Detecção automática de falhas em software JavaScript: Uma revisão sistemática da literatura

Alternative Title: Automatic fault detection in JavaScript software: A systematic review of the literature

Charles Mendes de Macedo
Universidade de São Paulo
charlesmendes@usp.br

Karina Valdivia Delgado
Universidade de São Paulo
kvd@usp.br

Igor Wiese
Universidade Tecnológica Federal do Paraná
igor@utfpr.edu.br

RESUMO

As aplicações desenvolvidas com a linguagem JavaScript, vêm aumentando a cada dia, não somente aquelas baseadas em Web (*client-side*), como também as aplicações para servidor (*server-side*) e dispositivos móveis (*mobile*). Neste contexto, a necessidade de ferramentas para identificação de falhas é fundamental, para auxiliar desenvolvedores durante a evolução destas aplicações. Diferentes ferramentas e abordagens têm sido propostas ao longo dos anos e este conhecimento encontra-se fragmentado na literatura, o que dificulta os desenvolvedores a escolherem as melhores ferramentas para identificação de falhas. Desta forma, o objetivo desta pesquisa é analisar sistematicamente os métodos e ferramentas que estão sendo utilizadas para detecção automática de falhas em software escrito em JavaScript. Além de catalogar as falhas mais procuradas pelos estudos em cada ambiente e identificar as lacunas existentes na área. Para tal, foi realizada uma revisão sistemática da literatura (RSL), que resultou em 19 estudos primários relevantes para o objetivo desta pesquisa. A maioria desses estudos estão voltados para ambiente Web (*client-side*), mostrando a falta de trabalhos para detecção de falhas em ambientes *server-side* e multiplataforma.

Palavras-Chave

Detecção de falhas, *JavaScript*, *análise estática*

ABSTRACT

Applications developed with the JavaScript language, have been increasing every day, not only those based on Web (client-side), but also server-side applications and mobile devices. In this context, the need for tools to identify failures is fundamental, to help developers during the evolution of these applications. Different tools and approaches have been proposed over the years and this knowledge is fragmented in the literature, which makes it difficult for developers to choose the best tools for fault identification. In this way, the objective of this research is to systematically analyze the methods and tools that are being used for automatic detection of failures in software written in JavaScript. Besides cataloging the failures most sought in each environment and identify the gaps in the area. For this, a systematic review of the literature was carried out, which resulted in 19 primary studies relevant to the objective of this research. Most of these studies focus on the web-environment (client-side), showing the lack of research to detect failures in server-side and multiplatform environments.

SBSI 2017 June 5^{th} – 8^{th}, 2017, Lavras, Minas Gerais, Brazil

CCS Concepts

•Software and its engineering → Software defect analysis; Software testing and debugging;

Keywords

Fault Detection, JavaScript, Static Analysis.

1. INTRODUÇÃO

Com o crescimento da utilização de software web e aplicativos em dispositivos no geral, a necessidade de desenvolver sistemas de software com alta qualidade aumenta a cada dia. Nesse contexto, a detecção precoce de falhas ajuda no desenvolvimento de software e faz com que a correção seja menos onerosa e a entrega do software passiva de poucas manutenções. A detecção de falhas visa encontrar o código defeituoso depois que aconteceu a falha, levando em consideração a análise do contexto em que ocorreu o erro. Essa detecção pode ser realizada durante a escrita do código ou após o desenvolvimento.

Segundo Hanam [8], atualmente JavaScript é considerada uma das linguagens mais populares usada por desenvolvedores. JavaScript além de ser uma linguagem interpretada por todos os navegadores web (*client-side*), atualmente é executada em ambientes de desktop (*server-side*) e em dispositivos móveis, como aplicativos. Assim, aparecem preocupações distintas que os desenvolvedores, arquitetos e engenheiros de software têm que levar em consideração sobre o ambiente em que será desenvolvida a aplicação ou o software. Por exemplo, em aplicações *client-side*, a versão do navegador interfere na execução do código JavaScript, podendo apresentar falta de compatibilidade para interpretar

algum código [27].

Além disso, JavaScript tem suas particularidades únicas, como ser uma linguagem dinâmica, fracamente tipada e assíncrona. A linguagem suporta recursos intrincados, como protótipos, funções de primeira classe e funções que se referem a variáveis independentes que estão dentro de uma função superior (*closures*). O código fonte pode ser desenvolvido no paradigma estrutural, funcional e orientado à objeto. Também é importante destacar que a maioria dos projetos possui um grande número de dependências com bibliotecas de terceiros (*plug-ins*) e *APIs* externas [22].

Desta forma, torna-se um grande desafio realizar detecção de falhas em códigos JavaScript. Diferentes ferramentas e abordagens têm sido propostas ao longo dos anos e este conhecimento encontra-se fragmentado na literatura, o que dificulta os desenvolvedores escolherem as melhores ferramentas para identificação de falhas. Desta forma, o objetivo desta revisão sistemática da literatura (RSL) é identificar os métodos e ferramentas para detecção automática de falhas em sistemas de software desenvolvidos na linguagem JavaScript. Buscando catalogar as falhas mais procuradas pelos estudos, identificando os ambientes de desenvolvimento que estão sendo mais usados e descobrindo as lacunas existentes na área.

2. FUNDAMENTOS TEÓRICOS

2.1 Terminologia usada

Os termos erro, falha e defeito (*bug*) estão comumente relacionados a sistemas de software como aplicativos, *sites* e programas de computadores. Neste trabalho, utilizaremos as definições apresentadas em [23] e que são brevemente descritas a seguir.

