods

Figure 11: Survival analyzes of the largest smells of less.js.


(a) Variable Re-Assign


(b) This Assign


(c) Chained Methods

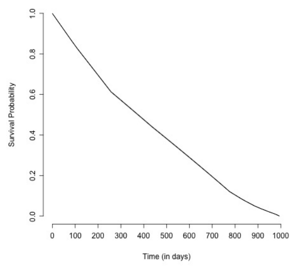
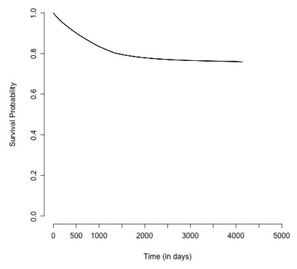
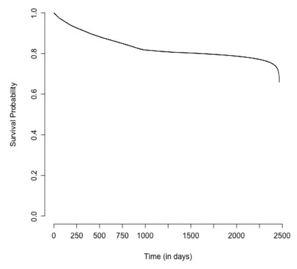
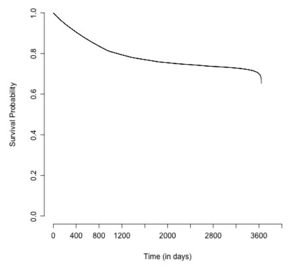

(d) Nested Callbacks

Figure 12: Survival analyzes of the largest smells of request.js.


(a) Variable Re-Assign


(b) Complex Switch Case


(c) This Assign

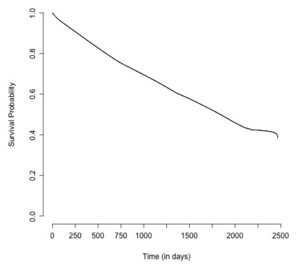
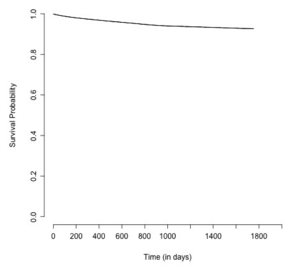
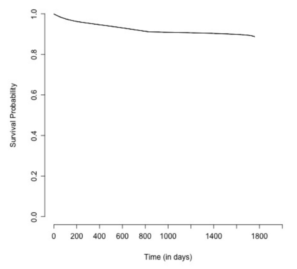
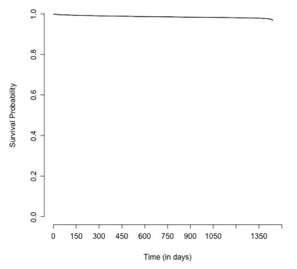

(d) Chained Methods

Figure 13: Survival analyzes of the largest smells of jquery.js.


(a) Variable Re-Assign


(b) Lengthy Lines


(c) Long Parameter List

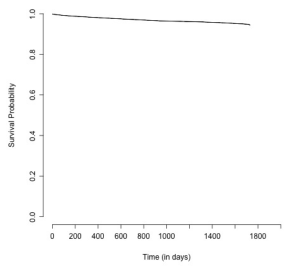
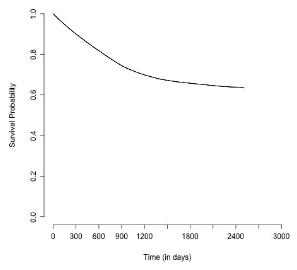
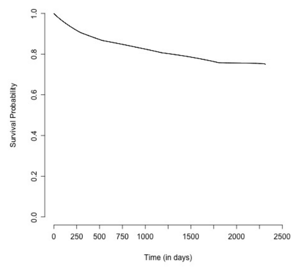
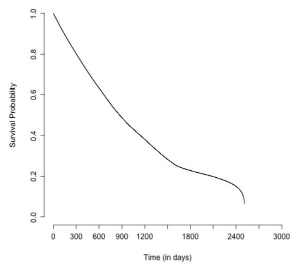

(d) Complex Code

Figure 14: Survival analyzes of the largest smells of hexo.js.


