Pós-Graduação em Ciência da Computação

Igor Simões de Oliveira Lima

Leveraging Diversity to Find Bugs in JavaScript Engines

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2020

Igor Simões de Oliveira Lima

Leveraging Diversity to Find Bugs in JavaScript Engines

A M.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering, Software Testing
Advisor: Marcelo Bezerra d'Amorim

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

L732l Lima, Igor Simões de Oliveira
Leveraging diversity to find bugs in JavaScript engines / Igor Simões de Oliveira Lima. – 2020.
53 f.: il., fig., tab.

Orientador: Marcelo Bezerra d'Amorim.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de software. 2. JavaScript. I. d'Amorim, Marcelo Bezerra (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 124

Igor Simões de Oliveira Lima

“Leveraging Diversity to Find Bugs in JavaScript Engines”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 17 de janeiro de 2020.

BANCA EXAMINADORA

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática/UFPE

_____

Prof. Dr. Igor Scaliante Wiese
Departamento Acadêmico de Ciência da Computação/UTFPR

_____

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática/UFPE
(**Orientador)**

Dedico este trabalho à minha família e amigos que foram meus alicerces perante as dificuldades deste percurso.

ACKNOWLEDGEMENTS

Primeiramente, agradeço a minha família que mesmo estando longe, me apoiaram em todas as decisões que tomei na vida. Até hoje me recordo de quando saí de casa para estudar em outra cidade e vocês sempre me incentivando para buscar novos horizontes. De 2010 até hoje, passei pelas cidades de Aracaju/SE, Arapiraca/AL e Recife/PE. Eu não estaria aqui se não fosse por todo sacrifício que vocês fizeram por mim. Obrigado.

Computação sempre foi uma paixão, desde o acesso ao primeiro computador até a implementação do primeiro programa. Fiz o curso de Ciência da Computação da Universidade Federal de Alagoas campus Arapiraca, ao qual tive excelente professores e desde então meu gosto pela área só aumentou. Deixo meus agradecimentos aos professores Mário Hozano e Alexandre Barbosa pela iniciação científica no LAMP (Laboratório de Análise, Modelagem e Programação).

Em 2016, fui selecionado para a 18ª turma de residência em análise de testes de software do projeto Motorola/CIn-UFPE, ao qual me incentivou a seguir a área de testes de software. Fiz grandes amigos durante esse percurso, em especial, o pessoal do laboratório de automação ao qual aprendi muito com eles e todo pessoal da minha turma, que se tornaram meus melhores amigos aqui em Recife.

Com a residência em testes, me interessei pela área e decidi fazer o mestrado em Ciência da Computação do Centro de Informática. De todo lugar que já passei, o CIn/UFPE será sempre minha referência de melhor instituição. Fiz grandes amigos por aqui, em especial, meus irmãos acadêmicos Luis Melo, Jeanderson Cândido, Sotero Júnior e Davino, só a gente sabe o que o laboratório do INES nos representou nestes anos. Agradeço também a meus amigos da pós e de apartamento, Raphael Dourado, Francisco Cabeça, Ramon Maciel e Edton Lemos.

E por fim, agradeço ao meu orientador Marcelo d'Amorim por acreditar em meu trabalho, estar sempre presente e por incentivar o gosto pela pesquisa. Graças ao nosso esforço, tivemos até o momento dois papers aceitos, o que para mim, era meu objetivo inicial ao entrar no programa e me sinto honrado em contribuir de alguma forma para a pesquisa acadêmica.

Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better (BECKETT, 1983).

ABSTRACT

JavaScript is a very popular programming language today with several implementations competing for market dominance. Although a specification document and a conformance test suite exist to guide engine development, bugs occur and have important practical consequences. This work evaluates the importance of different techniques to find functional bugs in JavaScript engines. For that, we explored two existing techniques—test transplantation and cross-engine differential testing. The first technique runs test suites of a given engine in another engine. The second technique fuzzes existing inputs and then compares the output produced by different engines with a differential oracle. We considered engines from four major players in our experiments–V8, SpiderMonkey, ChakraCore, and JavaScriptCore. We present a tool capable of running tests on any javascript engine and obtaining reports based on the test output. It was possible to run the four engines in a test suite extracted from open-source projects, using the two techniques mentioned and we analyzed the behavior of each engine, classifying the output as a bug or not. The results indicate that both techniques revealed several bugs, many of which confirmed by developers. Overall, we reported 50 bugs in this study. Of which, 36 were confirmed by developers and 29 were fixed. To sum, our results show that the techniques are easy to apply and very effective in finding bugs in complex software, such as JavaScript engines.

Keywords: Diversity. Test Transplantation. Differential Testing. JavaScript.

RESUMO

Atualmente, o JavaScript é uma linguagem de programação muito popular, com várias implementações competindo pelo domínio do mercado. Embora exista um documento de especificação e um conjunto de testes de conformidade para orientar o desenvolvimento do motor (do inglês, *engine*), bugs ocorrem e têm importantes consequências práticas. Este trabalho avalia a importância do uso de diferentes técnicas para encontrar erros funcionais nos motores JavaScript. Para isso, exploramos duas técnicas de testes existentes - teste de transplante e teste diferencial entre motores. A primeira técnica executa suítes de teste de um determinado mecanismo em outro mecanismo. A segunda técnica aplica fuzzing nas entradas de teste e depois compara o resultado produzido em diferentes motores através de um oráculo diferencial. Consideramos os quatro principais motores da atualidade em nossos experimentos - V8, SpiderMonkey, ChakraCore e JavaScriptCore. Apresentamos uma ferramenta capaz de executar testes em qualquer motor javascript e obter relatórios baseado na saída dos testes. Com esta ferramenta, foi possível executar os quatro motores em uma suíte de testes extraídos de projetos open-source, utilizando as duas técnicas citadas e analisamos o comportamento de cada motor, classificando a saída como um bug ou não. Os resultados indicam que ambas as técnicas revelaram vários bugs, muitos dos quais já foram confirmados pelos desenvolvedores. No geral, relatamos 50 bugs neste estudo. Dos quais, 36 foram confirmados pelos desenvolvedores e 29 foram corrigidos. Em resumo, nossos resultados mostram que as técnicas são fáceis de aplicar e são muito eficazes para encontrar bugs em softwares complexos, como motores JavaScript.

Palavras-chaves: Diversidade. Teste de Transplante. Teste Diferencial. JavaScript.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

DT	Differential Testing
FP	False Positives
JS	JavaScript
NaN	Not a Number
TP	True Positives
TT	Test Transplantation

CONTENTS

1 INTRODUCTION

JavaScript (JS) is one of the most popular programming languages today (Stackify, 2018; RedMonk, 2018), which is used in various software development segments including, web, mobile, and, more recently, the Internet of Things (IoT) (Simply Technologies, 2018) and Machine Learning (ELLIOTT, 2019). The interest of the community for the language encourages constant improvements in its specification (TC39, 2018c). It is natural to expect that such improvements lead to sensible changes in engine implementations (Kangax, 2018). Even small changes can have high practical impact. For example, in October 2014 a new attribute added to Array objects resulted in the MS Outlook Calendar web app, which is built in JS, to fail under Chrome (Chromium, 2015; ESDiscuss, 2014).

JS engines are virtual machines that parse source code, compile it in bytecodes, and run these bytecodes. These engines implement some version of the ECMAScript (ES), which emerged with the goal to standardize variants of the language, such as Netscape's JavaScript and Microsoft's JScript[1]. There are several implementations of engines by the community, the popular engines are embedded in browsers (e.g. Google Chrome and Mozilla Firefox), but engines also provides a runtime environment to run JS everywhere (e.g. Node.js runtime [2]).

An organization exists to regularize the JS language specifications. The ES specification is regulated by Ecma International (Ecma Internacional, b) under the TC39 (TC39, b) technical committee. Every year, a new version of the ES specification is produced with new features and minor fixes. The canonical spec today is ES6 (TC39, 2018b; TC39, 2018c).

Finding functional bugs in JS engines is an important problem given the range of applications that could be affected with those bugs. It is also challenging due the specifications are intentionally incomplete as to enable development flexibility. In addition, they evolve frequently (see Figure 1) to accommodate the pressing demands from developers (TC39, 2018b). The Figure 1 shows the number of small, medium and large features over the years, the ES6 introduced hundreds of modifications and until today some features are not implemented yet, for example, the tail call optimization (Kangax, 2018). An official conformance test suite exists for JS (TC39, b), but, naturally, many test scenarios are not covered in the suite. In addition, we noticed that a significant fraction (5 to 15%) of the tests fail regularly in the most popular engines, reflecting the struggle of developers in keeping up with the pace of spec evolution (see Table 3).

This work, which is empirical in nature, reports on a study to evaluate the importance of finding bugs in JS engines; it covers two complementary diversity-aware testing techniques.

[1] The name JavaScript still prevails today, certainly for historical reasons.
[2] The official NodeJS website. Available at <https://nodejs.org>

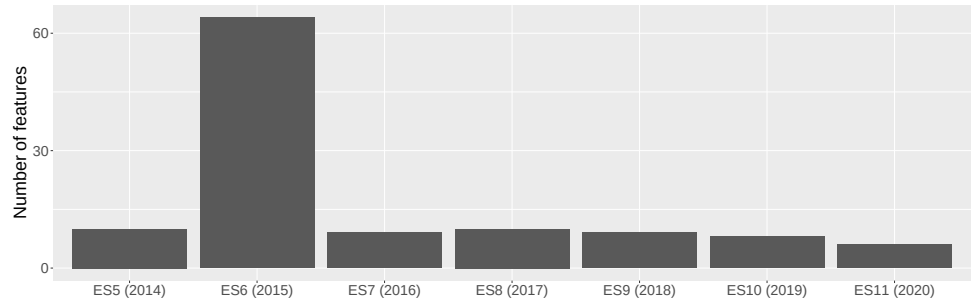


Figure 1 – Evolution of ECMAScript Specifications

- *Test Transplantation (TT)* leverages diversity of test cases. This technique evaluates the effect of running test files written for a given engine in other engines. The intuition is that developers design test cases with different objectives in mind. As such, replaying these tests in different engines could reveal unanticipated problems.
- *Cross-engine Differential Testing (DT)* leverages diversity of engine implementations. This technique fuzzes existing test inputs [3] and then compares the output produced by different engines.The intuition is that interesting inputs can be created from existing inputs and multiple engines can be used to address the lack of oracles.

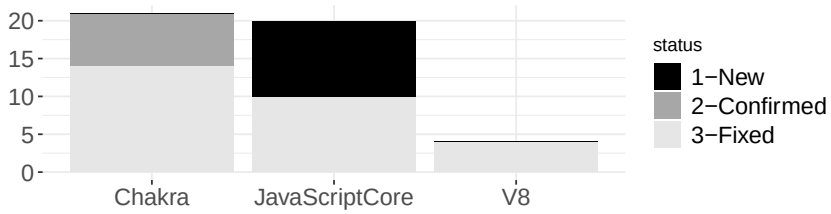
The study measures the ability of these techniques in finding bugs and the impact of False Positives (FP) on their practicality.