Erro: é uma ação que produziu um resultado incorreto. No nosso caso, a ação foi realizada por um humano na escrita do código fonte, que resultou em um comportamento incorreto no software.

Falha: é o impacto com que o erro pode interferir na execução comum do código escrito, fazendo com que mais erros sejam propagados no fluxo das execuções. Por exemplo, quando ocorre um erro em uma função *f1* que deveria retornar um resultado para ser utilizado por outra função *f2*, essa função *f2* vai receber um valor incorreto. E o mesmo pode continuar ocorrendo se outra função estiver esperando o resultado da função *f2*.

Defeito: é o resultado, como um todo, das execuções inesperadas ou indesejadas, provenientes da falha de um erro. Por exemplo, quando um usuário clica em um botão, esperando um resultado e não ocorre nada.

Assim, é possível concluir que procurar um defeito significa entender o fluxo antecessor para encontrar a origem do erro.

2.2 Prevenção e detecção de falhas

Segundo Ocariza [23], há duas abordagens utilizadas pelos pesquisadores para lidar com erros, falhas e defeitos: a prevenção e a detecção.

2.2.1 Prevenção

A prevenção envolve encontrar maneiras de minimizar o número de erros cometidos na etapa primária do desenvolvimento de um software, i.e., durante o processo da escrita do código [23]. Muitos ambientes de desenvolvimento integrado (IDE - *Integrated Development Environment*) auxiliam no desenvolvimento, visando prevenir erros. Contudo, é uma abordagem limitada, pois existem erros que não são provenientes do código escrito, por exemplo erros na biblioteca de terceiros, ou recebimento de um parâmetro inesperado. Desta forma, a prevenção auxilia o desenvolvedor durante a escrita do código, mas não garante total confiabilidade do código.

2.2.2 Detecção

A detecção visa encontrar o código defeituoso, levando em consideração a análise do contexto em que ocorreu o erro. Essa abordagem pode ser realizada durante a escrita do código, auxiliando o desenvolvedor na detecção de erros sintáticos ou semânticos da linguagem [23]. Também pode ser realizada após o desenvolvimento, com a execução do software ou trechos de código, para detectar a causa da falha [23, 24]. A análise da causa do erro é realizada sobre o fluxo de execução do código como um todo. Por exemplo, se o erro ocorreu ao receber um parâmetro inesperado, será possível identificar qual foi o parâmetro recebido. Nesta RSL estamos interessados na detecção de falhas.

3. MÉTODO

3.1 Planejamento

3.1.1 Questão de pesquisa

Na Tabela 1 são apresentadas as questões de pesquisa usadas nesta RSL. Na primeira questão (RQ1) é levantado um dos aspectos mais importantes nesta RSL, que é identificar quais os métodos utilizados para detecção automática de *falhas* em JavaScript.

A questão RQ2 tem o objetivo determinar algumas das características dos métodos propostos. Para isso, foram identificados: (i) os tipos de falhas que podem ser encontradas pelos métodos de detecção automática (RQ2-1); (ii) se os métodos propostos na literatura têm alguma base de dados com as falhas catalogados (RQ2-2); (iii) se as abordagens da literatura propostas para detecção de falhas são para *client-side*, *server-side*, *mobile* ou para aplicativos (RQ2-3); e (iv) se a detecção é realizada em tempo de desenvolvimento ou pós desenvolvimento (RQ2-4).

A RQ3 tem por objetivo determinar algumas das características dos experimentos realizados. Para isso são identificados (i) os sistemas de software que são utilizados nos experimentos (RQ3-1); (ii) o tipo do repositório de código fonte utilizado para realizar os testes dos métodos de detecção (RQ3-2); (iii) se são feitas comparações com outras técnicas (RQ3-3); (iv) o número de projetos ou aplicações utilizadas nos experimentos (RQ3-4); e (v) os principais resultados obtidos em termos de precisão, cobertura e tempo (RQ3-5).

3.1.2 Seleção de Fontes e String de Busca

Utilizamos duas bases de dados nesta RSL, Scopus e Web of Science, pois sabe-se que o motor de pesquisa do Scopus indexa as bases de dados da IEEE Xplore, Science Direct, ACM Digital Library e Engineering Village. Na *string* de busca foram incorporados os termos e sinônimos alternativos. Assim, foram geradas as *strings* para as duas bases de dados, com base em alguns artigos de controle, como visto na Tabela 2.

Table 1: Questões de Pesquisa

RQ#	Questões de Pesquisa
RQ1	Quais os métodos usados para realizar a detecção automática de *falhas*?
RQ2	Quais as características dos métodos propostos?
RQ2-1	Que tipo de *falhas* são detectados pelos métodos propostos?
RQ2-2	Os métodos propostos têm alguma base de dados pré-definida de *falhas*?
RQ2-3	Os métodos propostos são utilizados em JavaScript *client-side server-side*, *mobile* ou aplicativos?
RQ2-4	Os métodos de detecção de falha são executados durante o desenvolvimento do software ou após?
RQ3	Quais as características dos experimentos realizados?
RQ3-1	Quais os sistemas de software utilizados nos experimentos?
RQ3-2	Os testes são realizados com repositórios de código fonte públicos ou privados?
RQ3-3	Com quais outras técnicas foram comparados os métodos propostos?
RQ3-4	Quantos projetos ou aplicações foram utilizados nos experimentos?
RQ3-5	Quais foram os principais resultados dos experimentos em termos de precisão, cobertura e tempo?