(a) Variable Re-Assign


(b) Lengthy Lines


(c) Nested Callbacks

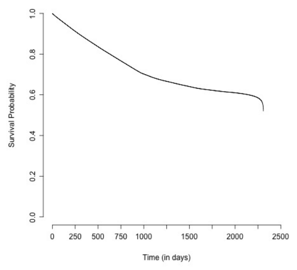
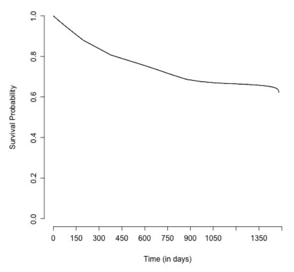
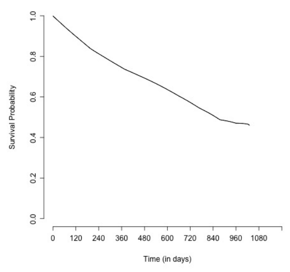

(d) Long Methods

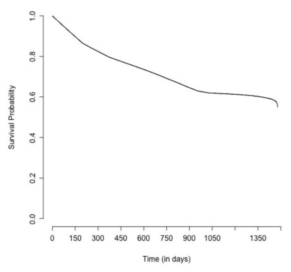
Figure 15: Survival analyzes of the largest smells of leaflet.js.



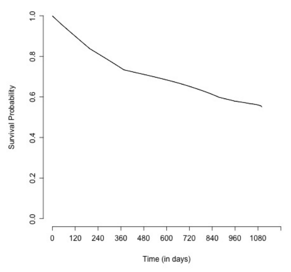
(a) Variable Re-Assign



(b) Chained Methods

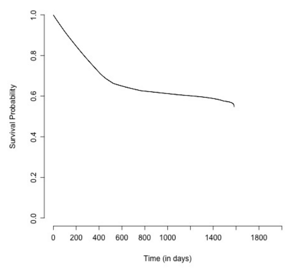


(c) Long Parameter List

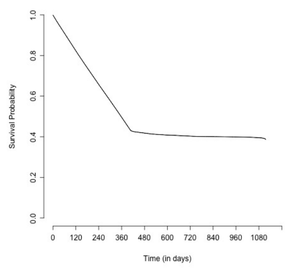


(d) Complex Code

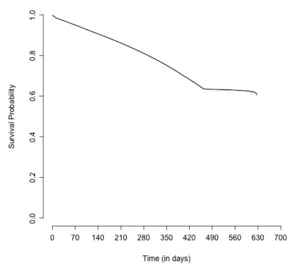
Figure 16: Survival analyzes of the largest smells of ramda.js.



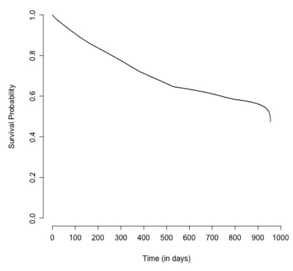
(a) Variable Re-Assign



(b) This Assign

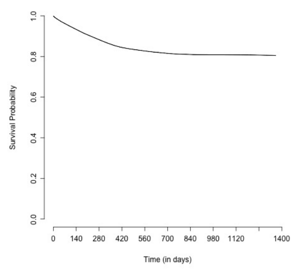


(c) Lengthy Lines

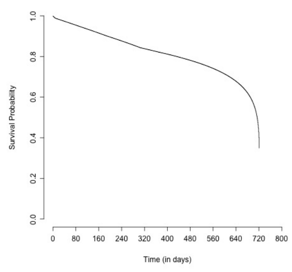


(d) Complex Switch Case

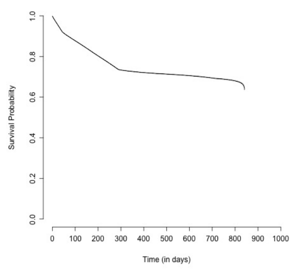
Figure 17: Survival analyzes of the largest smells of chart.js.