In addiction, we propose a tool to run suites of JS test cases to generate alarms based on outputs from JS engines. It is possible to integrate any JS engine and pure JS test files. The test cases are extracted from open source repositories and bug trackers. For the files in bug trackers, we obtain the attached files and we also selected JS code from bug trackers comments through a classifier implemented as an external component, responsible for extracting JS code in comments.

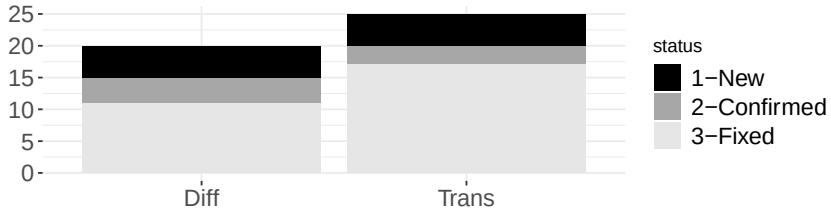
1.1 RELATED IDEAS

The idea of test set diversity dates back to the eighties (WHITE; COHEN, 1980; OSTRAND; BALCER, 1988). In contrast to prior work on this topic, this study explores diversity of implementations and diversity of sources of test cases as opposed to diversity of the test cases themselves. Section 7.1 elaborates related work on this topic. DT (BRUMLEY et al., 2007) has been applied in a variety of contexts to find bugs (YANG et al., 2011a; CHEN; SU, 2015; ARGYROS et al., 2016; CHEN et al., 2016; PETSIOS et al., 2017; SIVAKORN et al., 2017; ZHANG; KIM, 2017). It has shown to be specially practical in scenarios where the observation of difference gives a strong signal of a real problem. For example, Mozilla runs JS files against different configurations of a given build of their SpiderMonkey engine (e.g.,

[3] Fuzz Testing of Application Reliability. Available at <http://pages.cs.wisc.edu/~bart/fuzz/>



(a) Bug reports per engine.



(b) Bugs reports per technique.

Figure 2 – Summary of bug reports.

trying to enable or not eager JIT compilation[4]). A positive aspect of the approach is that it can be fully automated—as only one engine is used, the outcomes of the test in both configurations are expected to be identical. The Mozilla team uses this approach since 2002; they have been able to find over 270 bugs since then (Mozilla, a), including security bugs. Cross-engine differential testing, in contrast, has not been widely popularized. One possible reason is that it is still not practical to fully automate the technique due to the distinct configurations of each engine, as well as a universal oracle that would indicate possible true positives, this tasks still requires human inspection. In contrast to other applications of differential testing, a number of legitimate reasons exist, other than a bug, for a test execution to manifest discrepancy (see Tables 5 and 8). To sum up, variations of these ideas have been explored before in different contexts. The goal of the study is to assess the ability of the techniques aforementioned in finding bugs on JavaScript engines.

1.2 RESEARCH METHODOLOGY

The main purpose of this study is to finding and reporting functional bugs in JS engines. We conducted our study in JS engines, empirical in nature, to investigate the stability of the popular engines. We used three methods to reach the results described as Conformance Test, Test Transplantation and Differential Testing. In the first dimension we ran the conformance suite provided by the technical committee across the engines that we selected according the criteria (described in Section 2.2) to observe how the popular engines are stable compared to the current specification. In this case, we did not analyze the reports because these test cases are the basic requirements for a JS engine development and the development team is aware of the problems. The Test Transplantation and

[4] These files are created with the grammar-based fuzzer jsfunfuzz. Look for option “compare_jit” from funfuzz.

Differential Testing are similar, in nature. We ran the tests, compare the output of them and manually analyzed the results. The analysis is basically a manual log inspection made by experience developers (e.g. Test Transplantation), students in exploratory inspections and the authors with a guideline (e.g. Differential Testing). We used standard metrics to determine the effectiveness of the testing techniques, for example, the number of bugs confirmed, fixed and their severity.

1.3 RESULTS

We considered the following engines–Chakra (Microsoft), JavaScriptCore (Apple), V8 (Google), and SpiderMonkey (Mozilla). Figure 2 shows the breakdown of bug reports per engine (2a) and per technique (2b). Each stacked bar breaks down the bugs per status (e.g., "1-New"). The prefix number indicates the ordering that status labels are assigned. Several of these reports have the label "3-Fixed", indicating that bug fixes have been incorporated into the code already. Note that most of these bugs affected two engines–Chakra and JavaScriptCore (JSC). We reported five bugs in V8 (four confirmed) and none in SpiderMonkey. Furthermore, our results show that both techniques revealed several bugs, most of which confirmed by developers. Test transplantation revealed 28 bugs (of which, 21 were confirmed and 18 were fixed) whereas differential testing revealed 22 bugs (of which, 15 were confirmed and 11 were fixed). Overall, results indicate that both techniques were successful at finding bugs. The number of bug reports were similar, if we consider only those confirmed or fixed. Most bugs we found are of moderate severity because these violations are not involving engine crashes or security bugs.

1.4 KEY FINDINGS

The list of findings of this work includes–1) Not only multiple different implementations can be leveraged in differential testing, but differences in test suites can also be important. 2) Even for problems with fairly clear specifications, as in JavaScript, there is likely to be (a lot of) variation between different implementations. 3) Differential testing is feasible on real, complex, widely used pieces of software. The Chapter 5 expands and elaborates key findings and lessons learned.

1.5 CONTRIBUTIONS

The most important contribution of this work is empirical: we provide a comprehensive study analyzing the effectiveness of diversity-aware techniques to find functional bugs in popular JavaScript engines. Additional contributions are: 1) A number of bugs found and fixed. We reported a total of 50 bugs. Of these, 36 bugs were confirmed and 29 bugs were

fixed. 2) An infrastructure for diversity-aware testing. The scripts to run the experiments and the data are publicly available on our repository [5].

To summarize, this study provides initial, yet strong evidence that exploring diversity should be encouraged to find functional bugs in JS engines.

[5] JSEngines repository. Available at <https://github.com/damorimRG/jsengines-differential-testing>

2 BACKGROUND

2.1 JAVASCRIPT

The specification of JavaScript is incomplete for different reasons. Certain parts of the specification are undefined; it is responsibility of engineers to decide how to implement these functionalities. The JavaScript spec uses the label "implementation-dependent" to indicate these cases, where behavior may differ from engine to engine. One reason this flexibility in the spec exists is to enable compiler optimizations. For example, the JS `for-in` loop construct does not clearly specify the iteration order of elements (StackOverflow community, 2018; John Resig, 2018) and different engines capitalize on that for loop optimizations (Camilo Bruni–V8 engineer, 2018). As another example, the specification states that if the `Number.toPrecision()` function is called with multiple arguments then the floating-point approximation is implementation-dependent (Ecma Internacional, c). Various other cases like these exist in the specification. Given the speed the specification changes and the complexity of the language some features are not fully implemented as can be observed by the Kangax compatibility table (Kangax, 2018). It is also worth noting that, as in other languages, some elements in JS have non-deterministic behavior (e.g., `Math.random` and `Date`). A test that make decisions based on these elements could, in principle, produce different outcomes on different runs. Carefully-written test cases should not manifest this kind of flaky behavior. As previously mentioned, all those aspects make testing JS engines challenging.

2.2 ENGINES STUDIED

We selected JS engines according to the following criteria: 1) Released latest version after Jan 1, 2018, 2) Contains more than 1K stars on GitHub, and 3) Uses a public issue tracker. We looked for highly-maintained (as per the first criterion) and popular (as per the second criterion) engines. As we wanted to report bugs, we also looked for project with public issue trackers. Table 1 lists the engines we analyzed. It is worth noting that we used Google Chrome Lab's JSVU (Javascript Version Updater) tool [1] to automatically install and configure versions of different JS engines in our host environment. This is important as we aim to use the most recent stable versions of each engine as to avoid reporting old and already-fixed bugs to developers.

[1] JSVU tool. Available at <https://github.com/GoogleChromeLabs/jsvu>

Table 1 – Engines selected.

Team	Name	URL	# Stars	DOB
Apple	JSC (WebKit)	WebKit	3300+	Jun 2001
Google	V8	Chromium	9800+	Jun 2008
Microsoft	Chakra	Microsoft	7200+	Nov 2009
Mozilla	SpiderMonkey	Mozilla	1100+	Mar 1996

2.3 MINED JS FILES

A good test set is critical for finding bugs with the techniques used in this work. For that reason, we looked for JS files from various sources: 1) test files from the Test262 (TC39, b) conformance suite of the ECMA262 specification (TC39, 2018c), 2) test files from the test suite of the selected engines (see Section 2.2); these files are accessible from the engine's official repositories, 3) test files from the suites of public engines, and 4) test files mined from issue trackers of these engines.

Table 2 shows the breakdown of tests per engine. Column "full" shows the number of test cases associated with each engine. Column "pass-in-par." shows the number of test cases that pass in their parent engine. We discarded tests that fail in their parent engine as we could not reliably indicate the reason for the failure, so we assumed the test could be broken. We removed 63 test cases that fail for that reason–6 tests from JSC and 57 tests from SpiderMonkey. Column "type-in-all" shows the number of test cases whose executions do not throw dynamic type errors in any of the engines because of an undefined variable or property. These cases were captured by looking for the presence of `ReferenceError` and `TypeError` in the output. A `ReferenceError` (respectively, `TypeError`) is raised when test execution attempts to access an undefined variable (respectively, property of an object). We discarded those tests to avoid noise in the experiments as they clearly indicate some missing feature as opposed to bugs. For example, some tests use non-portable names (e.g., JSC's `drainMicrotasks()` and SpiderMonkey's `Error.lineNumber`) or use functions that, albeit part of the spec, not all engines currently support. For the evaluation of test transplantation, we used the 6,602 tests included in the dashed rectangle under column "type-in-all", i.e., all tests under that column but the Test262 tests. We did not consider tests from the conformance suite as they are more likely to indicate missing features as opposed to bugs. In addition, engine developers have access to these test and are encouraged to run them. Finally, column "no-fail-in-all" shows the tests for which all engines pass. Note that the set of tests in this column is a subset of the "type-in-all" test set. This test set is used in the evaluation of differential testing as fuzzing seeds. The guarantee that they pass in all engines assures that, if discrepancies occur, they are related to the changes in the input as opposed to the original cause of failure.

[Cleansing] We noticed that some of the tests we found depend on external libraries,

Table 2 – Number of test files.

Name	Source	# JS files			
		total	pass-in-par.	type-in-all	no-fail-in-all
Test262	GitHub	31,276	-	29,846	17,639
JSC	GitHub	1,265	1,130	1,122	1,054
SpiderMonkey	GitHub	3,122	2,155	2,103	1,837
V8	GitHub	1,084	482	478	426
Duktape	GitHub	1,195	1,195	921	915
JerryScript	GitHub	1,951	1,951	1,878	1,837
JSI	GitHub	99	99	63	63
Tiny-js	GitHub	49	49	37	37
		40,041	7,061	36,448	23,808

which not all selected engines support. We decided to discard those. For example, many tests we found required a Node.js runtime for execution. Also, we did not consider tests from the Chakra repository because they depend on non-portable objects. Finally, note that the number of tests in V8 is low; that happens because V8 uses many tests from Mozilla and JSC; we discarded those to avoid repetition and to give credit where it is due. [Test Harness] It is also worth mentioning that some engines use a custom shell to run tests, including a harness with specific assertions. For that, we needed to make minor changes in the testing infrastructure to be able to run the tests uniformly across all engines. More precisely, we needed to mock non-portable harness functions, which are only available in certain engines.

