Aug 24th, 2023

Dear Prof. Pezze and Reviewers,

Thank you for your helpful and insightful comments! We have greatly benefited from all the comments and suggestions. In revising this paper, we have conducted additional experiments and made major changes to address the comments and suggestions raised by each reviewer. Here is a summary of the major changes:

1. We add another research question to show the semantics-preserving of deobfuscated programs in Evaluation. In this RQ, we conduct case studies to analyze programs that are not semantically equivalent to the original ones. We use a modern JavaScript utility library recommended by Reviewer 1, lodash, which includes 400 JS files of source code and 94 test files with 408 unit test cases. Additionally, this dataset is a complement to our original dataset.
2. We refine the definition of JavaScript program obfuscation to keep aligned with prior definitions of obfuscation. Additionally, we make a further explanation to show JSimpo's generalization ability over various versions of JS obfuscators.

In the following sections, we provide detailed responses in blue to the comments from each reviewer in their appearance order in the reviews.

Thank you very much for your time and kind consideration.

Best regards,

Tianyu, Ding, Ying, and Tao

## Reviewer: 1

Comment 1.1

Confusing definition for obfuscation: see my initial comments about not considering tools like UglifyJS as obfuscators. In your response you say that "it can be reasonable to classify UglifyJS as an obfuscator". Please provide a reference to a prior paper, developers resource or any other solid empirical evidence that justifies this divergence from the traditional definition of obfuscation/minification, as presented by Skolka et al. [68] and Moog et al. [52].

Response 1.1

Thanks for your suggestion.

We clarify the definition of JavaScript obfuscation to keep aligned with prior definitions of obfuscation. As presented by Skolka et al. [68] and Moog et al. [52], we do not classify minification tools like UglifyJS as obfuscators. Then we make the following revisions in our paper:

1. In Background, we do not take UglifyJS as an example obfuscator, and use a statement-level obfuscator, Esoteric Javascript-obfuscator, instead.

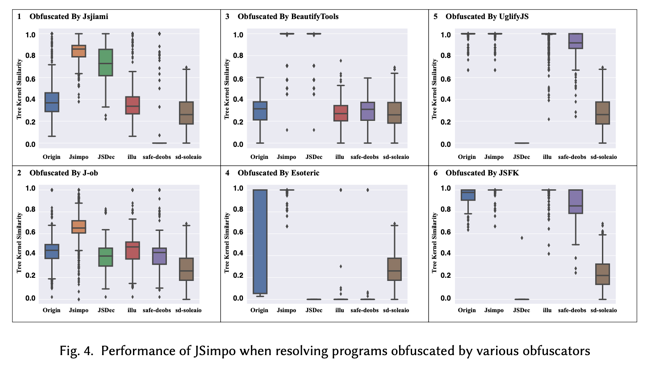
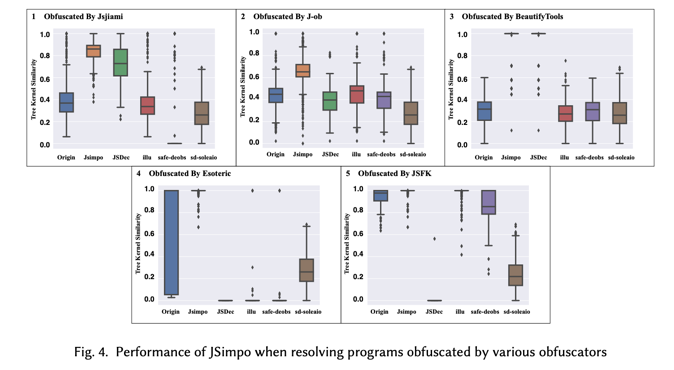
~~~

2.1 Textual Obfuscation

Textual obfuscation is a typical technique used in conventional obfuscation tools, such as Esoteric Javascript-obfuscator [7]. They convert identifiers, such as variable names and function names, into random strings. This technique is lightweight and can efficiently obfuscate JS programs on a large scale. However, textual obfuscation is not robust if the syntax tree structure is not changed. Previous research proposed deep-learning-based techniques [67, 79] to infer the original strings from an obfuscated program based on syntax tree structures. Modern advanced obfuscation tools [84] also provide structural obfuscation besides textual obfuscation.

~~~

2. In Evaluation (RQ3), we remove the results of UglifyJS. We change the Figure 4 from the left one to the right one.

3. In Related Work, we add another paragraph to explain the difference between JavaScript obfuscation and minification as follows:

~~~

Besides JavaScript Obfuscation, JavaScript minification [57, 73] also changes the structure of JS programs. Different from obfuscation, the target of minification is to reduce code size during transformation, thus improving website performance. It includes techniques like deleting whites- paces and comments, which is also included in obfuscation because it reduces the readability of JS programs. Advanced minification techniques even modify a program’s structure, e.g., removing dead code (e.g., unreachable or redundant code) and inlining functions [1, 2]. UglifyJS [20] is one representative JS minification approach. The main difference between these approaches and obfuscators is that the program structure modified by minification approaches does not increase the burden of program analysis. Therefore, we do not conduct structural deobfuscation on programs after minification.