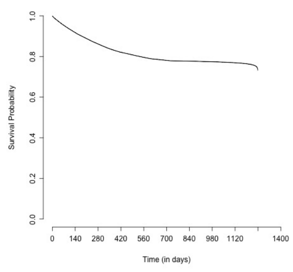
(a) Variable Re-Assign



(b) Lengthy Lines

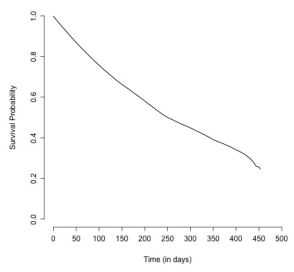


(c) This Assign

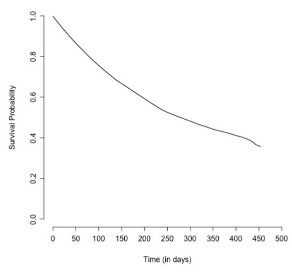


(d) Assignment in Conditional Statement

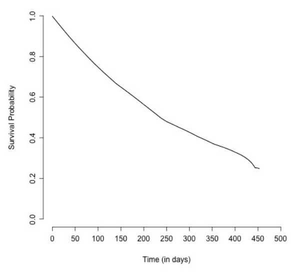
Figure 18: Survival analyzes of the largest smells of riot.js.



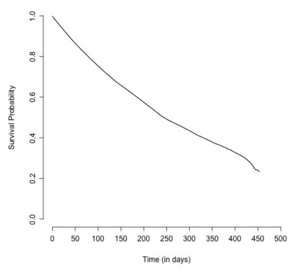
(a) Variable Re-Assign



(b) Lengthy Lines

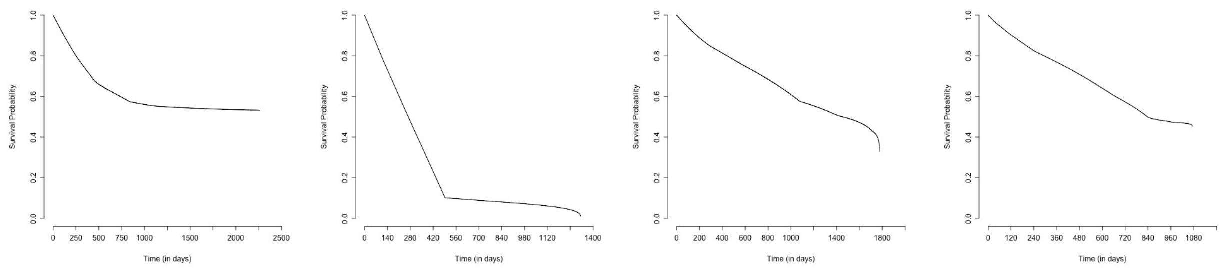


(c) Complex Code



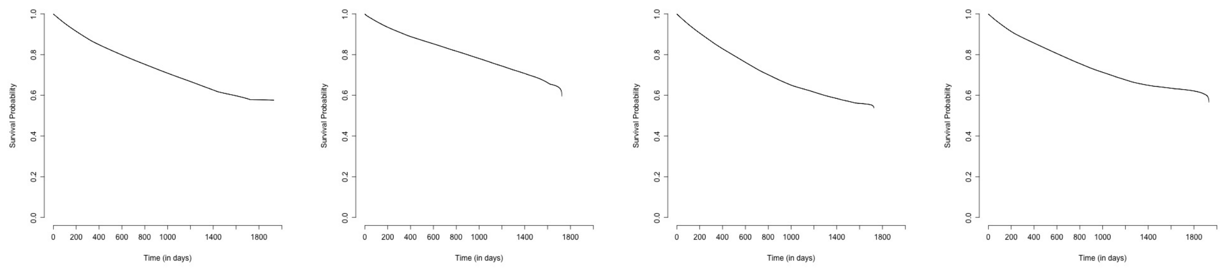
(d) Long Methods

Figure 19: Survival analyzes of the largest smells of vue.js.