2.4 MINING TESTS FROM ISSUE TRACKERS

We observed that issue trackers are an important source of test data and should not be ignored. Analyzing a sample of issues, we observed that developers either 1) add test cases as attachments of issues or 2) embed test cases within the textual description of an issue. The test cases in attachments are longer compared to the test cases embedded in issue descriptions whereas the latter are more common. Consequently, we thought we should handle both cases.

To obtain test files included as attachments, we wrote a crawler to visit the issue trackers of V8 and SpiderMonkey listed in Table 1 and we were able to retrieve a total of 490 files. To mine tests from the textual descriptions we implemented a separated component and proceeded as follows. First, we broke the text describing the issue in paragraphs and used a binary classifier to label each paragraph as “code” or “not code” (i.e., natural language). Then, based on that information, we merged consecutive paragraphs labeled as “code” and used a JS parser to check well-formedness of the retrieved code fragment.

Using that method we were able to retrieve a total of 1,240 files. All those files were included in Table 2.

For the classification of "code" vs. "not code", we used a pipeline of two Neural Networks (NN), a popular design for solving NLP classification problems (KUSNER et al., 2015). The first net in the pipeline takes as input an arbitrary sentence and produces on output a characterization vector for that sentence. The second net in the pipeline takes that vector on input and produces a yes/no answer, determining whether or not the input sentence was code. During the training phase, the second neural net in the pipeline takes additionally on input the value for the class attribute for the sample; in this case, "code" or "not code".

More in detail, the Figure 3 illustrates how the model was created. For the first net, we used word2vec (MIKOLOV et al., 2013), a popular NLP technique to produce word embeddings. A word embedding is a mapping of words to vectors of real numbers. We chose the Word2Vec technique to extract context information of each JS token or english word. Since the dictionary of a programming language is shorten than english natural language, we could identify similarity between JS tokens thought the distance vector in the embedding matrix maximizing the likelihood that words are predicted from their context (LING et al., 2015). For the second net, we used a multi-layer perceptron (RUMELHART; HINTON; WILLIAMS, 1986) to infer the probability of the input belonging or not to the class. The classifier labels the input as code if the predicted probability of the input being code is 0.7 or higher. We used a corpus with 25K samples of English paragraphs and 25K snippets of JS code to train and test the classifier and obtained an accuracy of 98%.

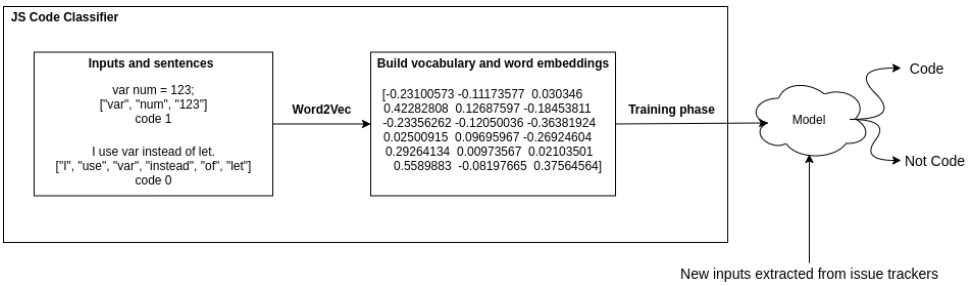
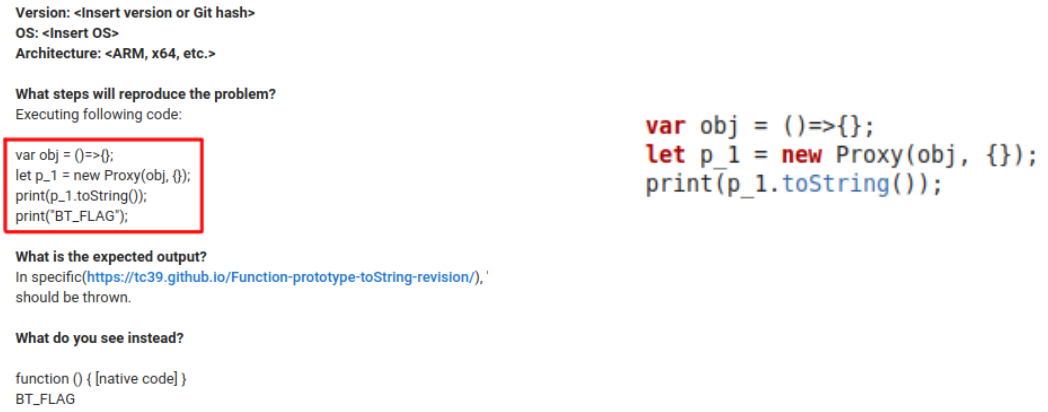


Figure 3 – JavaScript Code Classifier

The Figure 4 shows a case of a valid js file extracted from v8 bug tracker. The Figure 4a shows a bug that violates the specification (under revision) of the function *toString* that should throws a TypeError if the object is not acceptable.

The bug tracker mining works as following: a) It converts the text of the first comment in the issue to a list of sentences. b) For each item in the list, we used the model described in this section to verify if the sentence is a code or not; c) The sentences classified as code is added in a JS file (see Figure 4b). Notice the extracted code is missing the line `print("BT_FLAG")`, it occurs because the `print` function is not a standard function and our model was trained with the `console.log` to get the outputs. Although the



(a) Bug reported in V8 bug tracker (b) Test file extracted from bug tracker

Figure 4 – Example of a mined test using the classifier.

`print(p_1.toString());` was classified as a code because the `toString` method is a common type conversion function and our dataset contains several snippets of codes involving this function. However, this issue[2] was defined as a WontFix status due the specification violated was a older revision, a V8's developer confirmed that the V8 supports the correct specification.

This classifier is publicly available from our repository as a separate component at <https://github.com/damorimRG/jsengines-differential-testing>.

[2] V8 Bug#8109. Available at <https://bugs.chromium.org/p/v8/issues/detail?id=8109>

3 OBJECTS OF ANALYSIS

3.1 INFRASTRUCTURE

This section describes the infrastructure that we used in our experiments. The Figure 5 illustrates the workflow of the infrastructure. In general, we have three main components: the JS files, the engines and the fuzzers. In the first step, we mine the test files (see Section 2.3) to obtain the suites of testing, in this case each suite is a set of JS files extracted from a source. In the next step, we need to download the engines and make it runnable by command-line to run a test as a parameter (e.g. `./v8 test.js`). As described in Section 2.2, we used JSVU tool to download the binaries and keep updating the engines. It is possible to run the testing files in two configurations, running the original files and running with fuzzing (e.g. radamsa and quickfuzz). We described the fuzzing ecosystem in Section 3.1.3. We used the first configuration to run the conformance test and the transplantation testing across engines, the other configuration we used for cross-engine differential testing.

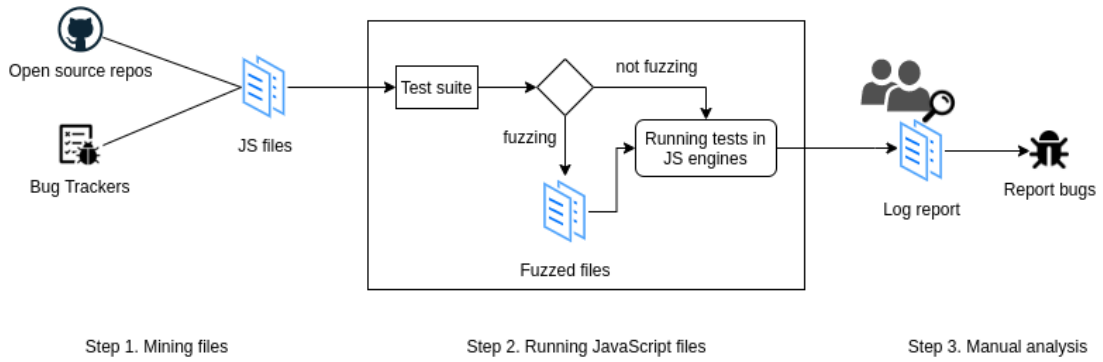


Figure 5 – Overview of the infrastructure.

Figure 6 illustrates the workflow of the approach of the step 2 and step 3. It takes on input a list of JS files and generates warnings on output. Numbered boxes in the figure denote the data processors and arrowed lines denote data flows. The cycle icons indicate repetition–the leftmost icon indicates that each file in the input list will be analyzed in separate whereas the rightmost icon shows that a single file will be fuzzed multiple times.

The bug-finding process works as follows. For a given test input, the toolchain produces new inputs using some off-the-shelf input fuzzer (step 1). (Section 3.1.3 describes the fuzzers we selected.) Then, the oracle checks whether or not the output produced for the fuzzed file is consistent across all engines (step 2). In case the test passes in all engines or fails in all engines (i.e., the output is consistent), the infrastructure ignores the input. Otherwise, it considers the input as potentially fault-revealing; hence, interesting for human inspection. Finally, to facilitate the human inspection process, the infrastructure

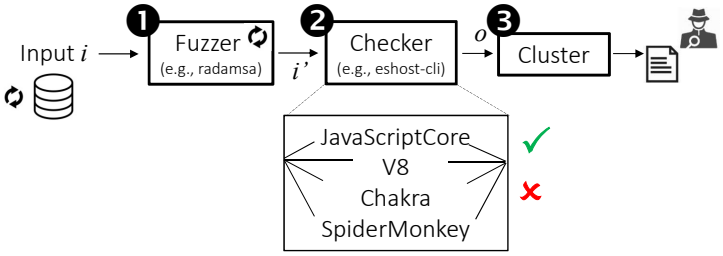


Figure 6 – Differential Testing infrastructure overview.

prioritizes warnings and clusters them in groups (step 3). We describe these features in Sections 3.1.1 and 3.1.2. Note that a number of reasons exist, other than a bug, to explain discrepancy (see Tables 5 and 8) and there is no clear automatic approach to precisely distinguish false and True Positives (TP). As such, a human needs to inspect the warning to classify the issue. As mentioned earlier, this justifies why differential testing is challenging to automate at the functional level.

For step 2, we considered using the open-source tool eshost-cli [1], also used at Microsoft, for checking output discrepancy. However, we noticed that eshost-cli does not handle discrepancies involving crashes, but our oracle checks if there are a crash report on the running. It is also important to note that our checker does not support the case where the test fails in all engines but the kind of failure (e.g., exception thrown) is different. Currently, our infrastructure does not report discrepancy for that case. For that, it would be necessary to properly parse the error message to retrieve the error types. We left that as future work as we already found several discrepancies even without that.

3.1.1 Prioritization

We prioritized warnings based on their types, reflecting likelihood of manifesting a real bug. We defined two types—"hi" and "lo".

