May 17th, 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 a paragraph in Section 3 to show the purpose of JSimpo, which focuses on only malware detection and comprehension, and we use this purpose to indicate the rationality of our tree kernel metric.
2. We add a paragraph in Section 5 to show the generalization ability of JSimpo: we have tried our best to raise the bar for obfuscators.
3. We add an example JS program that is not completely recovered in RQ1 to illustrate how JSimpo recovers an obfuscated program, and discuss why the deobfuscated programs cannot achieve a tree kernel similarity of 100%.
4. We add a detailed table in RQ3 to show the runtime overhead of JSimpo.
5. We add a paragraph in Section 7 to introduce the obfuscation and deobfuscation techniques of other programming languages.

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

The framing of your paper is flawed. You claim that prior work on JavaScript obfuscation does not study structural obfuscation, and instead targets tools like UglifyJS. However, both prior work [46, A] and practitioners do not consider such tools as obfuscators. Even though they perform program transformation on JavaScript code, as you also acknowledge in Section 5, their goal is often orthogonal, i.e., to "shorten programs to reduce program-downloading bandwidth" instead of to hinder program understanding. This being said, prior work on obfuscation detection [A] also support tools like javascript-obfuscator that perform structural obfuscation. Your claim about being the first academic work that aims to reverse this transformation may still stand, but you need to do a better job at positioning related work that previously studied advanced obfuscation techniques for JavaScript. More so, because in your evaluation (RQ3) you put a lot of emphasis on obfuscation detection, which prior work also tackles, in the presence of structural obfuscation.

Response 1.1

Thanks for your suggestion.

In terms of reducing the readability of JS programs, it can be reasonable to classify UglifyJS as an obfuscator. However, we do admit that it might be improper to focus on deobfuscating programs obfuscated by UglifyJS. In our article, we mainly focus on jsjiami and javascript-obfuscator, which are two widely used structural obfuscators.

As for obfuscation detection, we set this research question just to show the root cause of those programs failed to be deobfuscated. Our evaluation results show that JSimpo can detect most of these obfuscation techniques, so our detection rules for indicators and spurious functions are effective enough, and the next step to improve JSimpo is to improve the effectiveness of recovering.

Comment 1.2 and 1.3

Your papers lacks a qualitative evaluation. You claim that JSimpo does not change the semantics of the transformed problems, but you do not provide any evidence to support this assertion. In Section 7, you say that you "manually inspect the intermediate analysis results", but you do not provide concrete results about this process. Did you try to execute the deobfuscated programs? One could imagine a (correctness) evaluation in which you consider a well-tested project like lodash, obfuscate it with the consider obfuscators, and deobfuscate the result with JSimpo. You then execute the tests using the deobfuscated program and show that most tests pass. In this way, you would provide concrete empirical evidence that your transformations do not break the semantics of the original program. However, this evaluation may be challenging to set up because, as Skolka et al. [A] show, tools like javascript-obfuscator often change the semantics of the transformed programs in the obfuscation process. Alternatively, you can perform a user study with experts to show that the deobfuscated programs are useful for reverse engineering, even if JSimpo occasionally alters the semantics of the original programs. You could also perform such an analysis yourself using obfuscated samples created by others, e.g., https://software-lab.org/projects/obfuscation\_study.html and show that you can uncover some interesting behavior.

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 and Response 1.3

To reflect the real-world distribution of JS programs, we do not filter the JS programs in Normal-JS.

As for the correctness of deobfuscated JS programs, JSimpo should keep the semantics of programs before and after deobfuscation. We make conservative analysis when conducting deobfuscation. JSimpo might not find the indicators or spurious functions, and it is also possible that JSimpo might fail during slice symbolic execution and dynamic code execution. In these scenarios, JSimpo does not conduct further deobfuscation to keep the semantics of deobfuscated programs.

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.

Another reason is that considering our purpose of deobfuscation, deobfuscated programs with a little different semantics can also be tolerable for malware detection and comprehension. Review 1 also mentions that even obfuscators might change the semantics of programs, so it is possible that JSimpo might change the semantics of programs. We add the following paragraph in Section 3 to show the purpose of JSimpo.

We propose JSimpo to help developers and administrators audit the security of JavsScript applications. For example, cloud server hosts, such as our industrial partner, need to check uploaded JavaScript apps from third-party developers to ensure there are no security risks in the uploaded apps. Web developers can also use JSimpo to examine third-party libraries to avoid supply-chain threats.