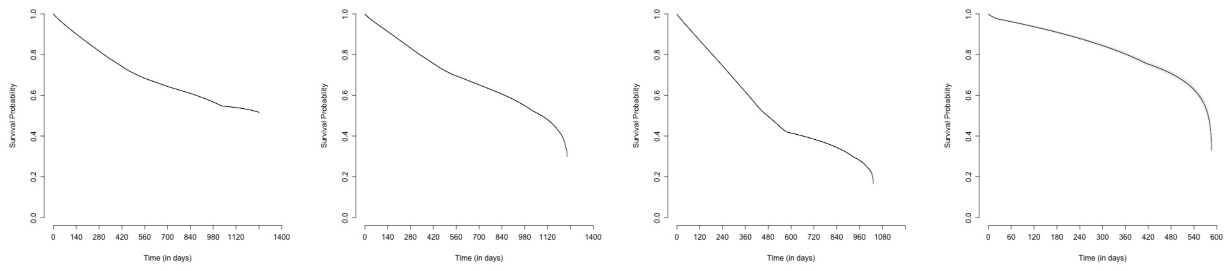
(a) Variable Re-Assign (b) Nested Callbacks (c) Complex Switch Case (d) Chained Methods

Figure 20: Survival analyzes of the largest smells of moment.js.



(a) Variable Re-Assign (b) Nested Callbacks (c) Chained Methods (d) Lengthy Lines

Figure 21: Survival analyzes of the largest smells of webpack.js.



(a) Variable Re-Assign (b) This Assign (c) Nested Callbacks (d) Chained Methods

Figure 22: Survival analyzes of the largest smells of webtorrent.js.

should be aware to their code when they create a JavaScript file, because it is precisely at this moment that most of the smells are introduced into the system. We also notice that, for the major part of the studied systems (eight out of fifteen), more than 20% of the smells created still survive presently; and for thirteen systems, over than 10% of the smells created are now present in those systems. It reveals that a significant part of the smells are never removed from the system once they are introduced in the code. Plus, after analyzing the commits of the studied systems, it is interesting to notice that most of time, the killed smells are removing at the same time than the file containing them (and not because of a file fix). The Table gives us also an overview of the smells lifetime, and we observe that for most of the systems (nine out of fifteen), the median and average days of survival of the most significant smell types are greater than 100 days; and for fourteen systems out of fifteen (except *hexo*), at least one of the most sizable smell types has an average and median lifetime greater than 100 days. This observation highlights that in general, smells tend to survive a very long time inside the system once they are introduced. Finally, Table XII presents an interesting result, which is that the smell *Variable Re-assign* is always the most considerable smell type (in our fifteen studied systems) in term of number of created smells, and its survival rate follows the trend of the sum of the smells (when we consider all the created smells of the system). For every studied system, over 1000 *Variable Re-assign* smells are created, and for eight systems out of fifteen, the number of created smells of this type exceeds 10000. Once again, *Variable Re-assign* is at the heart of our analysis, because as said previously, it is one of the most risky smell in terms of fault-proneness and vulnerability. According to our Figures 8 to 22, the four most significant smell types of each studied system have a considerable chance of surviving 500 days after their introduction. This is indeed the case for all the most significant smell types for nine systems out of fifteen (except *request*, *chart*, *moment*, *webtorrent*, and *vue*), with over 50% chance of surviving 500 days after the introduction of their largest smell types. Also, for fourteen studied systems out of fifteen (except *vue*), at least one of the most sizable smell types has more than 50% chance of surviving 500 days after its smells introduction. Plus, for twelve systems out of fifteen (except *bower*, *vue*, and *webtorrent*), the *Variable Re-assign* smell type has over than 50% chance of surviving 1500 days after its introduction. These observations show the trend of the smells of the studied systems to be persistent and survive a long time after their were introduced into the code, and also the significance of *Variable Re-assign* which is strongly linked to fault-proneness, and is the most proliferated smell type in the studied systems with a very high chance of surviving over time.

Most of the studied smells (from 34.5% to 91.4%) are introduced during the creation of JavaScript files. Once introduced, in most of half of the cases (eight systems), over than 20% of the studied smells are not removed and have a high chance of surviving a very long time. Plus, Variable Re-assign, which is the most subject to fault-proneness and vulnerability, is also the most sizable smell type with the highest chance of surviving over time.