Warnings of the kind "hi" are associated with the cases where the test code executes without violating any internal checks, but it violates an assertion declared in the test itself or its harness. The rationale is that the test data is more likely to be valid in this case as execution does not raise exceptions in application code. Warnings of kind "lo" cover the remaining cases. These warnings are more likely to be associated with invalid inputs. They reflect the cases where the anomaly is observed during the execution of application functions as opposed to assertions. We observed that different engines often check pre-conditions of functions differently. It can happen, for example, that one engine enforces a weaker pre-condition, compared to another engine, on the inputs of a function and that is acceptable. In those cases, the infrastructure would report a warning that is more likely to be associated with an invalid input produced by the fuzzer, i.e., it is likely to be a "bug"

[1] Eshost-cli. Available at <https://github.com/bterlson/eshost-cli>

```
var buffer = new ArrayBuffer(64);
var view = new DataView(buffer);
view.setInt8(0,0x80);
assert(view.getInt8(-1770523502845470856) === -0x80);

// Engines Messages (1:V8, 2:JavaScriptCore, 3:SpiderMonkey):
// 1. RangeError: Offset is outside the bounds of the DataView
// 2. RangeError: byteOffset cannot be negative
// 3. RangeError: invalid or out-of-range index
```

Figure 7 – Example of a “lo” warning.

in the test code as opposed to a bug in the engine. Recall that, for differential testing, we only use seed tests that pass in all engines.

Despite the problem mentioned above, “lo” warnings can reveal bugs. Figure 7 shows one of these cases. In this example, the test instantiates an `ArrayBuffer` object and stores an 8-bit integer at the 0 position. According to the specification (TC39, 2018d), a `RangeError` exception should be thrown if a negative value is passed to the function `ToIndex`, indirectly called by the test case from the function call `getInt8()`. In this case, however, the Chakra engine did not throw any exception, as can be confirmed from the report that our infrastructure produces starting with text “Engine Messages” at the bottom of Figure 7. This is a case of undocumented precondition. It was fixed by Chakra developers and is no longer present in the most recent release of Chakra.

3.1.2 Clusterization

Clusterization is complementary to prioritization; it helps to group similar warnings reported by our infrastructure. We only clustered “lo” warnings as “hi” warnings produce messages that arise from the test case, which are typically distinct.

Figure 7 shows, at the bottom, a sequence of three elements that we use to characterize a warning–1) the identifier of an engine, 2) the exception it raises, and 3) the message it produces on a “lo” warning. This sequence of triples defines a warning signature that we use for clustering. It is worth mentioning that we filter references to code in messages as to increase ability to aggregate warnings. Any warnings, including this one, that has this same signature will be included in the same “bucket”. Considering the example from Figure 7, the signature for that cluster will be [(JavaScriptCore, “RangeError”, “byteOffset cannot be negative”), (SpiderMonkey, “RangeError”, “invalid or out-of-range index”), (V8, “RangeError”, “Offset is outside the bounds of the DataView”)].

3.1.3 Fuzzers

Fuzzers are typically categorized in two main groups–those that build inputs anew (generational) and those that modify existing inputs (mutational). We used two black-box mutational fuzzers in this study. In the following, we provide rationale for this selection.

Generational fuzzers are typically grammar-based. These fuzzers generate a new file using the grammar of the language whose inputs should be fuzzed. Intuitively, those fuzzers implement a traversal of the production rules of the input grammar to create syntax trees, which are then pretty-printed. Consequently, this approach produces inputs that are syntactically valid by construction. We analyzed four grammar-based fuzzers– Grammarinator (HODOVÁN; KISS; GYIMÓTHY, 2018), jsfunfuzz [2], LangFuzz (HOLLER; HERZIG; ZELLER, 2012), and Megadeth (GRIECO; CERESA; BUIRAS, 2016). Unfortunately, none of those were effective out of the box. For example, we produced 100K inputs with Grammarinator and only few inputs were valid. With Megadeth, we were able to produce morevalid inputs as it contains some heuristics to circumvent violations of certain typing rules. Nonetheless, running those inputs in our infrastructure we were unable to find discrepancies. Inspecting those inputs, we realized that they reflected very simple scenarios. To sum up, a high percentage of inputs that Grammarinator and Megadeth generated were semantically-invalid that we needed to discard whereas the valid inputs manifested no discrepancies. Considering jsfunfuzz, we noticed that, in addition to the issues mentioned above, it produces inputs that use functions that are only available in the SpiderMonkey engine. We would need either to mock those functions in other engines or to discard those tests. Considering LangFuzz (HOLLER; HERZIG; ZELLER, 2012), the tool is not publicly available. Another fundamental issue associated with generational fuzzers in our context is that the tests they produce do not contain assertions; to enable the integration of this kind of fuzzers in our infrastructure—we would need to look for discrepancies across compiler error messages as opposed to assertion violations. All in all, although grammar-based fuzzers have been shown effective to find real bugs (HOLLER; HERZIG; ZELLER, 2012), we did not consider those fuzzers in this study for the reasons above.

Mutational fuzzers can be either white-box or black-box. White-box mutational fuzzers are typically coverage-based. American Fuzz Lop (AFL) [3] and LibFuzzer [4] are examples of this kind of fuzzers. These fuzzers run tests inputs against instrumented versions of the program under testing with the typical goal of finding universal errors like crashes and buffer overflows. The instrumentation adds code to collect branch coverage and to monitor specific properties[5]. AFL uses coverage to determine inputs that uncover a new branch and hence should be fuzzed more whereas libFuzzer uses evolutionary generation–it tries to minimize the distances to still-uncovered branches of the program. AFL takes the instrumented program binary (say, a JS engine) and one seed input to that program (say, a JS program) and produces on output fault-revealing inputs, if found. Considering our

[2] Mozilla jsfunfuzz. Available at <https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz>

[3] American Fuzz Loop. Available at <http://lcamtuf.coredump.cx/afl/>

[4] LibFuzzer. Available at <https://llvm.org/docs/LibFuzzer.html>

[5] There are options in the clang toolchain to build programs with fuzzing instrumentation (LIBFUZZER). clang provides several sanitizers for property checking (LLVM).

context of application, we needed to instrument one runtime engine for fuzzing. We chose V8 for that. Unfortunately, we found that most of the inputs produced by AFL violate the JS grammar. Furthermore, the fuzzing task can take days for a single seed input and there is no simple way to guide the exploration[6]. That happens because the fuzzer aims to explore the entire decision tree induced from the engine's main function, including the branches associated with the higher layers of the compiler (e.g., lexer and parser). It is worth mentioning that Google mitigates that problem with libFuzzer by asking developers to create fuzz targets for specific program functions (GOOGLE, b; GOOGLE, a). Although that approach has shown to be effective, it requires domain knowledge to create the calling context to invoke the fuzz target. For that, we decide not to consider coverage-based in this study.

We used two black-box mutational fuzzers in this study–radamsa and quickfuzz (GRIECO; CERESA; BUIRAS, 2016). These fuzzers require no instrumentation and domain-knowledge. They mutate existing inputs randomly. The strength of the approach is limited by the quality of the test suite and the supported mutation operators, which are typically simple. We chose these specific fuzzers because, conceptually, one complements the other. quickfuzz creates mutations like radamsa. However, in contrast to radamsa, quickfuzz is aware of the JS syntax; it is able to replace sub-trees of the syntax tree (GRIECO; CERESA; BUIRAS, 2016) with trees created anew.

[6] Exchanged emails with the tool author.

4 EVALUATION

4.1 RESULTS

The goal of this work is to assess ability of techniques that leverage diversity to find functional bugs in JavaScript engines. Based on that, we pose three questions:

RQ1. How conformant are the engines to the Test262 suite?

RQ2. How effective is test transplantation to find bugs?

RQ3. How effective is cross-engine differential testing to find bugs?

The first question focuses on the conformance of our selected engines to the official Test262 suite (TC39, a) (Section 4.1.1). In the limit, bugs would have low relevance if the engines are too unreliable. The second question focuses on the effectiveness of test transplantation (Section 4.1.2). The rationale for using inputs from different engines is that developers consider different goals when writing tests—suites written for a given engine may cover scenarios not covered by a different engine. The third question evaluates the effectiveness of cross-engine differential testing to find bugs (Section 4.1.3). The rationale for this question is that fuzzing inputs may explore scenarios not well-tested by at least one of the engines.

4.1.1 Answering RQ1 (Conformance)

The ECMA Test262 (TC39, a) test suite serves to check conformance of engines to the JS standard. It is acceptable to release engines fulfilling the specification only partially (Kangax, 2018). We expect that the pass rate on this suite provide some indication of the engine's maturity. In the limit, it is not desirable to flood bug reports on engines at early stages of development. For this experiment, we ran the suite once a day for seven consecutive days and averaged the passing ratios. Table 3 shows the average number of passing tests over this period. The variance of results was negligible; for that reason, we omitted standard deviations. We noticed that all four engines but Chakra used some variant of the Test262 suite as part of their regression process, but we used the same version in this experiment (TC39, a).

Results show that there are still many unsupported scenarios as can be observed from the percentages in the Table 3. The number of passing tests is high and similar for JSC, V8, and SpiderMonkey whereas Chakra performs worse compared to the other engines. Note also that Chakra is both the engine that has the lowest passing ratio in this test suite and the one we were able to find more bugs (as per Figure 2), the engine is recent as well (see Table 1). Although it is plausible to find correlation between the passing ratios

Table 3 – Percentage of passing tests on the Test262 conformance suite.

engine	% passing
JSC	92%
V8	95%
Chakra	75%
SpiderMonkey	93%

Table 4 – Number of failures with Test Transplantation.

test suite\engine	JSC	V8	SpiderMonkey	Chakra
JSC	-	10	10	59
V8	41	-	3	5
SpiderMonkey	218	107	-	281
Duktape	0	4	4	1
JerryScript	23	25	22	23
JSI	0	0	0	0
Tiny-js	0	0	0	0
total	282	146	39	369

and reliability as measured by the number of bugs found, we do not imply causality. It is important to note that failures in this conformance test suite indicates missing features as opposed to bugs.

> *Summary:* All engines seem to adhere well to the JS standard. Except for Chakra, the passing ratio of all engines is above 90%.

4.1.2 Answering RQ2 (Test Transplantation)

This section reports results of test transplantation. More specifically, we analyzed the failures observed when running a test suite original from a given engine in another engine. Intuitively, we want to assess how effective is the idea of cross-fertilization of testing knowledge among JS developers.

4.1.2.1 Methodology

In this experiment, a developer with experience in JS analyzed each test failure, affecting a particular engine, and classified that failure as potentially fault-revealing or not. The authors supervised the classification process to validate correctness. For the potentially fault-revealing cases, one of the authors inspected the scenario and, if agreed on the classification, reported the bug to the issue tracker of the affected engine. Notice that we did not tracking the time-consuming of the experiments and the time required to analyze these alarms.

4.1.2.2 Results

Table 4 shows the number of failures observed for each pair of test suite and engine. The first column shows the test suites and the first row shows the engines that run those tests. We use a dash ("-") to indicate that we did not consider the combinations that associate a test suite with its parent engine; failures in those cases would either indicate regressions or flaky tests as opposed to unknown bugs for that engine. As explained in Section 2.3, we used in this experiment the 6,602 tests included in the rectangle area under column "type-in-all" from Table 2. Running those tests we observed a total of 836 failures manifested across 612 distinct files (9.2% of total). Table 4 shows that SpiderMonkey was the engine that failed the least whereas Chakra was the engine that failed the most. The SpiderMonkey test suite also revealed more failures than any other, as expected, given that it is the suite with more tests (see Table 2).