Table 2: String de busca

Fonte	String
Scopus	*TITLE-ABS-KEY ((detection OR (code AND analysis) OR (static AND analysis)) AND (bug OR error OR fault)) AND TITLE-ABS-KEY (java*script)*
Web of Science	*(TS= ((detection OR (code AND analysis) OR (static AND analysis)) AND (bug OR error OR fault)) AND TS= (javascript)*

3.1.3 Critérios de Inclusão e Exclusão

Os critérios de inclusão (CI) e exclusão (CE) foram desenvolvidos para selecionar os estudos primários, estabelecendo critérios mínimos de qualidade e assim, selecionar fontes acessíveis para consultas posteriores. Os critérios foram:

CI-1 Os artigos devem estar disponíveis integralmente online ou de forma gratuita via a instituição dos autores da RSL;

CI-2 Os artigos devem estar inteiramente escritos em língua inglesa.

CE-1 Trabalho de detecção de falhas que não são automatizados.

CE-2 Artigos que não abordem principalmente detecção de falhas em sistemas implementados em JavaScript.

CE-3 Artigos publicados em veículos que não exigem revisão por pares, por exemplo, capítulos de livro.

CE-4 Estudos secundários ou terciários.

CE-5 Estudos não relacionados às áreas de Ciência da Computação, Matemática e Engenharia.

CE-6 Estudos incompletos ou que não relatam de maneira adequada os métodos ou experimentos realizados.

CE-7 Artigos focados em testes e criação de testes automáticos.

CE-8 Artigos com ênfase em vulnerabilidade de segurança em software escrito em JavaScript.

O critério CE-7 foi considerado pois segundo Ocariza [23] testes não são suficientes para melhorar a confiabilidade do código, sendo que os desenvolvedores ainda teriam que depurar quaisquer problema detectado pelos casos de teste.

3.2 Condução

A pesquisa realizada no motor de busca Scopus em 8 de fevereiro de 2017 retornou 95 trabalhos e no motor Web of Science retornou 44 estudos. Do total de 139 artigos, 16 foram excluídos pelos critérios de exclusão CE-3, CE-4, CE-5, CE-7 por meio das ferramentas de filtros dos motores de busca, restando 123. Destes 123, foram identificados 42 artigos duplicados. Após isso, os critérios foram aplicados manualmente em cada um dos 81 estudos restantes analisando o título, resumo e palavras chaves, e no caso de dúvida o estudo foi analisado por completo. No fim desta etapa foram selecionados 19 estudos para o desenvolvimento da RSL.

Para auxiliar na condução desta RSL foi utilizado o software StArt *(State of the Art through Systematic Review)*[1].

4. RESULTADOS E DISCUSSÃO

Na Tabela 3 são listados os estudos selecionados ordenados pelo ano de publicação. O identificador ID inclui o veículo de publicação (C–Congresso e J–Periódico) e um número de sequência. Note que, apesar de não termos restringido o ano de publicação nos motores de busca, todos os estudos selecionados são recentes (publicados a partir de 2009).

4.1 Catálogo de falhas em JavaScript

Um dos poucos trabalhos que fazem uma categorização de falhas é Hanam [8]. Porém, nesse trabalho é apresentada apenas uma lista pequena de falhas mais comuns. Assim, um dos objetivos desta RSL é fornecer um catálogo de falhas mais completo criado a partir dos estudos selecionados, o qual é apresentado na Tabela 4. Cada falha apresenta um identificador (ID), o nome, a descrição e o ambiente em que pode ser identificado.

4.2 Métodos usados para detecção: RQ1

Atualmente, existem muitas ferramentas que fazem análise estática do código, verificando problemas de sintaxe, como por exemplo JsLint[2] ou Closure Compiler[3]. Contudo, muitos erros são semânticos ou lógicos. Para lidar com erros desta natureza, analisadores semânticos de código estão sendo construídos. Por exemplo, a ferramenta TAJS [11], que verifica tipos, funções/propriedades inexistentes e códigos não utilizados. Um outro exemplo é a ferramenta de análise estática SAFE[4]. Estas ferramentas geralmente criam uma Árvore de Sintaxe Abstrata (em inglês Abstract Syntax Tree – AST) e geram um grafo de fluxo de controle. Além disso, existem ferramentas que fazem análises dinâmica, como DLint [6] e JSNOSE [5], que verificam regras de consistência baseadas em *má práticas de codificação.*

[1]http://lapes.dc.ufscar.br/tools/start_tool
[2]http://www.jslint.com/
[3]https://developers.google.com/closure/compiler/
[4]https://github.com/sukyoung/safe