V. Perceived Criticality of Code Smells by JavaScript Developers

To understand the perception of developers towards our studied code smells, we conducted a qualitative study with JavaScript developers. In total 1,484 developers took part in our qualitative study. The survey consisted of 3 questions about the participant background and 15 questions about the studied code smells. We designed a website [33] to run the survey. The study took place between October 4th and October 17th, 2016. The link to the survey was shared within the *Hacker News community* [34] and the *EchoJS community* [35]. Participants were free to skip any question and they could leave the survey at any time. However, none of the participants used the skip button. 68% of the participants to our survey had more than 3 years of experience writing Javascript applications. We asked the participants about their usages of JavaScript and found that 92% of them use JavaScript to write client-side applications and 51% use it for server side applications. Over 63% of participants were familiar with the concept of *code smell* and 19% never heard of it.

The results of our survey showed that 20% of participants use pure callbacks to handle asynchronous logic, while 66% use *Promises* and 13% use the newest ES6 and ES7 features to control the flow of asynchronous codes. 92% of participants indicated that nesting the callbacks makes the code harder to maintain.

[33]https://srvy.online/js
[34]https://news.ycombinator.com/
[35]http://www.echojs.com/

86% of our participants reported that they prefer codes using `const` instead of `var` to declare variables and not re-using them in the same scope. 73% indicated that re-using variables makes the code harder to maintain.

Surprisingly, 74% of our participants said they preferred having assignments in conditional statements while using *Regular Expressions*, however, 54% of them acknowledged that this practice makes the code harder to maintain.

55% of our participants reported that they prefer using `.bind(this)` instead of assigning `this` to other variables. However, only 16% of the participants indicated that they use `.bind`. 55% of the participants indicated that they use *arrow functions* to have lexical `this`.

Although the JavaScript documentation lists *Complex Switch Case* as a code smell, only 14% of our participants preferred `if/else` structures over `switch/case`.

In the survey, we asked participants to rank the 12 studied code smells on a Likert scale from 1 to 10, based on their impact on the software understandability, debugging and maintenance efforts. Results show that participants consider *Nested Callbacks* to be the most hazardous code smells (with a rating of 8.1/10), followed by *Variable Re-assign* (with a rating of 6.5/10) and *Long Parameter List* (with a rating of 6.2/10). They claimed that these code smells negatively affect the maintainability and reliability of JavaScript systems. This assessment is in line with the findings of our quantitative analysis.

VI. Contributions (part to remove)

In this section, we discuss the contributions that I provided, in order to make myself clear about what I have done during these four last months. All the modifications are colored in blue in this paper, and all the Tables and Figures have been updated.

My first task was to understang Amir Saboury's code that he did to produce this paper (first version). The difficulty was essentially the lack of code documentations, which makes my objective harder to achieve. More specifically, some parts of the code were missing, like the smells information report, and I had to write the missing parts of this code. Some Javascript library (ESLint) needed to be modified in order to print correctly the information about smells localization and weight, to easily parse the smells report and do the next tasks. Then, I reused Amir Saboury's code to get all the information I needed about smell information (after modifying myself the code like I said), and did my second task, which was to make the smells genealogy. All my methology has already been presented in the previous sections (sections III-B and IV). Then, I rewrited Amir Saboury's code about combining smells detection and fixes detection (in Python), for consistency and understanding reasons, and I then reused this Python code to make the combination between smells detection and vulnerabilities detection. I reused also the R file created by Amir Saboury to make all the Cox survival and hazard models.

Finally, one of my aims was to extend the original paper by introducing new systems. Thus, I introduced ten new systems, and their selection was obviously based on their popularity inside the JavaScript community.

All the code that I writed is present on my Github repository[36], as well as all the results. In this paper, I condensed the results that I got not to overload the paper, but all the detailed results appear on my Github repository (*Results* repository). All the code is commented (unfortunately in French) for comprehension reasons and for a better reuse.

VII. Threats to validity

In this section, we discuss the threats to validity of our study following common guidelines for empirical studies [46].

Construct validity threats concern the relation between theory and observation. In our study, threats to the construct validity are mainly due to measurement errors. The number of previous faults in each source code file was calculated by identifying the files that were committed in a fault fixing revision. This technique is not without flaws. We identified fault fixing commits by mining the logs searching for certain keywords (*i.e.,* "bug","fix"',"defect" and "patch") as explained in Section III-B. Following this approach, we are not able to detect fault fixing revisions if the committer either misspelled the keywords or failed to include any commit message. Nevertheless, this heuristic was successfully used in multiple previous studies in software engineering [6], [47]. The SZZ heuristic used to identify fault-inducing commits is not 100% accurate. However, it has been successfully used in multiple previous studies from the literature, with satisfying results. In our implementation, we remove all fault-inducing commit candidates that only changed blank or comment lines. When analyzing the *smelliness* of files that experienced fault-inducing changes, we only tracked the presence of the smell in the file as a whole. Hence, the smell contained in the file may not have been involved in the changed lines that induced the fault.