The sources of False Positives (FP) found in this experiment are as follows: **Undefined Behavior.** FP of this kind are manifested when tests cover implementation-dependent behavior, as defined in the ECMA262 specification (TC39, 2018c). For example, one of the tests from JerryScript uses the function `Number.toPrecision([precision])`, which translates a number to a string, considering a given number of significant digits. The floating-point approximation of the real value is implementation-dependent, making that test to pass only in Chakra. **Timeout/OME**[1]**.** FP of this kind typically manifest when the engine that runs the test does not optimize the code as the original engine of the test. As result, the test fails to finish at the specified time budget or it exceeds the memory budget. For example, a test case from JSC defines a function with a tail-call recursion. The test fails in all engines but JSC, which implements tail-call optimization. **Not implemented.** FP of this kind manifest when a test fails because it covers a function that is part of the official spec, but is not implemented in the target engine yet. For example, at the time of writing, Chakra did not implement by default various properties from the `Symbol` object. These properties are only available activating the ES6 experimental mode with the flag `-ES6Experimental`. **Non-Standard Element.** These cases manifest when a function or an object property is undefined in the execution engine but we were unable to capture that by looking for error types like `ReferenceError` and `TypeError`. **Other.** This category includes other sources of FP. For example, it includes the cases where the test was valid for some previous version of the spec but is no longer valid for the current spec.

Table 5 shows the distribution of False Positives (FP) and True Positives (TP). The sum of the numbers in this table correspond to the number of files that manifested failures, i.e., 612. Considering false positives, "Undefined Behavior" was the most predominant source. Considering TP, we found a reasonable number of duplicate reports, but not high enough to justify attempting to automate the detection of duplicates.

Table 6 lists all bugs we found with test transplantation. The first column shows

[1] ome is for out of memory error.

Table 5 – Distribution of FP and TP (TT).

	source	#
FP	Undefined Behavior	204
	Timeout/OME	23
	Not Implemented	54
	Non-Standard Element	122
	Other	174
TP	Duplicate	12
	Bug	25
	WontFix	1

Table 6 – List of bugs reports from Test Transplantation.

Issue#	Date	Engine	Version	Status	Url	Severity	Suite
1	4/18	JSC	606.1.9.4	New	#184749	-	JerryScript
2	4/23	Chakra	1.9	**Confirmed**	#5033	2	SpiderMonkey
3	4/29	Chakra	1.9	**Fixed**	#5065	2	SpiderMonkey
4		Chakra	1.10-beta	**Confirmed**	#5067	2	SpiderMonkey
5	4/29	JSC	606.1.9.4	New	#185130	-	SpiderMonkey
6	5/02	JSC	606.1.9.4	New	#185208	-	SpiderMonkey
7		JSC	606.1.9.4	**Fixed**	#185211	2	SpiderMonkey
8	5/02	Chakra	1.10-beta	**Fixed**	#5087	3	SpiderMonkey
9	5/17	Chakra	1.10-beta	**Fixed**	#5187	2	JSC
10	5/21	Chakra	1.10-beta	**Fixed**	#5203	2	SpiderMonkey
11	6/28	Chakra	1.11-beta	**Fixed**	#5388	2	JSC
12	7/10	Chakra	1.11-beta	**Confirmed**	#5442	2	JerryScript
13	7/18	Chakra	1.10.1	**Fixed**	#5478	2	SpiderMonkey
14	7/18	JSC	233840	Duplicated	#187777	2	JerryScript
15	7/18	Chakra	1.10.1	**Fixed**	#5549	2	JerryScript
16	8/07	Chakra	1.10.1	**Fixed**	#5576	3	JerryScript
17	8/07	JSC	234555	**Fixed**	#188378	2	JerryScript
18	8/07	Chakra	1.10.1	**Fixed**	#5579	3	JerryScript
19	8/07	JSC	234654	**Fixed**	#188382	2	JerryScript
20	8/08	V8	7.0.181	**Fixed**	#8033	3	JerryScript
21	8/08	JSC	234689	New	#188407	-	JerryScript
22	8/16	V8	7.0.237	WontFix	#8064	2	Duktape
23	8/22	Chakra	1.10.2	**Fixed**	#5621	2	SpiderMonkey
24	8/22	JSC	235121	**Fixed**	#188874	2	SpiderMonkey
25		JSC	235121	**Fixed**	#188875	2	SpiderMonkey
26	8/22	V8	7.0.244	**Fixed**	#8082	2	SpiderMonkey
27		Chakra	1.10.2	**Fixed**	#5624	2	SpiderMonkey
28	8/23	JSC	235121	New	#188877	-	SpiderMonkey

the identifier we assigned to the bug,column “Engine” shows the affected engine, column “Status” shows the status of the bug report at the time of the writing. The status string appears in bold face for status “Confirmed” or higher, i.e., “Assigned” and “Fixed”. Column “Severity” shows the severity of confirmed bugs, and, finally, column “Suite” shows the name of the engine that originated the test. Considering severity levels, we found that JSC (Apple, 2018) and SpiderMonkey (Mozilla, 2018) developers use five levels, whereas Chakra (Microsoft, 2018) and V8 (The Chromium Project, 2018) developers use only three. As usual, the smallest the severity value the highest the severity of the bug. We use a

dash ("-") in place of the severity level for the cases where the bug report is pending confirmation. Of the 28 bugs we reported in this experiment, 21 were promoted from the status "New" to "Confirmed". Of these, 19 are severity-2 bugs. Although we did not find any critical bugs, most of the bugs are seemingly important as per the categorization given by engineers. Analyzing the issue tracker of Chakra, we found that severity-1 bugs are indeed rare. Considering the number of bug reports confirmed by developers, Chakra was the engine with the highest number–14 (with 11 bugs fixed). Considering the remaining engines, V8 developers confirmed the three bugs we reported, fixing two. Curiously, Google engineers confirmed the issued bug reports in a few hours. Overall, we found that development teams of other engines, specially JSC took much longer to analyze bug reports as can be observed in the JSC stacked bar from Figure 2a. However, the bugs the JSC team confirmed were quickly fixed.

Summary: Test transplantation was effective at finding functional bugs. Although the cost of classifying failures was non-negligible, the approach revealed several non-trivial bugs in three of the four engines we analyzed.

4.1.3 Answering RQ3 (Differential Testing)

This section reports the results obtained with cross-engine differential testing. More precisely, we report results obtained by fuzzing test inputs, running those inputs on different engines, and checking the outcomes with a differential oracle.

4.1.3.1 Methodology

The experimental methodology we used is as follows. As explained in Section 3.1.3, we used Radamsa [2] and QuickFuzz (GRIECO; CERESA; BUIRAS, 2016) for fuzzing. To avoid experimental noise, we only fuzz test files that pass in all engines–a total of 23,808 tests satisfy this restriction. Those tests appear under the column "no-fail-in-all" on Table 2. We want to avoid the scenario where fuzzing produces a fault-revealing input based on a test that was already revealing failures on some engine. This decision facilitates our inspection task; it helps us establish cause-effect relationship between fuzzing and the observation of discrepancy. We configured our infrastructure (see Figure 6) to produce 20 well-formed fuzzed files per input file, i.e., the number of fuzzing iterations can exceed the number above as we discard generated files that are syntactically invalid.

[Exploratory Phase.] For the first three months of the study, our inspection process was exploratory. In this phase, we wanted to learn whether or not black-box fuzzers could reveal real bugs and how effective was the hi-lo warning classification. We expected the

[2] Radamsa official repository. Available at <https://github.com/aoh/radamsa>

number of warnings to increase dramatically compared to the previous experiment and, if we realized that the ratio of bugs from lo warnings was rather low, we could focus our inspection efforts on hi warnings. To run this experiment, we trained eight students in analyzing the warnings that our infrastructure produces. The students were enrolled in a graduate-level testing class. We listed warnings in a spreadsheet and requested the students to update an "owner" column indicating who was working on it, but we did not enforce a strict order on the warnings the students should inspect. Recall from Section 3.1.2 that we clustered lo warnings in buckets. For that reason, we only listed one lo warning per representative class/bucket in the spreadsheet. First, we explained, through examples, the possible sources of false alarms they could find and then we asked the students to use the following procedure when finding a suspicious warning. Analyze the parts of the spec related to the problem and, if still suspicious, look for potential duplicates on the bug tracker of the affected engine using related keywords. If none was reported, indicate in the spreadsheet that that warning is potentially fault-revealing. We encouraged students to use lithium [3] to minimize long test cases. A bug report was filed only after one of the authors reviewed the diagnosis. Each student found at least one bug using this methodology.

[Non-Exploratory Phase.] Results obtained in the exploratory phase confirmed our expectations that most of the bugs found during the initial period of investigation were related to hi warnings. For that reason, we changed our inspection strategy. This time, only the co-authors inspected the bugs using a similar procedure as before. However, the set of warnings inspected and the order of inspection changed. We restricted our analysis to hi warnings and, aware that we would be unable to analyze each and every warning reported, we grouped those warnings per engine, analyzing each group in a round-robin fashion. At each iteration, we analyzed five warnings in each group. A warning belongs to the group of a given engine if only that engine manifests distinct behavior, i.e., it produces a distinct output compared to others. We separated in a distinct group the warnings for which two engines diverge. The rationale for this methodology was to give attention to each engine more uniformly, enabling more fair comparison across engines.

4.1.3.2 Results

Table 7 shows statistics of hi warnings. The table breaks down hi warning by the affected engine, i.e., the engine manifesting distinct output among those analyzed. Column "+1" shows the cases where more than one engine disagree on the output. Note from the totals that the ordering of engines is consistent with the one observed on Table 4, with Chakra and JSC in first and second places, respectively, in number of warnings.

Table 8 shows the distribution of false positives per source. The sources of imprecision

[3] Lithium official repository. Available at <https://github.com/MozillaSecurity/lithium>

Table 7 – Number of hi warning reports per engine.

fuzzer\engine	JSC	V8	Chakra	SpiderMonkey	+1
radamsa	151	50	331	94	628
quickfuzz	83	63	351	21	403
total	234	113	682	115	1031

Table 8 – Distribution of FP and TP (DT).

		radamsa	quickfuzz
FP	Undefined Behavior	42	16
	Timeout/OME	30	15
	* Invalid Input	46	55
	* Error Message Mismatch	41	12
TP	Duplicate	36	28
	Bug	15	7

are as defined in Section 4.1.2 with the addition of two new sources, which we did not observe before. These new sources are marked with a "*" in the table. The source "Invalid Input" indicates that the test input violated some part of the specification. For example, the test indirectly invoked some function with unexpected arguments; this happens because fuzzing is not sensitive to function specifications. Consequently, it can replace valid with invalid inputs. The source "Error Message Mismatch" corresponds to the cases where the fuzzer modifies the assertion expression (e.g., some string expression or regular expression).

Table 9 shows the list of bugs we reported. The table shows the fuzzing tool used ("Fuzzer"), the JS engine affected ("Engine"), the status of the bug report ("Status"), the severity of the bug report ("Sev."), the priority that we assigned to the warning that revealed the bug ("Priority"), and the test suite from the original test input ("Suite"). So far, fifteen of the bugs we reported were confirmed, eleven of which were fixed. Note that one bug report that we submitted was rejected on the basis that the offending JS file manifested an incompatibility across engine implementations that was considered to be acceptable. As of now, we did not find any new bugs on SpiderMonkey; the bugs we found were duplicates and were not reported. For V8, we reported 2 bugs, all of them confirmed and fixed.