~~~

Comment 1.2

Claim that JSimpo's transformations are semantics-preserving: Both in your response and in the discussion section you hold to your claim that "JSimpo should keep the semantics of programs before and after deobfuscation", without providing any empirical evidence to support this. You discarded the suggestion to perform a study of deobfuscation's correctness by saying "One major reason for not examing the correctness of deobfuscated programs is that JS programs are executed in website engines with remote servers, so it is difficult to execute them locally". I believe this is a poor counter-argument: As suggested in my review, you can take some well-tested, standalone JavaScript project like lodash and perform some correctness experiments using it (obfuscate + deobfuscate + run tests). Such libraries/projects do not have any dependencies on remote servers and can be run outside of the browser, e.g., in Node.js.

Overall, your evaluation is not very convincing, i.e., you consider only 2000 programs, and you use poor metrics. As I understand, some of the considered programs may be trivial, e.g., no function definitions, or they might not compile. Why not filter these out in the first place? Also, considering how efficient your tool is, why not run on a larger dataset, e.g., https://www.sri.inf.ethz.ch/js150 to show that your tool works with a variety of code samples? Regarding the selection of metrics, I found the structural similarity and relative token number metrics insufficient. What does it mean that two programs are 100% structurally similar? How likely is it that they have equivalent semantics? Also, two programs with similar (number of) tokens may behave very differently when executed. I believe that these metrics are too coarse-grained and you should consider instead some notion of functional equivalence (see the suggestion above about using unit tests).

Response 1.2

Thanks for your suggestion.

We add another research question to show the semantics-preserving of deobfuscated programs in Evaluation.

- In this research question, we choose a modern JavaScript utility library recommended by Reviewer 1, lodash, which includes 400 JS files of source code and 94 test files with 408 unit test cases. Here each JS file corresponds to one functionality provided by lodash.

- Methodology: We obfuscate all these 400 JS files and deobfuscate them to see how many unit test cases among 408 ones can be passed. Note that lodash has matured test cases, programs passed all test cases can be regarded as semantically correct, thus equivalent to the original ones.