Internal validity threats concern our selection of systems and tools. The metric extraction tool used in this paper is based on the AST provided by ESLint. The results of the study are therefore dependent on the accuracy of ESLint. However, we are rather assured that this tool functions properly as it is being used widely by big companies. *e.g.,* Facebook, Paypal, Airbnb. We chose a logarithmic link function for some of our covariates in the survival analysis. It is possible that a different link function would be a better choice for these covariates. However, the non-proportionality test implies that the models were a good fit for the data. Also, we do not claim causation in this work, we simply report observations and correlations and tries to explain these findings.

Threats to conclusion validity address the relationship between the treatment and the outcome. We are careful to acknowledge the assumptions of each statistical test.

[36]https://github.com/DavidJohannesWall/smells_project

Threats to external validity concern the possibility to generalize our results. In this paper, we have studied fifteen large JavaScript projects. We have also limited our study to open-source projects. Still, these projects represent different domains and various project sizes. Table I shows a summary of the studied systems, their domain and their size. Nevertheless, further validation on a larger set of JavaScript systems, considering more types of code smells is desirable.

Threats to reliability validity concern the possibly of replicating our study. In this paper, we provide all the details needed to replicate our study. All our fifteen subject systems are publicly available for study. The data and scripts used in this study is also publicly available on Github[37].

Threats to external reliability concern the use of the vulnerability database Snyk[38] to collect vulnerabilities on our studied systems. As said previously, this database presents a lack of completeness and accuracy, and we are aware that a better vulnerability database should be used to provide better observations and conclusions about the vulnerability comparison between files with and without code smells.

Threats to internal genealogy construction is about our way to get the smells genealogy of the studied smells, more specifically the recognition of the smells over time and commits. Indeed, we set a similarity threshold of 70%, meaning that if two smells of the same type have a similarity greater than 70%, there are likely the same. Obviously, this threshold is not perfect and can associate two different smells together, or dissociate two smells, which are in reality the same. However, we changed it in order to see if some significant differences would appear, but no relevant difference was revealed.

VIII. Related Work

In this section, we discuss the related literature on code smell and JavaScript systems. Code Smells [4] are poor design and implementation choices that are reported to negatively impact the quality of software systems. They are opposite to design patterns [48] which are good solutions to recurrent design problems. The literature related to code smells generally falls into three categories: (1) the detection of code smells (e.g., [3], [49]); (2) the evolution of code smells in software systems (e.g., [50]–[53]) and their impact on software quality (e.g., [6], [53]–[56]); and (3) the relationship between code smells and software development activities (e.g., [56], [57]).

Our work in this paper falls into the second category. We aim to understand how code smells affect the fault-proneness of JavaScript systems. Li and Shatnawi [54] who investigated the relationships between code smells and the occurrence of errors in the code of three different versions of Eclipse reported that code smells are positively associated with higher error probability. In the same line of study, Khomh et al. [55] investigated the relationship between code smells and the

[37]https://github.com/DavidJohannesWall/smells_project
[38]https://snyk.io/test

change- and fault-proneness of 54 releases of four popular Java open source systems (ArgoUML, Eclipse, Mylyn and Rhino). They observed that classes with code smells tend to be more change- and fault-prone than other classes. Tufano et al. [53] investigated the evolution of code smells in 200 open source Java systems from Android, Apache, and Eclipse ecosystems and found that code smells are often introduced in the code at the beginning of the projects, by both newcomers and experienced developers. Sjoberg et al. [57], who investigated the relationship between code smells and maintenance effort reported that code smells have a limited impact on maintenance effort. However, Abbes et al. [56] found that code smells can have a negative impact on code understandability. Recently, Fard et al. [3] have proposed a technique named JNOSE to detect 13 different types of code smells in JavaScript systems. The proposed technique combines static and dynamic analysis. They applied JNOSE on 11 client-web applications and found "lazy object" and "long method/function" to be the most frequent code smells in the systems. WebScent [58] is another tool that can detect client-side smells. It identifies mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. ESLint [11], JSLint [59] and JSHint [60] are rule based static code analysis tools that can validate source codes against a set of best coding practices. Despite this interest in JavaScript code smells and the growing popularity of JavaScript systems, to the best of our knowledge, there is no study that examined the effect of code smells on the fault-proneness of JavaScript server-side projects. This paper aims to fill this gap.