Table 3: Estudos selecionados nesta RSL

ID	Referência	Título	Ano
C1	[11]	Type Analysis for JavaScript	2009
C2	[14]	Cleanroom: Edit-time error detection with the uniqueness heuristic	2010
C3	[15]	FeedLack detects missing feedback in web applications	2011
C4	[30]	Statically locating web application bugs caused by asynchronous calls	2011
C5	[10]	Modeling the HTML DOM and browser API in static analysis of JavaScript Web Applications	2011
C6	[25]	AutoFLox: An automatic fault localizer for client-side JavaScript	2012
C7	[5]	JSNOSE: Detecting javascript code smells	2013
C8	[19]	Practical static analysis of JavaScript applications in the presence of frameworks and libraries	2013
C9	[28]	JavaScript API misuse detection by using typescript	2014
C10	[13]	Type Refinement for Static Analysis of JavaScript	2014
C11	[9]	Using web corpus statistics for program analysis	2014
C12	[18]	PatBugs: A Pattern-Based Bug Detector for Cross-Platform Mobile Applications	2014
C13	[26]	Vejovis: Suggesting fixes for JavaScript faults	2014
C14	[3]	SAFEWAPI: Web API misuse detector for web applications	2014
C15	[2]	Determinacy in static analysis for jQuery	2014
C16	[27]	Static analysis of JavaScript web applications in the wild via practical DOM modeling	2015
C17	[29]	Improving Precision of JavaScript Program Analysis with an Extended Domain of Intervals	2015
C18	[20]	Static analysis of event-driven Node.js JavaScript applications	2015
J19	[24]	Automatic fault localization for client-side JavaScript	2016

4.2.1 Estudos relacionados com a ferramenta TAJS

Na pesquisa **C1** foi criado o *plug-in* para o IDE Eclipse que analisa programas JavaScript chamado TAJS[5]. É usada a técnica de *interpretação abstrata* e é gerado um grafo do fluxo de dependência das funções JavaScript do arquivo (*".js"*) através do WALA[6].

Em **C5**, foi desenvolvida uma extensão do TAJS [11] que diferente da versão inicial utiliza uma análise de controle. Também utilizam um modelo de *heap* baseado na abstração do local de alocação, que por exemplo, identifica anomalia em funções $ (jQuery) que tem um comportamento diferente, dependendo se é passada uma função, uma *string* HTML, uma *string* CSS ou um elemento DOM.

O estudo **C15** realiza uma análise estática, baseada na investigação sistemática de imprecisão da ferramenta TAJS (desenvolvida em C1). Nessa ferramenta, foi incluído o contexto seletivo e a sensibilidade do caminho, que são variações de ideias bem conhecidas mas, que nunca tinham sido utilizadas para analisar programas JavaScript com jQuery.

4.2.2 Estudos relacionados com a ferramenta Rhino

Em **C3**, foi realizada uma análise estática. Primeiro é feita a identificação do código JavaScript (arquivos *".js"* e em *tags script* no HTML), utilizando o analisador Rhino[7] que gera um conjunto de ASTs que posteriormente são convertidas em grafos de fluxo de controle. Finalmente são criados *clusters*, agrupando as possíveis saídas de cada função. É realizada uma iteração através de todos os caminhos no *cluster* para identificar quais levam a uma sequência crítica.

No estudo **C13**, o processo para detecção começa a partir dos métodos e propriedades do DOM API, tendo uma combinação de análise estática e dinâmica para identificar as linhas de código defeituosas. Os métodos utilizados foram o RHINO, para obter os códigos JavaScript que são executados no software através de CRAWLJAX [21] e um algoritmo que procura por substituições viáveis entre os parâmetros válidos encontrados. As abordagens foram implementadas na ferramenta Vejovis, que também fornece sugestões de correção para as falhas encontradas.

4.2.3 Estudos relacionados com a ferramenta SAFE

Na pesquisa **C9** a ferramenta SAFE foi estendida. Foi utilizado o repositório do GitHub chamado DefinitelyTyped[8], que fornece definições dos tipos (tipagem dos atributos e funções) para TypeScript[9], das mais de 300 bibliotecas de JavaScript frequentemente utilizadas. Desta forma, foi criada uma ferramenta de análise escalável utilizando tipos extraídos da especificação correspondente das bibliotecas em TypeScript. Os arquivos HTML e JavaScript são analisados gerando árvores DOM a partir de arquivos HTML e ASTs para serem comparados com o catalogo de definições *DefinitelyTyped*.

Na pesquisa **C14**, os autores estenderam a ferramenta SAFE propondo o SAFEWAPI. O foco principal, é analisar o uso indevido das APIs Web, cometido pelos desenvolvedores. Utilizaram as especificações das APIs Web, que são disponibilizadas pelos fornecedores dos serviços e são frequentemente declaradas em linguagens descritivas de interfaces (IDLs). Assim, a detecção é realizada em cima destas descrições e está focada em aplicações *mobile*.

No estudo **C16** foi estendida a ferramenta SAFE criando o SAFEWApp. Os autores utilizaram o Jericho[10] para extrair o código JavaScript e seu local de origem do HTML, e em seguida utilizam o CyberNeko[11] para construir uma árvore DOM. O resultado é passado para os módulos DOMModeler e DOMBuilder. O DOMModeler adiciona a um objeto protótipo do DOM, simulando um *browser*. O DOMBuilder traduz uma árvore DOM concreta para uma DOM abstrata. Depois de analisar um determinado programa, são analisados todos os manipuladores de eventos registrados pelo programa, para posteriormente combinar com segurança os resultados da análise.