- We include the content of this research question as follows:

~~~

RQ2:To what extent can JSimpo preserve the semantics of programs?

Methodology. This research question evaluates the semantic correctness of programs deobfuscated by JSimpo. Note that an existing study [73] shows that current obfuscators, e.g., J-ob, might change the semantics of the transformed programs in the obfuscation process, so we directly compare the semantics of one program after obfuscation and deobfuscation to its original one.

In this research question, we choose a modern JavaScript utility library, lodash [24], as our dataset for evaluation. It has more than 57 thousand stars on GitHub. It contains 400 files of source code as provided functionalities and 94 locally executable test files (including 408 unit test cases). We first obfuscate its source-code files by jsjiami and J-ob and then deobfuscate them by JSimpo to see whether they are semantically equivalent to the original files. Here, we take the number (portion) of passed test files and unit test cases as our evaluation metric for semantic equivalence. We mark one program after obfuscation and deobfuscation as semantically equivalent to its original program if all its test cases are passed.

Overall Statistics. Table 4 shows the number (portion) of passed test files and unit test cases after we obfuscate and deobfuscate the source-code files in lodash. We show that JSimpo keeps the semantics of almost all programs after deobfuscation. Specifically, 98.9% of test files on programs obfuscated by jsjiami and 100.0% of test files on programs obfuscated by J-ob are passed. This result indicates that JSimpo can preserve the semantics of most JS programs during deobfuscation.

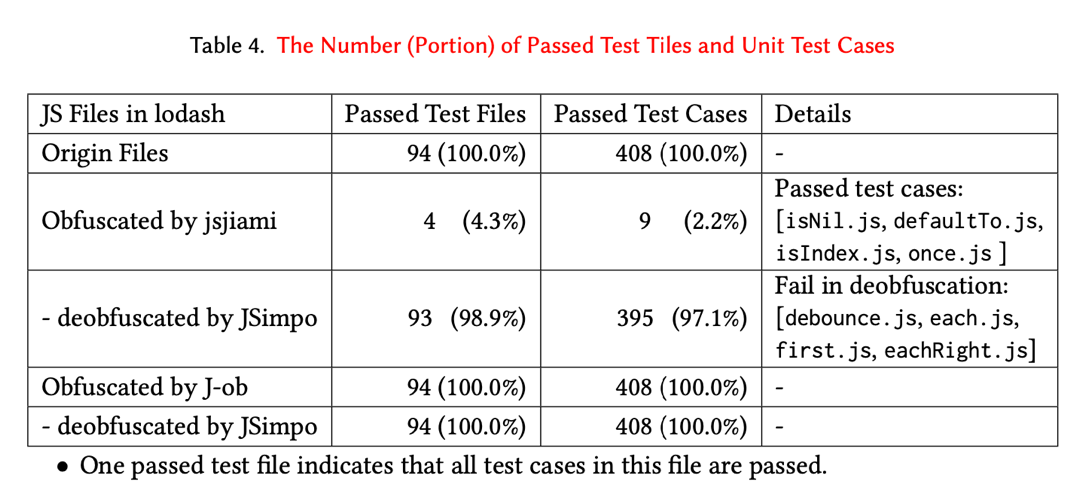
Table 4 also shows that only 4.3% of test files on programs obfuscated by jsjiami are passed, which is mainly caused by lodash’s test engine, mocha. Here, we take Listing 3 as an example for explanation. In Lines 5-8, it invokes an anonymous function, whose return statement defines an array variable, \_0x5f4c, and this variable is used in a spurious function in Line 11. However, \_0x5f4c is not recognized during the execution of its test file, thus leading to an execution failure. Only four programs (as listed in Table 4) do not include this pattern after obfuscation, so their test file are passed. Although lodash’s test engine cannot recognize this variable, our dynamic execution engine, nodejs, can recognize \_0x5f4c, thus removing them during deobfuscation.

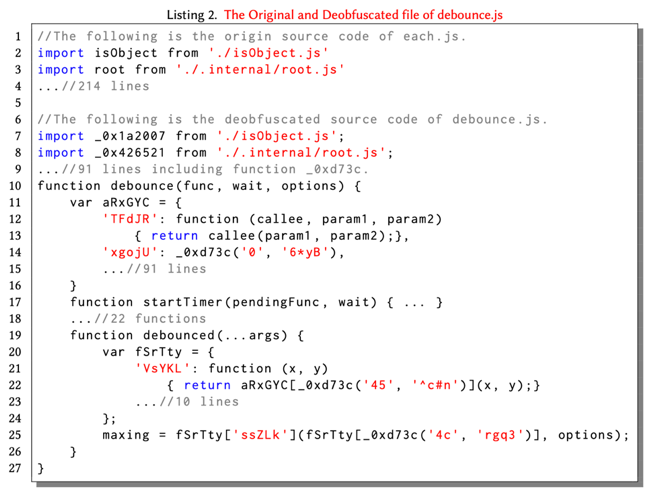
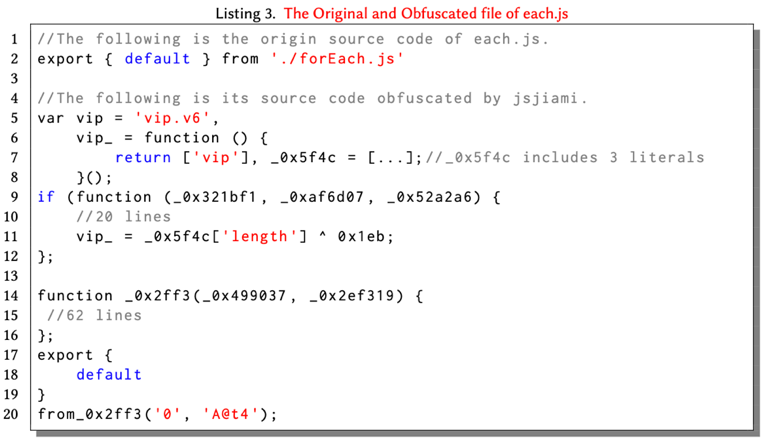
A Case Study of Failed Test Cases. As listed in Table 4, there are four programs whose test cases are not passed. We first make a case study on debounce.js because each.js, first.js, and eachRight.js do not have corresponding test cases. The main reason of debounce.js’s failed test cases is that JSimpo does not deobfuscate this file due to its long dependency chain of spurious functions. Listing 2 shows a code snippet of debounce.js. In Line 25, this program calls function fSrTty and function \_0xd73c. In Lines 21-22, function fSrTty calls function aRxGYC and function \_0xd73c. In Lines 11-16, function aRxGYC also calls function \_0xd73c. During deobfuscation, JSimpo scans function fSrTty before function aRxGYC, thus not labeling function fSrTty as a spurious function. This failure can be addressed by executing JSimpo several times, which increases the efficiency consumption as a trade-off.

Case Studies of Semantic Difference. Besides these failed test cases, we also make case studies on programs with semantic difference after obfuscation and deobfuscation. Lines 2-3 and 7-8 in Listing 2 shows that jsjiami might change the names of imported modules (J-ob also changes these names). For example, the module isObject is changed into \_0x1a2007. This difference does not change the semantics of this program because JavaScript allows users to rename the module when there is only one module from a given file, e.g., ‘./isObject.js’. However, when there is more than one module from one file, both obfuscators still change their names, thus leading to an execution failure. This scenario does not occur in lodash because it is recommended to include one module in one JS file [23].

As for each.js, first.js, and eachRight.js, we also show that their semantics are changed during obfuscation. Listing 3 shows that the export statement export { default } from ‘./forEach.js’ in Line 2 is changed into export { default } from\_0x2ff3(‘0’, ‘A@t4’) while there should be a space between the token from and the token \_0x2ff3. This difference is caused by jsjiami, thus not affecting the semantic-preserving property of JSimpo.