IX. Conclusion

In this study, we examine the impact of code smells on the fault-proneness and vulnerability of JavaScript systems. Also, we present a survival study of the smells of JavaScript systems. We present a quantitative study of fifteen JavaScript systems that compare the time until a fault occurrence or a vulnerability appearance in JavaScript files that contain code smells and files without code smells, with three different approaches: file grain, line grain, and line grain including dependencies approaches. This quantitative study also present some descriptive statistics about the twelve studied smells, as well as their survival by computing their lifetime. Results show that JavaScript files without code smells have hazard rates 76% lower than JavaScript files with code smells in the file grain study, and 38% lower than JavaScript files with code smells in the line grain including dependencies study. In other terms, the survival of JavaScript files against the occurrence of faults increases with time if the files do not contain code smells. We further investigated hazard rates associated with different types of code smells and found that "Variable Re-assign", "Assignment in Conditional Statements" Complex Code smells have the highest hazard rates. However, in regards to vulnerability study, we could not say that JavaScript files with code smells are more vulnerable than those without code smells, but we still consider that a better vulnerability database needs to be used in order to get more precise conclusions. Nevertheless, we found that "Variable Re-assign" and "This Assign" code smells are more subject to vulnerability than the other smell types. The survival results show us that smells are introduced at the JavaScript files creation most of the time, and a big part of them still survived presently; those smells, and particularly "Variable Re-assign" which is the most proliferated into the studied systems, have a high chance of surviving a very long time. In addition, we conducted a survey with 1,484 JavaScript developers, to understand the perception of developers towards our studied code smells, and found that developers consider *Nested Callbacks*, *Variable Re-assign*, *Long Parameter List* to be the most hazardous code smells. JavaScript developers should consider removing *Variable Re-assign* code smells from their systems in priority since this code smell is consistently associated with a high risk of fault, the highest risk of vulnerability, and because it is the most sizable code smell with a high chance of surviving over time. They should also prioritize *Assignment in Conditional Statements*,*Complex Code*, *This Assign*, *Nested Callbacks*, and *Long Parameter List* code smells for refactoring.

References

[1] Stackoverflow, "Developer survey results 2016," 2016, [Online; accessed 11-August-2016]. [Online]. Available: http://stackoverflow.com/research/developer-survey-2016

[2] Githut, "Discover languages in github," 2016, [Online; accessed 11-August-2016]. [Online]. Available: http://githut.info/

[3] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 116–125.

[4] M. Fowler, "Refactoring: Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[5] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9171-y

[6] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness." in *WCRE*, 2013, pp. 351–360.

[7] "npm-coding-style," 2016, [Online; accessed 17-October-2016]. [Online]. Available: https://docs.npmjs.com/misc/coding-style

[8] "Node.js style guide," 2016, [Online; accessed 17-October-2016]. [Online]. Available: https://github.com/felixge/node-style-guide

[9] "Airbnb javascript style guide," 2016, [Online; accessed 17-October-2016]. [Online]. Available: https://github.com/airbnb/javascript

[10] "jquery javascript style guide," 2016, [Online; accessed 17-October-2016]. [Online]. Available: https://contribute.jquery.org/style-guide/js/

[11] "Eslint: The pluggable linting utility for javascript and jsx. http://eslint.org/."

[12] "Wordpress javascript coding standards," 2016, [Online; accessed 17-October-2016]. [Online]. Available: https://make.wordpress.org/core/handbook/best-practices/coding-standards/javascript/

[13] J. Chaffer, *Learning JQuery 1.3: Better Interaction and Web Development with Simple JavaScript Techniques*. Packt Publishing Ltd, 2009.