4.2.4 Outros

No **C2**, foi apresentada uma ferramenta simples chamada

[5]http://www.brics.dk/TAJS
[6]http://wala.sourceforge.net/
[7]http://www.mozilla.org/rhino/
[8]https://github.com/DefinitelyTyped/DefinitelyTyped
[9]https://www.typescriptlang.org/
[10]http://jericho.htmlparser.net
[11]http://nekohtml.sourceforge.net

Table 4: Catálogo de falhas

ID	Nome da falha	Descrição	Ambiente
Bg1	Função inexistente	Chamada realizada para uma função inexistente.	Todos
Bg2	Variável inexistente	Acessar uma variável *null* ou *undefined*.	Todos
Bg3	Propriedade inexistente	Acessar propriedades inexistentes, retornando *null* ou *undefined*).	Todos
Bg4	Variável morta	Variável não está sendo utilizada mais em nenhum lugar.	Todos
Bg5	Número de argumentos inválidos	Acionando funções com um número errado de argumentos.	Todos
Bg6	Tipo de argumento inesperado	Passar para a função, argumentos com tipos distintos do esperado.	Todos
Bg7	Argumento morto	Argumento que não está sendo utilizado em nenhum lugar da função.	Todos
Bg8	Código morto	Trecho de código não utilizado.	Todos
Bg9	Exceção não tratada	Erros que geram exceção e que não são tratadas.	Todos
Bg10	Erro na manipulação do DOM	Acessar objeto DOM inexistente, página ou tipo do objeto inesperado.	*client-side*
Bg11	Funções de eventos inexistente ou inválidas	Erros gerados na interação do usuário com a página web, ocorridos normalmente em eventos que não possuem funções JavaScript ou inválidas.	*client-side*
Bg12	Conversões em tempo de execução	Conversões de tipos primitivos ou não, durante a execução do software.	Todos
Bg13	Evento morto	Evento que não está sendo utilizado em nenhum lugar do software.	Todos
Bg14	Uso errado do *instanceof*	Nenhum objeto usado nos lados do operador do *instanceof*.	Todos
Bg15	Condicional inoperante	Expressões condicionais com valores constantes, de forma a sempre retornar valor verdadeiro ou sempre falso.	Todos
Bg16	Erros em execução assíncrona	Erros que podem ocorrer em execução assíncrona com AJAX [4], causando problemas de execução por ordem inesperada.	Todos
Bg17	Ausência de funções *callback* de erro	Caso ocorra erro durante a execução da função em API, sem função *callback*, não será possível saber o erro que ocorreu.	Todos
Bg18	Elemento DOM inexistente	Propriedade dentro de um elemento DOM inexistente.	*client-side*
Bg19	Modificação do método ou propriedade em tempo de execução	Alterar o método ou a propriedade durante o momento de execução do software.	Todos
Bg20	*Loop* infinito ou oneroso	Laço sem condição de parada ou oneroso.	Todos
Bg21	Código duplicado no projeto	Código repetido dentro do mesmo projeto.	Todos
Bg22	Código duplicado em projetos de terceiros	Código repetido em projetos de terceiros, que são referenciados dentro do projeto.	Todos
Bg23	Retorno morto	Retorno da função não utilizado em nenhum lugar.	Todos
Bg24	Blocos de *catch* vazio	Utilizar *try/catch* sem capturar a lógica do erro.	Todos
Bg25	Objeto grande	Objeto com muitas responsabilidades.	Todos
Bg26	Objeto preguiçoso	Objeto com pouca funcionalidade.	Todos
Bg27	Função longas	Função com muitas responsabilidades.	Todos
Bg28	Lista de parâmetros longa	Função com muitos argumentos.	Todos
Bg29	Instruções *Switch*	Código duplicado e com alta complexidade.	Todos
Bg30	*Closure Smells*	Funções aninhadas com encadeamento de escopo.	Todos
Bg31	Acoplamento entre JavaScript, HTML e CSS	Código JavaScript acoplado com HTML e CSS.	*client-side*
Bg32	Excessivas variáveis globais	Muitas variáveis globais, podendo ter dados inconsistentes.	Todos
Bg33	Cadeia de mensagens longa	Trecho de código que é muito longo e aninhado. Por exemplo, o encadeamento de funções em jQuery.	Todos
Bg34	Retorno de chamada aninhado	Funções aninhadas, retornando para outra função, diretamente nos parâmetros de entrada.	Todos
Bg35	Recusa legado	Altera muitas propriedades do pai, a ponto de não fazer sentido existir a herança.	Todos
Bg36	Erro de tipo numérico	Erros que podem ocorrer com relação ao tipo.	Todos
Bg37	Erro de cálculo numérico	Erros que podem ocorrer no cálculo matemático em JavaScript.	Todos
Bg38	Erro na chamada do método	Instrução de chamada de método que envolve Of, chamada (*MethodCall*).	*Mobile*
Bg39	Erro no retorno	Uma instrução de retorno que retorna Of.	*Mobile*
Bg40	*Assign*	Uma declaração de atribuição que envolve Of.	*Mobile*
Bg41	*NullEqualityIfCheck*	Uma instrução *if* que faz uma verificação de igualdade entre Of e *null*.	*Mobile*
Bg42	*DirectIfNotCheck*	Uma instrução *if* cuja condição Of é diretamente prefixada com o operador de negação, tais como "*if* (!Of)".	*Mobile*
Bg43	Converter *undefined* para número	Converter *undefined* para número, retornando tipo *NaN* (não é um número).	Todos
Bg44	Invocar um valor não funcional	Tentar invocar um objeto ou valor como função.	Todos
Bg45	Variáveis ou propriedades de objeto inacessíveis	Tentar escrever ou acessar variáveis ou propriedades de objeto inacessíveis.	Todos
Bg46	Nomes únicos	Nomes de variáveis, funções ou propriedades, que aparecem apenas uma vez no projeto (pode ser erro).	Todos

Cleanroom que é uma versão modificada do editor Bespin[12]. A ferramenta destaca qualquer nome ou par de nomes que aparece apenas uma vez no programa. São identificados os *tokens* (identificadores únicos de trechos de código) e aplicadas expressões regulares simples. Por fim, é utilizado o cálculo da distância de cordas de Levenshtein [16] entre os *tokens* e a lista que contem todos os trechos de código encontrados no momento da análise.