~~~



Comment 1.3

Lack of qualitative evaluation: While the added example helps the reader understand the limits of your tool and the output it produces, it would help if you would describe much more such examples even if not adding them verbatim in the paper. For example, cases in which the deobfuscation alters the semantics of the original program but "a little different semantics can also be tolerable for malware detection and comprehension".

Response 1.3

Thanks for your suggestion.

In response 1.2, the RQ we added also includes case studies on programs that are not semantically equivalent to the original programs. Our case studies show that only four programs obfuscated by jsijiami are not semantically equivalent due to JSimpo's failing in deobfuscation. According to these case studies, the semantics of three programs (each.js, first.js, and eachRight.js) have been changed after obfuscation. Our case studies in Listing 3 also show that how we comprehend these files.

The failure upon only one JS file (debounce.js) is caused due to its long dependency chain, and such failure can be addressed by executing JSimpo several times, which increases the efficiency consumption as a trade-off.

The case studies are listed as follows:

~~~

A Case Study of Failed Test Cases. As listed in Table 4, there are four programs whose test cases are not passed. We first make a case study on debounce.js because each.js, first.js, and eachRight.js do not have corresponding test cases. The main reason of debounce.js’s failed test cases is that JSimpo does not deobfuscate this file due to its long dependency chain of spurious functions. Listing 2 shows a code snippet of debounce.js. In Line 25, this program calls function fSrTty and function \_0xd73c. In Lines 21-22, function fSrTty calls function aRxGYC and function \_0xd73c. In Lines 11-16, function aRxGYC also calls function \_0xd73c. During deobfuscation, JSimpo scans function fSrTty before function aRxGYC, thus not labeling function fSrTty as a spurious function. This failure can be addressed by executing JSimpo several times, which increases the efficiency consumption as a trade-off.

Case Studies of Semantic Difference. Besides these failed test cases, we also make case studies on programs with semantic difference after obfuscation and deobfuscation. Lines 2-3 and 7-8 in Listing 2 shows that jsjiami might change the names of imported modules (J-ob also changes these names). For example, the module isObject is changed into \_0x1a2007. This difference does not change the semantics of this program because JavaScript allows users to rename the module when there is only one module from a given file, e.g., ‘./isObject.js’. However, when there is more than one module from one file, both obfuscators still change their names, thus leading to an execution failure. This scenario does not occur in lodash because it is recommended to include one module in one JS file [23].

As for each.js, first.js, and eachRight.js, we also show that their semantics are changed during obfuscation. Listing 3 shows that the export statement export { default } from ‘./forEach.js’ in Line 2 is changed into export { default } from\_0x2ff3(‘0’, ‘A@t4’) while there should be a space between the token from and the token \_0x2ff3. This difference is caused by jsjiami, thus not affecting the semantic-preserving property of JSimpo.

~~~

Comment 1.4

Generalization of JSimpo: You seem to miss the point of Referee 1 and mine that the indicators you identified for tools like javascript-obfuscator might be an implementation artifact of these tools, and not generic signatures of JavaScript obfuscators. Regarding the different versions of the tools, you seem to misunderstand my comment as multiple configuration options. What I meant is different releases of the same tool: https://github.com/javascript-obfuscator/javascript-obfuscator/tags

Response 1.4

Sorry for our misunderstanding. We first list your previous comment here.

~~~

Previous Comment:

Finally, my impression is that you overfit to jsjiami and javascript-obfuscator, an aspect you also acknowledge throughout the paper. In particular, the Spurious Function Detection (3.3.1) and Indicator Detection (3.2.1) seem to be specially designed with these obfuscators in mind. While RQ2 aims to study how well your tool generalizes to other obfuscators, the results are not convincing because of the poor metrics selection and the limited dataset (see discussion above). It is also unclear how stable the considered indicators are over different versions of the obfuscators. Does javascript-obfuscator at version 3.0.1 produce code with the same indicators as version 4.0.2?

~~~

- According to our investigation, javascript-obfuscator at version 3.0.1 produce code with the same indicators as version 4.0.2 for two main reasons.

- First, we investigate the commit message of javascript-obfuscator in the reference URL: https://github.com/javascript-obfuscator/javascript-obfuscator/tags. These commits fall into various categories, e.g., fix bugs (3.2.5), semantic inequivalence (3.2.3), option conflict (3.2.6), typo in CHNAGELOG (4.0.0), and support new JS language features (3.0.0). However, these different versions do not change how javascript-obfuscator generates indicators and spurious functions.

- Second, we make further case studies of how javascript-obfuscator with two versions (v4.0.0 and v2.19.1) works. The time gap between these two versions is about two years. From our following results, we show that javascript-obfuscator with various versions only changes the order of spurious functions. For example, fuction \_0x2b43 (v4.0.0) corresponds to function \_0x3b11 (v2.19.1), and function \_0x4363 corresponds to function \_0x1c1f.