Summary: Cross-engine differential testing was effective at finding JS engines bugs, several of which have been fixed already.

Data Availability. The data, including the tests, warning reports, and diagnos-

Table 9 – List of bugs reports from Differential Testing.

Issue#	Date	Fuzzer	Engine	Version	Status	Url	Sev.	Priority	Suite
1	4/12	radamsa	Chakra	1.9	**Fixed**	#4978	2	**lo**	JSC
2	4/12	radamsa	Chakra	1.9	WontFix	#4979	-	hi	JSC
3	4/14	radamsa	JSC	606.1.9.4	New	#184629	-	hi	JSC
4	4/25	radamsa	Chakra	1.9	**Fixed**	#5038	2	hi	JerryScript
5	4/29	radamsa	JSC	606.1.9.4	**Fixed**	#185127	2	hi	JerryScript
6	4/30	radamsa	Chakra	1.10-beta	**Confirmed**	#5076	2	hi	TinyJS
7	4/30	radamsa	JSC	606.1.9.4	New	#185156	-	hi	TinyJS
8	5/02	radamsa	JSC	606.1.9.4	**Fixed**	#185197	2	**lo**	SpiderMonkey
9	5/10	radamsa	Chakra	1.10-beta	**Confirmed**	#5128	3	hi	JerryScript
10	5/17	radamsa	Chakra	1.10-beta	**Fixed**	#5182	2	hi	V8
11	5/24	radamsa	JSC	606.1.9.4	**Fixed**	#185943	2	hi	JSC
12	6/26	radamsa	JSC	606.1.9.4	**Fixed**	#187042	2	hi	JerryScript
13	7/10	quickfuzz	JSC	606.1.9.4	**Fixed**	#187520	2	hi	JerryScript
14	7/10	quickfuzz	Chakra	1.11-beta	**Confirmed**	#5443	2	hi	JerryScript
15	8/21	radamsa	Chakra	1.10.2	**Fixed**	#5617	3	hi	Test262
16	8/21	radamsa	V8	7.0.244	**Fixed**	#8078	2	hi	Test262
17	8/23	quickfuzz	JSC	235121	New	#188899	-	hi	Test262
18	8/23	quickfuzz	V8	7.0.244	**Fixed**	#8088	3	hi	Test262
19	8/24	quickfuzz	Chakra	1.10.2	**Confirmed**	#5630	2	hi	Test262
20	8/24	quickfuzz	JSC	235318	New	#188920	-	hi	Test262
21	8/24	quickfuzz	JSC	235318	New	#188930	-	hi	Test262
22	8/21	radamsa	Chakra	1.11.6.0	WontFix	#5968	3	hi	SpiderMonkey

tic outcomes, is publicly available from a preserved repository <https://github.com/damorimRG/jsengines-differential-testing>.

4.2 DISCUSSION

4.2.1 Overview

Table 10 shows the overview of the warnings found and reported in this study. Notice that there are 3 rejected bugs reported, we observed that the behavior of the engines affected working as intended, two of them refer to lack of memory in Chakra and the other in V8, the test case was not well implemented. The duplicated bugs are related to warnings that we obtained in manual analysis, but only the issue#14 was reported in JSC engine. It was duplicated due to an older issue[4] reported in 2016 which reports an incorrect behavior on Array methods with objects whose length exceeds the max value. In this study, we reported a total of 50 with 36 confirmed and 29 fixed. In Section 4.2.2 we described how the engines are violated by those bugs.

[4] WebKit Issue#163417. Available at <https://bugs.webkit.org/show_bug.cgi?id=163417>

Warnings	#
Reported	50
New	10
Confirmed	36
Fixed	29
Rejected	3
Duplicated	67
Severity-2	31
Severity-3	8

Table 10 – Overview of the experiments.

4.2.2 Example Bug Reports

This section discusses a sample of bugs reports. The selection criteria we used was: (i) to cover all engines we found bugs–Chakra, JSC, and V8, (ii) to cover each technique–test transplantation and differential testing (with radamsa and with quickfuzz), (iii) to cover a case of rejected bug report, and (iv) to use short tests (for space).

Issue #4, Table 9. The code snippet in Figure 8 shows a test that reveals a bug in Chakra.

```
{
    var a = {
        valueOf: function(){
            return "\x00"
        }
    }
    assert(+a === 0)
}
```

Figure 8 – Warning captured involving unary plus expression.

The object property `valueOf` stores a function that returns a primitive value identifying the target object (MDN, a). The original version of this code returns an empty string whereas the version of the code modified by the radamsa fuzzer returns a string representation of a null character (`NUL`).The unary plus expression `"+a"`, used in the assertion, is equivalent to the operation `ToNumber(a.valueOf())` that converts a string to a number, otherwise the operation returns Not a Number (NaN) (MDN, b). This test fails in all engines but Chakra. For all three engines the string cannot be parsed as an hexadecimal. As such, they produce a NaN as result and the test fails as expected. Chakra, instead, incorrectly converts the string to zero, making the test to pass. As Table 9 shows, the Chakra team fixed the issue soon after we reported the problem.

Issue #18, Table 9. The code in Figure 9 shows the test input we used to reveal a bug in

V8. This code snippet was obtained by fuzzing a Test262 test with quickfuzz. In its original version, a string (omitted for space), passed as argument to the `eval` function, encoded the actual test. The fuzzer replaced the string argument with a function whose body is a `break` statement outside a valid block statement. Section B.3.3.3 from the spec (Ecma Internacional, a) documents how eval should handle code containing function declarations. According to the spec (Ecma Internacional, d), the virtual machine should throw an early error–in this case, a SyntaxError–if the break statement is not nested in an iterable or a switch statement. All engines, but V8, behave as expected in this case.

```
eval(
    function b(a){
        break;
    }
)
```

Figure 9 – Warning classified as true positive in V8 engine.

Issue #18, Table 6. The code snippet in Figure 10 shows the test input we used to reveal a bug in JSC. This test originates from JerryScript test suite; the bug was found during the test transplantation experiment.

```
var obj = {};
var arr = [];
try {
    arr.sort(obj);
    assert(false);
} catch (e) {
    assert(e instanceof TypeError);
}
```

Figure 10 – Warning captured involving a non-callable object.

According to the specification (TC39, 2018a), the parameter to the `Array.sort` function should be a comparable object or an undefined value, otherwise it should throw a `TypeError`. In this case, JSC accepts a non-callable object as argument to `sort` and the test fails in the subsequent step. The other engines raise a `TypeError` as expected.

Issue #2, Table 9. The code snippet below shows an example input related to a bug report that we issued to the Chakra development team, but they did not accept.

```
function test() {
    return typeof String.prototype.repeat === "function"
                && "foo".repeat(657604378) === "foofoofoo";
}
```

Figure 11 – Warning classified as incompatibility by design.

This is a testcase original from the JSC suite that the radamsa fuzzer modified. The original test used the integer literal 3 as argument to `repeat()`, but this test uses a long integer instead. As result, the engine crashes. The team answered that this was an incompatibility by design as the function was not expected to receive such a long value.

5 KEY FINDINGS AND LESSONS

The main findings of this study are as follows.

- Reporting defects in open source projects;
- Both techniques we investigated were successful in revealing bugs (overall, 50 bugs reported);
- Even simple black-box fuzzers can create surprisingly interesting inputs. We conjecture that there is room to find more bugs using other fuzzers;
- Mozilla's SpiderMonkey appears to be the most reliable JS engine we analyzed, with Google's V8 after it.

The main lessons we learned from this study are as follows: 1) Even for software projects with fairly clear specifications, as the case of JavaScript (TC39, 2018c), there is likely to be (a lot of) variation between different implementations and, therefore, opportunities for bugs. 2) Not only multiple different implementations can be leveraged in differential testing, but differences in test suites can also be important. 3) Finding functional/non-crash bugs with differential testing is feasible on real, complex, widely used pieces of software. 4) Further reducing cost of inspection is an important problem. Although the inspection activity was not uninterrupted, it is safe to say that each warning required a substantial amount of time to analyze for potential false alarms. In fact, many hi warnings reported with differential testing were not analyzed. In determining cost, we observed, from experience, that the complexity of the JS specifications that the original test covers increases cost of diagnosing (as developers need to read and understand those) and the availability of alternative implementations (for the cases warnings are revealed through differential testing) reduces the cost of diagnosis. We prefer to see such problem as an opportunity for future research. For example, applying learning techniques to prioritize the warnings more likely to be faulty (in the spirit of the work of Chen and colleagues (CHEN et al., 2017)) may be a promising avenue to explore. Recall that the rate of TP of the techniques we studied is rather small. 5) We learned that reporting real bugs is a great way to train (and encourage) students in software testing. Students praised the experience of diagnosing failures, understanding part of the specs (as needed), writing bug reports, participating in discussions on issue trackers, and observing the change of status. That was a relatively self-contained hands-on activity that enabled students to engage in a real-life serious industrial project.

6 THREATS TO VALIDITY

As it is the case for most empirical evaluations, our findings are subject to internal, external, and construct validity threats.

6.1 INTERNAL VALIDITY

Considering internal validity, conceptually, it is possible that the authors of this paper made mistakes in the implementation of the scripts supporting the experiments. To mitigate this threat, we carefully inspected the implementation and results, looking for inconsistencies whenever possible. For example, in our initial experiments, we intended to report some warnings that affects the engines as true bug which involves infinite recursion. We added a timeout on our experiments to ignores those cases. We applied fixes to some test files that had no assertions functions, but we inspectioned very cautious those fixes to avoid unexpected failures. There are test suites that require an external library to perform test file assertions, we have implemented a way to merge these files to run just on file per engine. Students participated in this experiment and were well advised and critical on executing, inspecting and reporting bugs.

6.2 EXTERNAL VALIDITY

As for external validity, our results might not generalize to other test inputs and engines. To mitigate this threat, we carefully selected inputs from various sources according to a well-defined criteria. Likewise, we selected the engines by using using a well-defined criteria and found that the engines selected were associated, certainly not by coincidence, with the browsers informally considered the most popular in the market.

The classifier described in Section 2.4 might not generalize to other language scenarios or inputs involving the new features of ES using the same params and constants, but to improve this classifier, the scripts to generate it are available and can easily updated for the new scenarios.

We ran the fuzzers without tracking seeds, but the metrics proposed in this study do not use this feature to measure performance and time spent to find bugs. Every JS engine can be integrated into our infrastructure, but with minor fixes if its supports the criteria. At this point, test files from frontend libraries using NodeJS cannot be integrated to run with the engines chosen. In addition, some of the selected suites contains failing tests. We removed those cases to not influence the experiments on DT. Finally, we ran the transplantation and mined suites several times to remove flaky tests (e.g. tests involving non-determinism).

6.3 CONSTRUCT VALIDITY

In terms of construct validity, we used standard metrics to determine the effectiveness of the testing techniques we studied (e.g., number of bugs confirmed and fixed and severity).

Engine developers were responsible for determining the labels of the bug reports and their severity. Consequently, these metrics originate from a trusted source.