Em **C4** foi proposto um analisador estático de código inter-procedural para verificar possíveis falhas que podem ocorrer em requisições assíncronas Ajax [4]. Utilizaram Datalog, que é uma notação Prolog-like [17], para fornecer uma representação para a análise de programa, especialmente análise de fluxo. Através da extração de códigos JavaScript, são verificadas as chamadas Ajax, realizando análise em cima das funções encontradas, exceto os corpos (código interno) das bibliotecas de terceiros, pois buscaram apenas abstração para as funções de interface.

No **C6** e no **J19** foram utilizadas a análise dinâmica e a

[12]https://wiki.mozilla.org/Labs/Bespin

técnica *dynamic backward slicing* [1] na ferramenta AutoFlox. Esta ferramenta pode rodar através do CRAWLJAX [21] e como *plug-in* do Eclipse. A abordagem realiza a separação das funções e executa os passos de cada uma delas. São utilizados os marcadores ASYNCCALL, ASYNC e FAILURE/ERROR. O marcador FAILURE/ERROR é adicionado quando um erro é detectado. O ASYNCCALL é colocado após uma chamada assíncrona para uma função e colocado no início da execução da função assíncrona. A função assíncrona contém o nome da função ASYNC bem como o seu identificador. Diferente de C6, o J19 pode localizar falhas em funções *eval*, funções anônimas e código JavaScript minificado.

Em **C7**, foram utilizadas também ASTs aplicadas na ferramenta JSNOSE, que realiza uma análise estática e dinâmica do código. A ferramenta intercepta em alto nível o código JavaScript da aplicação web que vai analisar, com um proxy configurado entre o servidor e o navegador. Depois, é extraído o código JavaScript de todos os arquivos *.js* e HTML, para logo em seguida, realizar a conversão do código em AST. Então, é feita uma análise em cada entidade, objeto, propriedade, função e bloco de código do programa JavaScript. Nesse processo, são coletados dados estáticos e dinâmicos para realizar a comparação com uma lista predefinida de códigos que podem indicar um problema potencial (*code smells*).

No estudo **C8** foi feita a combinação da tradicional *análise de ponteiro* [7] e uma nova análise chamada de *análise de uso*. Um dos principais objetivos deste estudo é conseguir analisar bibliotecas de terceiros dentro do software escrito em JavaScript. A análise proposta é declarativa e expressa como uma coleção de regras de inferência, permitindo fácil manutenção, portabilidade e modificação. Este trabalho faz análise iterativa parcial ou completa. A análise iterativa parcial depende da existência dos *stubs* que são mecanismo de especificação parcial para resolver o problema da biblioteca, a qual foi chamada. A análise iterativa completa independe de quaisquer *stubs* ou especificações de biblioteca. Esta abordagem foi implementada na API WinRT, com uso em aplicativos de Windows 8.

Em **C10** foi proposta a técnica chamada *refinamento de tipo* que trabalha com um domínio abstrato genérico criado com base em tipos. Essa técnica analisa o código e identifica o tipo (por exemplo, *string*, inteiro ou lógico) da variável ou propriedade. A chave deste trabalho é reconhecer que a semântica de JavaScript inclui muitas verificações condicionais implícitas nos tipos e que a execução do refinamento de tipo sobre essas verificações implícitas proporciona um benefício significativo para a precisão da análise.

No estudo **C11**, foram utilizados (i) o modelo N-Gram de frequência de texto, técnica comumente utilizada nos estudos de processamento de linguagem natural; e (ii) tokenização, para extrair as características dos trechos de código a partir do código fonte bruto. As declarações que carregam termos e conjuntos de palavras relevantes da linguagem são decompostas e depois é criado um grafo. Com a estrutura convertida em grafo, é feita uma busca em largura para identificar os códigos duplicados. O método proposto pode ser usado para detecção de plágio e para encontrar falhas por ter código repetido.

O estudo **C12** concentram-se na detecção automática de erros temporais através de padrões para aplicações móveis e multiplataforma durante o desenvolvimento. As verificações são realizadas para detectar falhas provenientes do ambiente *mobile*. Utilizaram uma notação chamada de *Flexible Bug Pattern Specification Notation* que trabalha com autômatos de estados finito para especificar falhas típicas ou cenários de uso incorreto de objetos com padrões de falhas já especificados. Implementaram essa abordagem na ferramenta chamada PatBugs, que é usada em aplicativos *mobile* desenvolvidos em Apache Cordova[13].

No trabalho **C17**, o objetivo foi melhorar o desempenho da ferramenta de análise JSAI [12] que não tinha uma análise de tipos numéricos. Focaram exclusivamente numa abordagem que busca identificar e analisar tipos numéricos chamada de *domínio estendido de intervalos numéricos*, a qual ajudou a fazer uma inferência mais precisa do tipo.