- The following is after.js (in lodash) obfuscated by [javascript-obfuscator@4.0.0](mailto:javascript-obfuscator@4.0.0)

~~~

function \_0x4363(\_0x17679c, \_0x3e2dc7) {

var \_0x2b43cb = \_0x2b43();

return \_0x4363 = function (\_0x4363ed, \_0x489707) {

\_0x4363ed = \_0x4363ed - 0xb9;

var \_0x4df944 = \_0x2b43cb[\_0x4363ed];

return \_0x4df944;

}, \_0x4363(\_0x17679c, \_0x3e2dc7);

}(function (\_0x49e36b, \_0x34ce91) {

var \_0x584215 = \_0x4363,

\_0x305475 = \_0x49e36b();

while (!![]) {

try {

var \_0xb6ebd4 = -parseInt(\_0x584215(0xb9)) / 0x1 + -parseInt(\_0x584215(0xc0)) / 0x2 + -parseInt(\_0x584215(0xbf)) / 0x3 \* (-parseInt(\_0x584215(0xbe)) / 0x4) + parseInt(\_0x584215(0xbd)) / 0x5 \* (-parseInt(\_0x584215(0xbb)) / 0x6) + -parseInt(\_0x584215(0xc2)) / 0x7 \* (parseInt(\_0x584215(0xbc)) / 0x8) + -parseInt(\_0x584215(0xc1)) / 0x9 + parseInt(\_0x584215(0xc5)) / 0xa;

if (\_0xb6ebd4 === \_0x34ce91) break;

else \_0x305475['push'](\_0x305475['shift']());

} catch (\_0xad5d01) {

\_0x305475['push'](\_0x305475['shift']());

}

}

}(\_0x2b43, 0x59987));

function after(\_0x24019a, \_0x1bd84f) {

var \_0x2cd788 = \_0x4363;

if (typeof \_0x1bd84f !== \_0x2cd788(0xba)) throw new TypeError(\_0x2cd788(0xc3));

return \_0x24019a = \_0x24019a || 0x0,

function (...\_0x862eb8) {

var \_0x35fb89 = \_0x2cd788;

if (--\_0x24019a < 0x1) return \_0x1bd84f[\_0x35fb89(0xc4)](this, \_0x862eb8);

};

}

export default after;

function \_0x2b43() {

var \_0x1d40d0 = ['function', '78EIeCWc', '2083496OvnBZb', '50275foKZkb', '8tIjXvu', '304965yJsjaw', '27722nQOxjQ', '1857996qHbxzp', '14YXtRbD', 'Expected a function', 'apply', '17034780utpWDx', '667911zTETYq'];

\_0x2b43 = function () {

return \_0x1d40d0;

};

return \_0x2b43();

}

~~~

The following is after.js obfuscated by javascript-obfuscator@2.19.1

~~~

function \_0x3b11() {

var \_0x2339aa = [

'Expected\x20a\x20function',

'793101ibxnIX',

'2lqVVJe',

'200454eeynRE',

'1579856srctGF',

'600OfTnwl',

'3378OWtfbM',

'112sNlTMC',

'341240HqtybE',

'1073088nsYtut',

'280010HvfTUA',

'11tGcwmF',

'function'

];

\_0x3b11 = function () {

return \_0x2339aa;

};

return \_0x3b11();

}

function \_0x1c1f(\_0x24dcec, \_0x106ada) {

var \_0x3b11c6 = \_0x3b11();

return \_0x1c1f = function (\_0x1c1f86, \_0x47211c) {

\_0x1c1f86 = \_0x1c1f86 - 0x15d;

var \_0x4b780a = \_0x3b11c6[\_0x1c1f86];

return \_0x4b780a;

}, \_0x1c1f(\_0x24dcec, \_0x106ada);

}

(function (\_0x1714ea, \_0x27dae0) {

var \_0x5ca287 = \_0x1c1f, \_0x396d1b = \_0x1714ea();

while (!![]) {

try {

var \_0x130fef = -parseInt(\_0x5ca287(0x15d)) / 0x1 + parseInt(\_0x5ca287(0x15e)) / 0x2 \* (parseInt(\_0x5ca287(0x15f)) / 0x3) + parseInt(\_0x5ca287(0x160)) / 0x4 + -parseInt(\_0x5ca287(0x161)) / 0x5 \* (-parseInt(\_0x5ca287(0x162)) / 0x6) + parseInt(\_0x5ca287(0x163)) / 0x7 \* (parseInt(\_0x5ca287(0x164)) / 0x8) + parseInt(\_0x5ca287(0x165)) / 0x9 + parseInt(\_0x5ca287(0x166)) / 0xa \* (-parseInt(\_0x5ca287(0x167)) / 0xb);

if (\_0x130fef === \_0x27dae0)

break;

else

\_0x396d1b['push'](\_0x396d1b['shift']());

} catch (\_0x378c35) {

\_0x396d1b['push'](\_0x396d1b['shift']());

}

}

}(\_0x3b11, 0x7c800));

function after(\_0x12a85c, \_0x28d835) {

var \_0x2663d2 = \_0x1c1f;

if (typeof \_0x28d835 !== \_0x2663d2(0x168))

throw new TypeError(\_0x2663d2(0x169));

return \_0x12a85c = \_0x12a85c || 0x0, function (...\_0x3ba5d7) {

if (--\_0x12a85c < 0x1)

return \_0x28d835['apply'](this, \_0x3ba5d7);

};

}

export default after;

~~~

Comment 1.5

Small data set: In your response you say that "To reflect the real-world distribution of JS programs, we do not filter the JS programs in Normal-JS". If your objective is indeed to reflect the real-world distribution of JS programs, you need to consider much more programs and to ensure that they are compatible with your setup.