7 RELATED WORK

7.1 DIVERSITY IN TESTING

The idea of test set diversity dates back to the eighties (WHITE; COHEN, 1980; OSTRAND; BALCER, 1988). The assumption behind test set diversity is that faults often spread contiguously in the input domain. Consequently, it should be beneficial to partition the input domain and explore it more evenly as to find bugs faster. Later in the nineties, Chen and Yu (Chen; Yu, 1996) analyzed numerical software and confirmed the continuity assumption. They found that faults, most often, manifest themselves in regular patterns across the input domain (CHEN et al., 2010; FELDT et al., 2016). Such observations led researchers to investigate generalizations of the approach to non-numerical data types and strategies to explore the input (or output) space to solve different problems in Software Engineering (e.g., test input generation and test selection) (MAYER, 2005; BUENO; WONG; JINO, 2007; CIUPA et al., 2008; ALSHAHWAN; HARMAN, 2012; ALSHAHWAN; HARMAN, 2014; FELDT et al., 2016). This study explores diversity of sources of test cases and diversity of engine implementations. We remain to investigate the diversity of test cases themselves. In principle, that could help reduce the number of alarms to inspect, for example.

7.2 DIFFERENTIAL TESTING

Several different applications of differential testing have been proposed in recent years. Chen and colleagues (CHEN et al., 2018) recently proposed a technique to generate X.509 certificates based on Request For Proposals (RFC) as specification with the goal of detecting bugs in different SSL/TLS implementations. Those bugs can compromise security of servers which rely on these certificates to properly authenticate the parties involved in a communication session. Lidbury and colleagues (LIDBURY et al., 2015) and Donaldson and colleagues (DONALDSON et al., 2017) have been focusing on finding bugs in programs for graphic cards (e.g., OpenCL). These programs use the Single-Instruction Multiple-Data (SIMD) programming abstraction and typically run on GPUs. Perhaps the application of differential testing that received most attention to date was compiler testing. In 1972, Purdom (PURDOM, 1972) proposed the use of a generator of sentences from grammars to test correctness of automatically generated parsers. After that, significant progress has been made. Lammel and Shulte proposed Geno to cross-check XPath implementations using grammar-based testing with controllable combinatorial coverage (LÄMMEL; SCHULTE, 2006). Yang and colleagues (YANG et al., 2011b) proposed CSmith to randomly create C programs from a grammar, for a subset of C, and then check the output of these programs in different compilers (e.g., GCC and LLVM). Le and colleagues (LE; AFSHARI; SU, 2014)

proposed “equivalence modulo inputs”, which creates variants of program which should have equivalent behavior compared to the original, but for which the compiler manifests discrepancy. Differential testing has also been applied to test refactoring engines (DANIEL et al., 2007), to test symbolic engine implementations (KAPUS; CADAR, 2017), to test disassemblers and binary lifters (PALEARI et al., 2010; KIM et al., 2017), and very recently to test JavaScript debuggers (LEHMANN; PRADEL, 2018). All in all, it has shown to be flexible and effective for a wide range of applications. Surprisingly, not much work has been done on differential testing of JS engines. Mozilla uses differential testing to look for discrepancies across different configurations of the same version of its SpiderMonkey engine (using the “compare_jit” flag of jsfunfuzz whereas we focus on discrepancy across engines). Patra e Pradel (2016) evaluated their language-agnostic fuzzing strategy using differential testing. Their focuses on finding differential bugs across multiple browsers. As such they specialized their fuzzer to HTML and JS (see Section 7.4). In Chen et al. (2016) is presented the *classfuzz*, a tool that uses coverage-directed fuzzing technique to improving the generation of valid mutants on different JVM implementations. In this case, their focuses is the Java environment and evaluate their technique with another coverage-directed fuzzing. In contrast to Patra e Pradel (2016) and Chen et al. (2016), we did not propose new techniques; our contribution was empirical.

7.3 TESTING JS PROGRAMS

Patra and colleagues (PATRA; DIXIT; PRADEL, 2018) proposed a lightweight approach to detect conflicts in JS libraries that occur when names introduced by different libraries collide. This problem was found to be common as the design of JS allows for overlaps in namespaces. A similar problem has been investigated by Nguyen and colleagues (NGUYEN; KäSTNER; NGUYEN, 2014) and Eshkevari and colleagues (ESHKEVARI et al., 2014) in the context of PHP programs, which are popular in the context of Content Management Systems as WordPress. The focus of this work is on testing JS engines as opposed to JS programs. Our goal is therefore orthogonal to theirs.

7.4 TESTING JS ENGINES

The closest work to ours was done by Patra e Pradel (2016). Their work proposes a language-agnostic fuzzer to find cross-browser HTML+JS discrepancies.The sensible parts of the infrastructure they built are the checks of input validity (as to reduce waste/cost) and output correctness (as to reduce FP). Patra and Pradel work is complementary to ours–in principle, we could use their fuzzer in our evaluation. Recently, Han, Oh e Cha (2019) shows a tool called CodeAlchemist which is a fuzzing tool for semantics-aware generation of test cases to find vulnerabilities in JS engines. The work proposed the technique to fragmentize JS seeds into blocks of code, those *code bricks* represents a

valid JS Abstract Syntax Tree (AST). The tool generates a test case semantically and syntactically valid and evaluates on real-world engines and found 19 bugs whereas 11 are exploitable bugs. The main difference of our work to theirs is in goal–we aim at assessing reliability of JS engines and find bugs on them using simple approaches whereas they aim at proposing a new technique. Fuzzing is an active area of investigation with development of new techniques both in academia and industry. Several fuzzing tools exist focused on JS. Section 3.1.3 briefly explain different fuzzing strategies and tools. Existing techniques prioritize automation with a focus on finding crashes; see the sanitizers used in libFuzzer [1], for instance. In general, it is important for these tools that a warning reveals something potentially alarming as a crash given that fuzzing is a time-consuming operation, i.e., the ratio of bugs found per inputs generated is often very low. Our approach contrasts with that aim as we focus on finding errors manifested on the output, which rarely result in crashes and, consequently, would go undetected by current fuzzing approaches. It is should be noted, however, that such problems are not unimportant as per the severity levels reported in Tables 6 and 9.

[1] Libfuzzer tutorial. Available at <https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md>

8 CONCLUSIONS

JavaScript (JS) is very popular today. Bugs in engine implementations often affect lots of people and organizations. Implementing correct engines is challenging because the specification is intentionally incomplete and evolves frequently. Finding bugs in JS engines is challenging for similar reasons.

This study reports on a study to evaluate two diversity-aware techniques for finding bugs in JS–test transplantation and cross-engine differential testing. The first technique explores diversity of test sources; it runs the test suite of one given engine in another engine. The second technique explores diversity of implementations; it fuzzes existing inputs and then compares the output produced by different engines with a differential oracle.

We found that both techniques were very effective at finding bugs in JS engines. Overall, we reported 50 bugs in this study. Of which, 36 were confirmed by developers and 29 were fixed. Although more work is necessary to reduce cost of manual analysis, we found that our results provide strong evidence that exploring techniques should be encouraged to find functional bugs in JavaScript engines.

For the future work, more research needs to be done to improve automation of the tool. We plan to explore techniques to prioritize the warnings reported by these techniques. Using machine learning techniques for prioritization, similar to what Chen and colleagues (CHEN et al., 2017) did to prioritize the warnings reported by CSmith (YANG et al., 2011b). We need to provide support for addition of new engines. Recently, the Facebook released the open-source engine called Hermes [1] which performs optimizations on Android applications in React Native [2] and seems a promising context for finding new bugs.

The scripts to run the experiments for this study is available in our repository at <https://github.com/damorimRG/jsengines-differential-testing> and the data with the bugs reported is publicly available at <https://figshare.com/s/ee2e3821c2f022c7f5cc>.

[1] Hermes engine. Available at <https://hermesengine.dev/>

[2] React Native. Available at <https://facebook.github.io/react-native/>

REFERENCES

ALSHAHWAN, N.; HARMAN, M. Augmenting test suites effectiveness by increasing output diversity. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 1345–1348. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337414>.

ALSHAHWAN, N.; HARMAN, M. Coverage and fault detection of the output-uniqueness test selection criteria. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2014. (ISSTA 2014), p. 181–192. ISBN 978-1-4503-2645-2. Disponível em: <http://doi.acm.org/10.1145/2610384.2610413>.

Apple. *Severity levels WebKit bugs (JavaScriptCore)*. 2018. Available at <https://webkit.org/bug-prioritization/>.

ARGYROS, G.; STAIS, I.; JANA, S.; KEROMYTIS, A. D.; KIAYIAS, A. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016. (CCS '16), p. 1690–1701. ISBN 978-1-4503-4139-4. Disponível em: <http://doi.acm.org/10.1145/2976749.2978383>.

BECKETT, S. *Worstward ho*. [S.l.]: John Calder London, 1983.

BRUMLEY, D.; CABALLERO, J.; LIANG, Z.; NEWSOME, J.; SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007. (SS'07), p. 15:1–15:16. ISBN 111-333-5555-77-9. Disponível em: <http://dl.acm.org/citation.cfm?id=1362903.1362918>.

BUENO, P. M. S.; WONG, W. E.; JINO, M. Improving random test sets using the diversity oriented test data generation. In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. Association for Computing Machinery, 2007. (ASE07), p. 10–17. Disponível em: <https://doi.org/10.1145/1292414.1292419>.

Camilo Bruni–V8 engineer. *for-in undefined behavior*. 2018. Available at <https://v8project.blogspot.com/2017/03/fast-for-in-in-v8.html>.

CHEN, C.; TIAN, C.; DUAN, Z.; ZHAO, L. Rfc-directed differential testing of certificate validation in ssl/tls implementations. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 859–870. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180226>.

CHEN, J.; BAI, Y.; HAO, D.; XIONG, Y.; ZHANG, H.; XIE, B. Learning to prioritize test programs for compiler testing. In: *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE '17), p. 700–711. ISBN 978-1-5386-3868-2. Disponível em: <https://doi.org/10.1109/ICSE.2017.70>.

CHEN, T. Y.; KUO, F.-C.; MERKEL, R. G.; TSE, T. H. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 83, n. 1, p. 60–66, jan. 2010. ISSN 0164-1212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2009.02.022>.

Chen, T. Y.; Yu, Y. T. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, v. 22, n. 2, p. 109–119, Feb 1996. ISSN 0098-5589.

CHEN, Y.; SU, T.; SUN, C.; SU, Z.; ZHAO, J. Coverage-directed differential testing of jvm implementations. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2016. (PLDI '16), p. 85–99. ISBN 978-1-4503-4261-2. Disponível em: <http://doi.acm.org/10.1145/2908080.2908095>.

CHEN, Y.; SU, Z. Guided differential testing of certificate validation in ssl/tls implementations. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2015. (ESEC/FSE 2015), p. 793–804. ISBN 978-1-4503-3675-8. Disponível em: <http://doi.acm.org/10.1145/2786805.2786835>.

Chromium. *V8 JavaScript Engine*. Available at <https://chromium.googlesource.com/v8/v8.git>.

Chromium. *Issue 4247*. 2015. Available at <https://bit.ly/2O08uW2>.