Em **C18**, os autores mostraram que os grafos de chamadas baseadas em eventos (clique, pressionar de teclada, etc.), podem ser usados como base para detectar vários tipos de erros e apresentar uma família de análises. As análises sensíveis ao evento e ao ouvinte (*listener*) calculam fatos dos fluxos de dados separados com base nos eventos que foram emitidos e no que está registrado nos *listeners*, respectivamente. A implementação desta abordagem foi realizada na ferramenta de análise estática, chamada RADAR, para ambientes de *server-side* com Node.js[14].

4.3 Características do método proposto: RQ2

Os estudos selecionados foram organizados na Tabela 5 para responder as questões RQ2-1, RQ2-2, RQ2-3 e RQ2-4.

Entre os tipos de falhas que são mais buscados pelos estudos estão Bg1, Bg2 e Bg3. Foi possível verificar que 52,6% dos estudos tentam detectar a falha Bg2. E 31,6% dos estudos buscam detectar as falhas Bg1 e Bg3.

Em relação ao ambiente para o qual o método foi proposto, 78,9% dos estudos selecionados estão focados em detectar falhas do ambiente *client-side*. Contudo, observa-se estudos, entre eles C8, C12, C14 e C18, que tem o objetivo de detectar falhas em aplicativos, *mobile* e *server-side*.

É importante destacar que os estudos C6 e J19 encontram falhas provenientes de *exceções não tratadas* e sem possuir nenhuma base de falhas predefinida. Além disso, realizam as detecções durante e após o desenvolvimento do software. Desta forma, essa abordagem pode detectar vários tipos de falhas que não foram previamente catalogadas, como erros que ocorrem por causa de dados ou por interação do usuário com o software.

4.4 Características dos experimentos: RQ3

As respostas para as questões RQ3-1, RQ3-2, RQ3-3, RQ3-4 e RQ3-5 para cada um dos estudos selecionados são mostradas na Tabela 6. Na tabela, NI significa *Não Informado*.

Os sistemas de software que mais foram utilizados nos experimentos são as aplicações web obtidas em repositórios públicos de código fonte. Note que, a maioria de estudos não faz comparação com outras abordagens nos experimentos. A quantidade de projetos ou aplicações usadas nos experimentos variam entre 1 e 100.000. O estudo que obteve maior precisão e cobertura foi o C7, sendo que o menor tempo usado para encontrar as falhas foi 5,1s do estudo C12.

5. CONCLUSÕES

[13]https://cordova.apache.org/
[14]https://nodejs.org/

Table 5: Características do método proposto dos estudos selecionados nesta RSL

ID	Tipo de falhas	Base predefinida de falhas	*Client-side*, *server-side*, *mobile*, aplicativos	Durante ou após desenvolvimento
C1	Bg2, Bg3, Bg5, Bg8, Bg12, Bg43, Bg44, Bg45	Sim	*client-side*	Durante
C2	Bg2, Bg46	Sim	*client-side*	Durante
C3	Bg11	Sim	*client-side*	Durante
C4	Bg16	Não	*client-side*	Durante
C5	Bg2, Bg3, Bg8, Bg45	Sim	*client-side*	Ambos
C6	Bg9	Não	client-side	Ambos
C7	Bg8, Bg24, Bg25, Bg26, Bg27, Bg28, Bg29, Bg8, Bg30, Bg31, Bg32, Bg33, Bg34, Bg35	Sim	*client-side*	Durante
C8	Bg2, Bg7, Bg8, Bg23	Não	Aplicativos Windows 8	Ambos
C9	Bg1, Bg2, Bg3, Bg5, Bg6,	Sim	*client-side*	Durante
C10	Bg1, Bg2, Bg12	Sim	*client-side*	Após
C11	Bg21, Bg22	Não	*client-side*	Durante
C12	Bg38, Bg39, Bg40, Bg41, Bg42	Sim	*mobile*	Durante
C13	Bg4, Bg5, Bg10	Sim	*client-side*	Ambos
C14	Bg1, Bg2, Bg3, Bg5, Bg6	Sim	*mobile*	Após
C15	Bg6, Bg8, Bg20	Não	*client-side*	Após
C16	Bg2, Bg3, Bg14, Bg15, Bg43, Bg44	Não	*client-side*	Após
C17	Bg1, Bg2, Bg3, Bg36, Bg37	Sim	*client-side*	Após
C18	Bg1, Bg2, Bg3, Bg13	Sim	*server-side*	Durante
J19	Bg9	Não	*client-side*	Ambos

Table 6: Características dos experimentos realizados nos estudos selecionados nesta RSL.

ID	Características dos softwares	Repositório	Comparação com	Nº de projetos ou aplicações	Precisão	Cobertura	Tempo
C1	Projetos JavaScript	Público	Nenhum	7 projetos	NI	NI	<6min
C2	Aplicações Web	Público	Nenhum	1 projeto	NI	NI	NI
C3	Aplicações Web	Público	Nenhum	330 aplicações	18%	NI	NI
C4	Ajax nas aplicações WEB	Público	Nenhum	8 aplicações	NI	NI	<13s
C5	Aplicações WEB sem bibliotecas	Público	Nenhum	53 aplicações web JavaScript	90%	NI	NI
C6	Aplicações WEB	Ambos	Nenhum	9 projetos e 7 aplicações web	NI	NI	NI
C7	Aplicações WEB	Público	Nenhum	11 aplicações web	93%	98%	NI
C8	Aplicativos JavaScript do Windows 8	Privado	Ferramentas como Eclipse, IntelliJ , Visual Studio	25 projetos	81,5%	NI	<10s
C9	Projeto de TypeScript	Público	Nenhum	NI	NI	NI	NI
C10	Frameworks e software JS	Público	Nenhum	8 aplicações JavaScript	86%	NI	NI
C11	Aplicações de JavaScript	Público	Nenhum	100.000 projetos	23,9%	NI	NI
C12	Aplicações *Mobile*	Privado	Nenhum	3 projetos	84%	NI	<5,1s
C13	Aplicações Web	Público	Nenhum	11 projetos	91%	73%	<91,1s
C14	Aplicações JavaSript de TV Inteligente e para Android	Privado	Nenhum	43 aplicações	NI	NI	NI
C15	Aplicações Web com jQuery	Público	Nenhum	154 programas	53%	NI	<24s
C16	Aplicações Web e bibliotecas JavaScript	Público	TAJS com Gatekeeper	5 bibliotecas de JavaScript e 10 *sites*	75,5%	NI	<3h
C17	Sistemas de software JavaScript	Público	Nenhum	19 programas JavaScript	NI	NI	NI
C18	Aplicações Node.js	Público	Nenhum	6 programas	NI	NI	NI
J19	Aplicações Web	Público	Nenhum	3 projetos	NI	NI	<33,2s