Response 1.5

Thanks for your suggestion.

Besides randomly chosen JS programs, such as Normal-JS and Complex-JS, we use a modern JavaScript utility library, lodash, for evaluation in RQ2 (semantics-preserving) as a complement to Normal-JS and Complex-JS. lodash includes 400 files of source code and more than 57 thousand stars. Thus, lodash is a representative dataset of real-world project-level JS files.

## Reviewer: 2

Comment 2

The authors have addressed my remarks, and overall the submission has improved. The responses to my questions are to the point, but only partially convincing. Particularly, it is not clear what improvements have been made to the paper for some questions. For example, the statistical tests should be added to the paper, not only to the letter.

Response 2.1

We sincerely thank your appreciation of our work. We have added the methodology and results of our statistical tests to our revised paper in RQ3 as follows:

~~~

To further validate JSimpo’s effectiveness, we also conduct the Kruskal-Wallis test [56] to see whether there exists a significant difference between programs deobfuscated by JSimpo and programs deobfuscated by other tools. The test statistic of programs obfuscated by jsjiami is 212.86 and its corresponding P-value is 6.44e-45. The test statistic of programs obfuscated by J-ob is 814.27 and its corresponding P-value is 6.23e-175. Note that both P-values are substantially less than 0.05 (a widely-used threshold), so we make the conclusion that there is a significant difference among the groups, thus indicating JSimpo’s effectiveness.

~~~

## Reviewer: 3

Comment 3.1 (Re: Comment and Response 2.1)

I respect that you have tried your best to raise the bar for obfuscators. I fully agree that the level of sophistication required to map obfuscated control flow indicators to existing program variables is not feasible for obfuscators. (Indeed, sound and precise analysis of JavaScript is well beyond the state of the art.)

However, for spurious function detection, I still think there that an obfuscator could invalidate the assumption that "spurious functions have no side effect outside of their belonging method". Imagine if an obfuscator introduced the following spurious function:

```

let globalIntroducedByObfuscator = 0;

function spuriousFunctionIntroducedByObfuscator() {

// complex logic...

globalIntroducedByObfuscator = someComputedValueFromAboveLogic;

// more complex logic...

globalIntroducedByObfuscator = Math.random();

// even more complex logic...

}

```

This is a side-effecting spurious function, but the side-effects are limited to executions of the function itself (so none of the existing program semantics should be affected). I believe it is possible for a spurious function to safely have side effects, as long as those side effects are only relevant to the code introduced by an obfuscator. This may technically be an unsound transformation for a few reasons, but I imagine it would work most of the time; Reviewer 1 brought up a paper that shows that obfuscators often introduce semantic differences in the code they transform, and so they are not enacting totally sound program transformations anyway. This transformation is less likely to fail than, e.g., renaming constructors.

At a high level, I am still unconvinced that the assumptions made in this work would hold up long term. I feel that small improvements to obfuscation techniques could invalidate many of the assumptions herein, such as the one presented above. The techniques employed are indeed novel, especially w.r.t. the state of the art and practice for obfuscators, but the assumptions seem unrealistic.

I will not die on this hill, though; the fact remains that JSimpo does seem to recover more of the original code than other state of the art/state of the practice obfuscators, and I do respect that.

Response 3.1

Thanks for your concern and respect.

The side effect of `globalIntroducedByObfuscator` might be mitigated by finding the closure of spurious functions. For example, the closure of spuriousFunctionIntroducedByObfuscator includes the definition of globalIntroducedByObfuscator if there is no other usage of globalIntroducedByObfuscator besides the spurious function. Then we can conduct dynamic execution on this closure. Listing 3 in our evaluation also provides an example. The variable vip might also be used in the spurious function \_0x2ff3. Considering that the dynamic execution of this closure might lead to semantic inequivalence as it might also be included by users, we decide to provide an option of whether to conduct conservative analysis when finding closure.

Additionally, as you have mentioned, JSimpo does seem to recover more of the original code than other state-of-the-art/practice obfuscators even if JSimpo does not conduct dynamic execution on such cases.

Comment 3.2 (Re: Comment and Response 2.3)

I have first hand experience with how difficult it is to run arbitrary JavaScript code, so I recognize that obstacle. That said, as an attempt to investigate the correctness of JSimpo, you could consider finding JavaScript projects with executable test suites, obfuscating them, and then deobfuscating them, running the test suite throughout to see how correctness is affected. (Reviewer 1 also mentioned this.)