CIUPA, I.; LEITNER, A.; ORIOL, M.; MEYER, B. Artoo: Adaptive random testing for object-oriented software. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 71–80. ISBN 978-1-60558-079-1. Disponível em: <http://doi.acm.org/10.1145/1368088.1368099>.

DANIEL, B.; DIG, D.; GARCIA, K.; MARINOV, D. Automated testing of refactoring engines. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 185–194. ISBN 978-1-59593-811-4. Disponível em: <http://doi.acm.org/10.1145/1287624.1287651>.

DONALDSON, A. F.; EVRARD, H.; LASCU, A.; THOMSON, P. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, ACM, New York, NY, USA, v. 1, n. OOPSLA, p. 93:1–93:29, out. 2017. ISSN 2475-1421. Disponível em: <http://doi.acm.org/10.1145/3133917>.

Ecma Internacional. *Changes to EvalDeclarationInstantiation*. Available at <https://www.ecma-international.org/ecma-262/8.0/index.html#sec-web-compat-evaldeclarationinstantiation>.

Ecma Internacional. *Ecma Internacional*. Available at <https://www.ecma-international.org>.

Ecma Internacional. *Number.toPrecision specification*. Available at <https://www.ecma-international.org/ecma-262/8.0/index.html#sec-number.prototype.toprecision>.

Ecma Internacional. *Static Semantics: Early Errors*. Available at <https://www.ecma-international.org/ecma-262/8.0/index.html#sec-break-statement-static-semantics-early-errors>.

ELLIOTT, T. *The State of the Octoverse: machine learning*. 2019. Available at <https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/>.

ESDiscuss. *Array new attributed caused bugs*. 2014. Available at <https://esdiscuss.org/topic/array-prototype-values-is-not-web-compat-even-with-unscopables>.

ESHKEVARI, L.; ANTONIOL, G.; CORDY, J. R.; PENTA, M. D. Identifying and locating interference issues in php applications: The case of wordpress. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 157–167. ISBN 978-1-4503-2879-1. Disponível em: <http://doi.acm.org/10.1145/2597008.2597153>.

FELDT, R.; POULDING, S.; CLARK, D.; YOO, S. Test set diameter: Quantifying the diversity of sets of test cases. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2016. p. 223–233.

GOOGLE. *Getting Started with libFuzzer at Chromium*. <https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/getting_started.md>.

GOOGLE. *libFuzzer Tutorial*. <https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md>.

GRIECO, G.; CERESA, M.; BUIRAS, P. Quickfuzz: an automatic random fuzzer for common file formats. In: ACM. *Proceedings of the 9th International Symposium on Haskell*. [S.l.], 2016. p. 13–20.

HAN, H.; OH, D.; CHA, S. K. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In: *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'19)*. [S.l.: s.n.], 2019.

HODOVÁN, R.; KISS, Á.; GYIMÓTHY, T. Grammarinator: a grammar-based open source fuzzer. In: ACM. *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. [S.l.], 2018. p. 45–48.

HOLLER, C.; HERZIG, K.; ZELLER, A. Fuzzing with code fragments. In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2012. (Security'12), p. 38–38. Disponível em: <http://dl.acm.org/citation.cfm?id=2362793.2362831>.

John Resig. *JavaScript in Chrome*. 2018. Available at <https://johnresig.com/blog/javascript-in-chrome/>.

Kangax. *ECMAScript6 compatibility*. 2018. Available at <http://kangax.github.io/compat-table/es6/>.

KAPUS, T.; CADAR, C. Automatic testing of symbolic execution engines via program generation and differential testing. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ASE 2017), p. 590–600. ISBN 978-1-5386-2684-9. Disponível em: <http://dl.acm.org/citation.cfm?id=3155562.3155636>.

KIM, S.; FAEREVAAG, M.; JUNG, M.; JUNG, S.; OH, D.; LEE, J.; CHA, S. K. Testing intermediate representations for binary analysis. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ASE 2017), p. 353–364. ISBN 978-1-5386-2684-9. Disponível em: <http://dl.acm.org/citation.cfm?id=3155562.3155609>.

KUSNER, M.; SUN, Y.; KOLKIN, N.; WEINBERGER, K. From word embeddings to document distances. In: *International Conference on Machine Learning*. [S.l.: s.n.], 2015. p. 957–966.

LÄMMEL, R.; SCHULTE, W. Controllable combinatorial coverage in grammar-based testing. In: UYAR, M. Ü.; DUALE, A. Y.; FECKO, M. A. (Ed.). *Testing of Communicating Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 19–38. ISBN 978-3-540-34185-7.

LE, V.; AFSHARI, M.; SU, Z. Compiler validation via equivalence modulo inputs. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2014. (PLDI '14), p. 216–226. ISBN 978-1-4503-2784-8. Disponível em: <http://doi.acm.org/10.1145/2594291.2594334>.

LEHMANN, D.; PRADEL, M. Feedback-directed differential testing of interactive debuggers. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. [s.n.], 2018. p. 610–620. Disponível em: <https://doi.org/10.1145/3236024.3236037>.

LIBFUZZER. *LibFuzzer*. <https://llvm.org/docs/LibFuzzer.html>.

LIDBURY, C.; LASCU, A.; CHONG, N.; DONALDSON, A. F. Many-core compiler fuzzing. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2015. (PLDI '15), p. 65–76. ISBN 978-1-4503-3468-6. Disponível em: <http://doi.acm.org/10.1145/2737924.2737986>.

LING, W.; DYER, C.; BLACK, A. W.; TRANCOSO, I. Two/too simple adaptations of word2vec for syntax problems. In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. [S.l.: s.n.], 2015. p. 1299–1304.

LLVM. *clang documentation*. Available at <http://clang.llvm.org/docs/>.

MAYER, J. Lattice-based adaptive random testing. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005. (ASE '05), p. 333–336. ISBN 1-58113-993-4. Disponível em: <http://doi.acm.org/10.1145/1101908.1101963>.

MDN. *Object.valueOf documentation*. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/ValueOf>.

MDN. *Unary Plus - ES6 specifications*. <https://www.ecma-international.org/ecma-262/8.0/index.html#sec-unary-plus-operator>.

Microsoft. *ChakraCore*. Available at <https://github.com/Microsoft/ChakraCore>.

Microsoft. *Severity levels Chakra bugs*. 2018. Available at <https://github.com/Microsoft/ChakraCore/wiki/Label-Glossary>.

MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G.; DEAN, J. Distributed representations of words and phrases and their compositionality. In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*. Lake Tahoe, Nevada, USA: Curran Associates Inc., 2013. p. 3111–3119.

Mozilla. *jsfunfuzz at Mozilla*. Available at <https://mzl.la/2LsctZL>.

Mozilla. *SpiderMonkey Project*. Available at <https://github.com/mozilla/gecko-dev>.

Mozilla. *Triage Process for Firefox Components in Mozilla-central and Bugzilla*. 2018. Available at <https://github.com/mozilla/bug-handling/blob/master/policy/triage-bugzilla.md>.

NGUYEN, H. V.; KäSTNER, C.; NGUYEN, T. N. Exploring variability-aware execution for testing plugin-based web applications. In: *ICSE*. [S.l.: s.n.], 2014. p. 907–918.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, ACM, New York, NY, USA, v. 31, n. 6, p. 676–686, jun. 1988. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/62959.62964>.

PALEARI, R.; MARTIGNONI, L.; ROGLIA, G. F.; BRUSCHI, D. N-version disassembly: Differential testing of x86 disassemblers. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2010. (ISSTA '10), p. 265–274. ISBN 978-1-60558-823-0. Disponível em: <http://doi.acm.org/10.1145/1831708.1831741>.

PATRA, J.; DIXIT, P. N.; PRADEL, M. Conflictjs: Finding and understanding conflicts between javascript libraries. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 741–751. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180184>.

PATRA, J.; PRADEL, M. *Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data*. [S.l.], 2016.

PETSIOS, T.; TANG, A.; STOLFO, S.; KEROMYTIS, A. D.; JANA, S. Nezha: Efficient domain-independent differential testing. In: *2017 IEEE Symposium on Security and Privacy (SP)*. [S.l.: s.n.], 2017. p. 615–632.

PURDOM, P. A sentence generator for testing parsers. *BIT Numerical Mathematics*, v. 12, n. 3, p. 366–375, Sep 1972. ISSN 1572-9125. Disponível em: <https://doi.org/10.1007/BF01932308>.

RedMonk. *The RedMonk Programming Language Rankings: June 2018*. 2018. Available at <https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/>.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. In: RUMELHART, D. E.; MCCLELLAND, J. L.; GROUP, C. P. R. (Ed.). Cambridge, MA, USA: MIT Press,

1986. cap. Learning Internal Representations by Error Propagation, p. 318–362. ISBN 0-262-68053-X. Disponível em: <http://dl.acm.org/citation.cfm?id=104279.104293>.

Simply Technologies. *Why is JavaScript So Popular?* 2018. Available at <https://www.simplytechnologies.net/blog/2018/4/11/why-is-javascript-so-popular>.

SIVAKORN, S.; ARGYROS, G.; PEI, K.; KEROMYTIS, A. D.; JANA, S. Hvlearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. [s.n.], 2017. p. 521–538. Disponível em: <https://doi.org/10.1109/SP.2017.46>.

Stackify. *Most Popular and Influential Programming Languages of 2018*. 2018. Available at <https://stackify.com/popular-programming-languages-2018/>.

StackOverflow community. *Elements order in a "for (... in ...)" loop*. 2018. Available at <https://stackoverflow.com/questions/280713/elements-order-in-a-for-in-loop>.

TC39. *Official ECMA262 Conformance Test Suite*. Available at <https://github.com/tc39/test262>.

TC39. *TC39 GitHub repo*. Available at <http://tc39.github.io/>.

TC39. *Array sort*. 2018. Available at <https://tc39.github.io/ecma262/#sec-array.prototype.sort>.

TC39. *ECMA262 repository*. 2018. Available at <https://tc39.github.io/ecma262/>.

TC39. *ECMAScript 2017 Language Specification* . 2018. Available at <https://www.ecma-international.org/ecma-262/8.0/>.

TC39. *TypeConversion, ToIndex function*. 2018. Available at <https://tc39.github.io/ecma262/#sec-toindex>.

The Chromium Project. *Chromium Bug Labels*. 2018. Available at <https://www.chromium.org/for-testers/bug-reporting-guidelines/chromium-bug-labels>.

WebKit. *WebKit Project*. Available at <https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore>.

WHITE, L. J.; COHEN, E. I. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6, n. 3, p. 247–257, May 1980. ISSN 0098-5589.

YANG, X.; CHEN, Y.; EIDE, E.; REGEHR, J. Finding and understanding bugs in c compilers. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2011. (PLDI '11), p. 283–294. ISBN 978-1-4503-0663-8. Disponível em: <http://doi.acm.org/10.1145/1993498.1993532>.

YANG, X.; CHEN, Y.; EIDE, E.; REGEHR, J. Finding and understanding bugs in c compilers. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2011. (PLDI '11), p. 283–294. ISBN 978-1-4503-0663-8. Disponível em: <http://doi.acm.org/10.1145/1993498.1993532>.

ZHANG, T.; KIM, M. Automated transplantation and differential testing for clones. In: *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE '17), p. 665–676. ISBN 978-1-5386-3868-2. Disponível em: <https://doi.org/10.1109/ICSE.2017.67>.