Esta revisão sistemática da literatura investigou e avaliou 19 estudos primários relacionados a detecção automática de falhas em JavaScript, visando buscar respostas para as questões de pesquisa. Podemos observar que o método mais utilizado entre os estudos para fazer as análises é criar ASTs. Contudo, alguns pesquisadores optam por estender ferramentas existentes, como ocorreu com os trabalhos que estenderam as ferramentas de análise SAFE e TAJS.

A maioria dos estudos estão voltados para ambiente Web (*client-side*), contudo poucas pesquisas focaram em detectar falhas em bibliotecas de terceiros (*plug-ins*). O que abre espaço para que pesquisas se desenvolvam nessa linha, pois hoje em dia, muitos desenvolvedores buscam os *plug-ins*, como o jQuery, para facilitar o desenvolvimento. O mesmo ocorre nos estudos para software de ambientes multiplataforma, principalmente, em *server-side*, pois na nossa revisão existem apenas quatro estudos focados na detecção de falhas para esses ambientes.

A grande maioria dos estudos utilizaram repositório de código fonte público, como o GitHub, sendo essa uma ótima fonte de dados para estudos na área. Uma vez que JavaScript é uma linguagem que está em constante desenvolvimento, é importante que as ferramentas de detecção de falhas não estejam engessadas com o uso de um catálogo predefinido de falhas. Além disso, é importante que façam a análise durante e após o desenvolvimento.

6. REFERENCES

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Sixth International Symposium on Software Reliability Engineering*, pages 143–151. IEEE, 1995.

[2] E. Andreasen and A. Møller. Determinacy in static

analysis for jQuery. In *ACM SIGPLAN Notices*, volume 49, pages 17–31. ACM, 2014.

[3] S. Bae, H. Cho, I. Lim, and S. Ryu. SAFEWAPI: Web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 507–517. ACM, 2014.

[4] M. Eernisse. *Build Your Own AJAX Web Applications*. SitePoint Pty Ltd, 2006.

[5] A. M. Fard and A. Mesbah. JSNOSE: Detecting JavaScript code smells. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM-2013)*, pages 116–125, 2013.

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

[7] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.

[8] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.

[9] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *ACM SIGPLAN Notices*, volume 49, pages 49–65. ACM, 2014.

[10] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[11] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[12] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132, 2014.

[13] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of JavaScript. In *ACM SIGPLAN Notices*, volume 49, pages 17–26. ACM, 2013.

[14] A. J. Ko and J. O. Wobbrock. Cleanroom: Edit-time error detection with the uniqueness heuristic. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 7–14. IEEE, 2010.

[15] A. J. Ko and X. Zhang. Feedlack detects missing feedback in web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2177–2186. ACM, 2011.

[16] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[17] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-answering algorithms for information agents. In *Thirteenth National Conference on Artificial Intelligence*, pages 40–47. AAAI, 1996.

[18] G. Liang, J. Wang, S. Li, and R. Chang. Patbugs: A pattern-based bug detector for cross-platform mobile applications. In *IEEE International Conference on Mobile Services (MS-2014)*, pages 84–91. IEEE, 2014.

[19] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[20] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node. js JavaScript applications. In *ACM SIGPLAN Notices*, volume 50, pages 505–519, 2015.

[21] A. Mesbah, A. Van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[22] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64. IEEE, 2013.

[23] F. S. Ocariza. *On the detection, localization and repair of client-side JavaScript faults*. PhD thesis, University of British Columbia, 2016.

[24] F. S. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability*, 26(1):69–88, 2016.

[25] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah. Autoflox: An automatic fault localizer for client-side JavaScript. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST-2012)*, pages 31–40. IEEE, 2012.

[26] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847. ACM, 2014.

[27] C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of JavaScript web applications in the wild via practical DOM modeling (t). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE-2015)*, pages 552–562. IEEE, 2015.

[28] J. Park. JavaScript API misuse detection by using typescript. In *Proceedings of the companion publication of the 13th international conference on Modularity*, pages 11–12. ACM, 2014.

[29] A. Younang and L. Lu. Improving precision of Java Script program analysis with an extended domain of intervals. In *IEEE 39th Annual Computer Software and Applications Conference (COMPSAC-2015)*, volume 3, pages 441–446. IEEE, 2015.

[30] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web*, pages 805–814. ACM, 2011.