[14] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[15] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[16] L. Griffin, K. Ryan, E. de Leastar, and D. Botvich, "Scaling instant messaging communication services: A comparison of blocking and non-blocking techniques," in *Computers and Communications (ISCC), 2011 IEEE Symposium on*. IEEE, 2011, pp. 550–557.

[17] E. Brodu, S. Frénot, and F. Oblé, "Toward automatic update from callbacks to promises," in *Proceedings of the 1st Workshop on All-Web Real-Time Systems*. ACM, 2015, p. 1.

[18] K. Gallaba, A. Mesbah, and I. Beschastnikh, "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)*. IEEE, 2015, pp. 1–10.

[19] M. Ogden, "Callback hell," 2015, [Online; accessed 14-August-2016]. [Online]. Available: http://callbackhell.com

[20] J. Archibald, "Es7 async functions," 2014, [Online; accessed 14-August-2016]. [Online]. Available: https://jakearchibald.com/2014/es7-async-functions/

[21] "Sei cert c++ coding standard - exp19-cpp. do not perform assignments in conditional expressions," [Online; accessed 23-August-2016]. [Online]. Available: https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=975

[22] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[23] D. Crockford, *JavaScript: The Good Parts: The Good Parts*. O'Reilly Media, Inc., 2008.

[24] R. Marinescu and M. Lanza, "Object-oriented metrics in practice," 2006.

[25] R. C. Martin, "The open-closed principle," *More C++ gems*, pp. 97–112, 1996.

[26] F. Martin, B. Kent, and B. John, "Refactoring: improving the design of existing code," *Refactoring: Improving the Design of Existing Code*, 1999.

[27] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[28] A. Mardan, *Express. js Guide: The Comprehensive Book on Express. js*. Azat Mardan, 2014.

[29] "About bower," 2016, [Online; accessed 4-October-2016]. [Online]. Available: https://bower.io/docs/about/

[30] "Who uses grunt," 2016, [Online; accessed 4-October-2016]. [Online]. Available: http://gruntjs.com/who-uses-grunt

[31] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[32] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[33] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[34] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[35] F. Pfenning and C. Elliott, "Higher-order abstract syntax," in *ACM SIGPLAN Notices*, vol. 23, no. 7. ACM, 1988, pp. 199–208.

[36] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[37] D. Mazinanian and N. Tsantalis, "Migrating cascading style sheets to preprocessors by introducing mixins," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 672–683.

[38] J. Fox and S. Weisberg, *An R companion to applied regression*. Sage, 2010.

[39] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Software Engineering*, vol. 13, no. 5, pp. 473–498, 2008.

[40] J. D. Singer and J. B. Willett, *Applied longitudinal data analysis: Modeling change and event occurrence*. Oxford university press, 2003.

[41] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 13–21.

[42] H. Westergaard, *Contributions to the History of Statistics*. P.S. King, London, 1932.

[43] A. G. Koru, D. Zhang, and H. Liu, "Modeling the effect of size on defect proneness for open-source software," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007, p. 10.

[44] T. M. Therneau and P. M. Grambsch, *Modeling survival data: extending the Cox model*. Springer Science & Business Media, 2000.

[45] T. Therneau, "R survival package," 2000.

[46] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[47] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[48] E. Gamma, R. Helm, R.Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, 1995.

[49] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, Apr. 2011.

[50] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the*. IEEE, 2010, pp. 106–115.

[51] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *3rd Int'l Symposium on Empirical Software Engineering and Measurement, ESEM 2009,*, 2009, pp. 390–400.

[52] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conf. on*. IEEE, 2012, pp. 411–416.

[53] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414.

[54] R. Shatnawi and W. Li, "An investigation of bad smells in object-oriented design," in *Information Technology: New Generations, 2006. ITNG 2006. 3rd Int'l Conf. on*. IEEE, 2006, pp. 161–165.

[55] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[56] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on*, March 2011, pp. 181–190.

[57] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[58] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 282–285.

[59] "Jslint: The javascript code quality tool. http://www.jslint.com."

[60] "Jshint: A static code analysis tool for javascript. http://jshint.com/."