I think the paper would greatly benefit from such a correctness evaluation. Even if the intent is for de-obfuscation to produce code to be manually audited, it would be easier to audit the code if it actually ran. Further, there is no explicit RQ currently investigating if the de-obfuscated programs are actually helpful to "help developers and administrators audit the security of JavaScript applications", so I feel that the evaluation is lacking. Even though it may be hard to find, there is JavaScript code out there that runs; for instance, Skolka et al. [A] have identified 10 libraries with executable test suites, and have investigated correctness w.r.t. those. To me, that would be a good start, and a correctness evaluation will only strengthen the contribution.

Response 3.2

Thanks for your suggestion.

We add another research question to show the semantics-preserving of deobfuscated programs in Evaluation.

- In this research question, we choose a modern JavaScript utility library recommended by Reviewer 1, lodash, which includes 400 JS files of source code and 94 test files with 408 unit test cases. Here each JS file corresponds to one functionality provided by lodash.

- Methodology: We obfuscate all these 400 JS files and deobfuscate them to see how many unit test cases among 408 ones can be passed. Note that lodash has matured test cases, programs passed all test cases can be regarded as semantically correct, thus equivalent to the original ones.

- We include the content of this research question as follows:

~~~

RQ2:To what extent can JSimpo preserve the semantics of programs?

Methodology. This research question evaluates the semantic correctness of programs deobfuscated by JSimpo. Note that an existing study [73] shows that current obfuscators, e.g., J-ob, might change the semantics of the transformed programs in the obfuscation process, so we directly compare the semantics of one program after obfuscation and deobfuscation to its original one.

In this research question, we choose a modern JavaScript utility library, lodash [24], as our dataset for evaluation. It has more than 57 thousand stars on GitHub. It contains 400 files of source code as provided functionalities and 94 locally executable test files (including 408 unit test cases). We first obfuscate its source-code files by jsjiami and J-ob and then deobfuscate them by JSimpo to see whether they are semantically equivalent to the original files. Here, we take the number (portion) of passed test files and unit test cases as our evaluation metric for semantic equivalence. We mark one program after obfuscation and deobfuscation as semantically equivalent to its original program if all its test cases are passed.

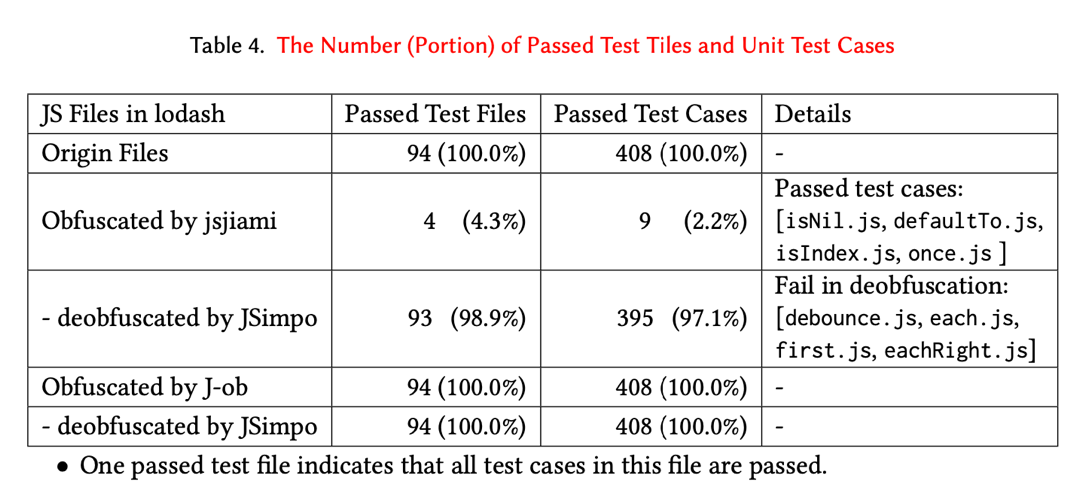
Overall Statistics. Table 4 shows the number (portion) of passed test files and unit test cases after we obfuscate and deobfuscate the source-code files in lodash. We show that JSimpo keeps the semantics of almost all programs after deobfuscation. Specifically, 98.9% of test files on programs obfuscated by jsjiami and 100.0% of test files on programs obfuscated by J-ob are passed. This result indicates that JSimpo can preserve the semantics of most JS programs during deobfuscation.

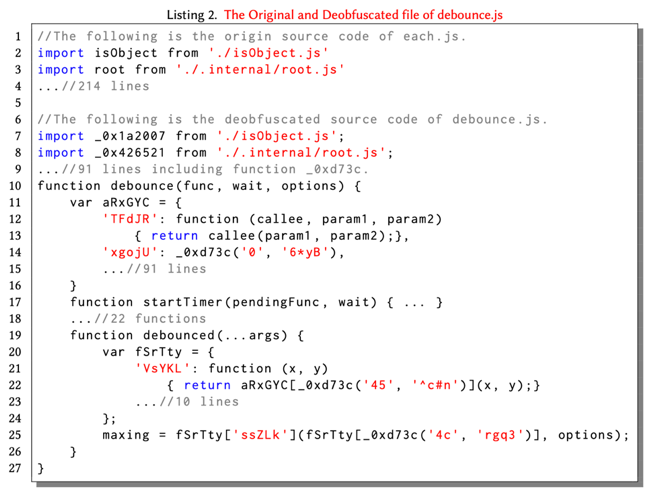
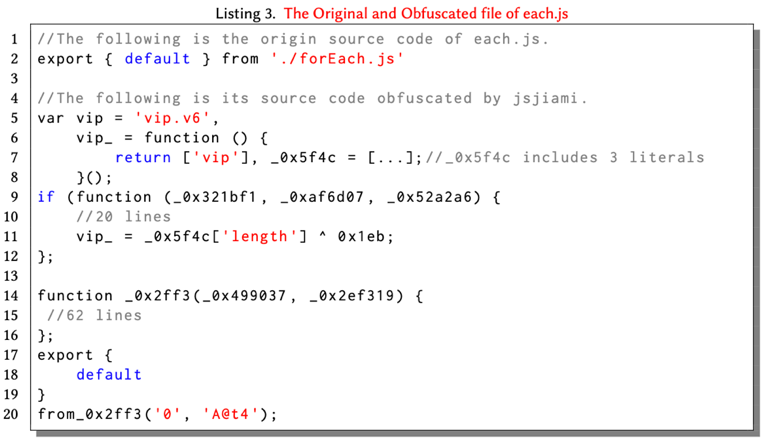
Table 4 also shows that only 4.3% of test files on programs obfuscated by jsjiami are passed, which is mainly caused by lodash’s test engine, mocha. Here, we take Listing 3 as an example for explanation. In Lines 5-8, it invokes an anonymous function, whose return statement defines an array variable, \_0x5f4c, and this variable is used in a spurious function in Line 11. However, \_0x5f4c is not recognized during the execution of its test file, thus leading to an execution failure. Only four programs (as listed in Table 4) do not include this pattern after obfuscation, so their test file are passed. Although lodash’s test engine cannot recognize this variable, our dynamic execution engine, nodejs, can recognize \_0x5f4c, thus removing them during deobfuscation.

A Case Study of Failed Test Cases. As listed in Table 4, there are four programs whose test cases are not passed. We first make a case study on debounce.js because each.js, first.js, and eachRight.js do not have corresponding test cases. The main reason of debounce.js’s failed test cases is that JSimpo does not deobfuscate this file due to its long dependency chain of spurious functions. Listing 2 shows a code snippet of debounce.js. In Line 25, this program calls function fSrTty and function \_0xd73c. In Lines 21-22, function fSrTty calls function aRxGYC and function \_0xd73c. In Lines 11-16, function aRxGYC also calls function \_0xd73c. During deobfuscation, JSimpo scans function fSrTty before function aRxGYC, thus not labeling function fSrTty as a spurious function. This failure can be addressed by executing JSimpo several times, which increases the efficiency consumption as a trade-off.

Case Studies of Semantic Difference. Besides these failed test cases, we also make case studies on programs with semantic difference after obfuscation and deobfuscation. Lines 2-3 and 7-8 in Listing 2 shows that jsjiami might change the names of imported modules (J-ob also changes these names). For example, the module isObject is changed into \_0x1a2007. This difference does not change the semantics of this program because JavaScript allows users to rename the module when there is only one module from a given file, e.g., ‘./isObject.js’. However, when there is more than one module from one file, both obfuscators still change their names, thus leading to an execution failure. This scenario does not occur in lodash because it is recommended to include one module in one JS file [23].

As for each.js, first.js, and eachRight.js, we also show that their semantics are changed during obfuscation. Listing 3 shows that the export statement export { default } from ‘./forEach.js’ in Line 2 is changed into export { default } from\_0x2ff3(‘0’, ‘A@t4’) while there should be a space between the token from and the token \_0x2ff3. This difference is caused by jsjiami, thus not affecting the semantic-preserving property of JSimpo.



Minor issues:

Also, in case the reference is missing in the reviews:

[A] P Skolka, CA Staicu, M Pradel, "Anything to hide? studying minified and obfuscated code in the web", WWW 2019

Response:

Thanks for your suggestion.

We have added this reference to Related Work-JS Obfuscation Detection, and we also include this related work to emphasize the importance of evaluating the semantic-preserving property of JSimpo. Its reference index is [73] now.

In RQ2, its description is listed as follows:

~~~

Methodology. This research question evaluates the semantic correctness of programs deobfuscated by JSimpo. Note that an existing study [73] shows that current obfuscators, e.g., J-ob, might change the semantics of the transformed programs in the obfuscation process, so we directly compare the semantics of one program after obfuscation and deobfuscation to its original one.

~~~