Comment 1.4

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?

Response 1.4

We add the following paragraph in Section 3.2.1 to show the generalizable of our rules for finding indicators.

The preceding two rules are generalizable enough based on two reasonable assumptions. First, obfuscators should know the assignment and jump order of the indicators (i.e., the deterministic control-flows and data-flows). Otherwise, they cannot keep the real execution order of the obfuscated programs, thus failing to keep the semantics of original programs. Second, obfuscators do not conduct heavy analysis on the original programs for effectiveness due to JavaScript's various language features. Thus, the control-flow and data-flow of indicators might be independent of those of the original programs. Considering the indicators' deterministic and independent control-flows and data-flows, we design the preceding two rules to find out the indicators added by obfuscators.

As for different versions of javascript-obfuscator, it includes more obfuscation options (i.e., obfuscation techniques), while the main obfuscation option is still kept as the intra-procedure obfuscation and the inter-procedure options. Additionally, JSimpo is easy to be extended to address these new options (e.g., Force Transform Strings) by simple and specific rules.

Comment 1.5 (Writing issues)

Due to the space limit, we do not list all these writing issues here.

Response 1.5

Thanks for your pointing out our writing issues. We have fixed these writing issues in our revised article.

The only thing we want to point out is that we have cited an ALEXA Top 100 website including structural obfuscation while ignoring to point out the specific obfuscated JS program. Its URL is `https://hectorstatic.baidu.com/cd37ed75a9387c5b.js`.

## Reviewer: 2

Comment 2.1

The approach presented in the paper would benefit from having its assumptions revisited and relaxed, or would benefit from a thorough argument that the approach generalizes. Currently, the approach makes assumptions related to the \_implementation\_ of inter-procedural obfuscation and control flow flattening, rather than making assumptions about the strategies themselves. Concretely, in the context of control flow flattening, authors assume that the control flow indicator variables introduced by the obfuscator are always assigned constant values, and that they are only used as loop sentinels. Such an assumption would be invalidated if the indicators were implemented differently (e.g., if obfuscators added accesses to indicators outside of the loop condition check). Obfuscators should not modify the original program semantics, and so control flow flattening must create code that deterministically has the same control flow as the de-obfuscated code. Authors should either further justify the generalizability of their approach (by, e.g., arguing that their assumptions are more general than they seem, concretely, by examining a claim like "control flow flattening must be implemented using indicator variables that can only be assigned constant values"), or revisit and relax the assumptions they make and modify the approach accordingly. Authors \_did\_ examine the generalizability of their approach in the evaluation, but I do not think the results support their claim.

Response 2.1

We add the following paragraph into Section 5 to explain JSimpo's generalization ability. We have tried our best to raise the bar for obfuscators.

The Generalization Ability of JSimpo. Although obfuscators may defeat our approach by introducing variables that have data and control dependencies on other variables of the target program, doing so may take the risk of compromising the correctness of the target program. For instance, making the indicator of control-flow flattening dependent on another variable of the target program may avoid Indicator Detection (Section-III.B). However, to automatically achieve such sophisticated obfuscation, the obfuscator needs to conduct symbolic analysis on the target program and get the precise possible values of variables in the target program. Unfortunately, this is particularly difficult for JavaScript. Existing symbolic analysis techniques for JavaScript programs have an accuracy of only 87% [B] (Figure-11(b)). Such accuracy cannot guarantee the correctness of the obfuscated program.

[B] Park, Joonyoung, et al. "Accelerating JavaScript static analysis via dynamic shortcuts." Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021.

Comment 2.2

I was not satisfied with RQ2, which examined the generalizability of the approach. Authors select four other obfuscation tools and run them on the programs used in their evaluation. In each of these cases, their tool JSimpo recovers (a median of?) 100% structural similarity with the original program; this is a bewildering result, and I would have liked to see authors examine \_why\_ their approach worked so incredibly well with respect to these other obfuscators. For example, it seems that only BeautifyJS employs some structural obfuscation (intra-procedural), and interestingly JSDec performed as well as JSimpo against BeautifyJS, which I think is worth discussion. To be convinced about the generalizability of the technique presented in this paper, I would like to see JSimpo run on programs obfuscated by other complex structural obfuscation tools, and an analysis of these results. If no such tools are available, a more in-depth analysis of why JSimpo and JSDec perform equally well in de-obfuscating programs obfuscated by BeautifyTools would be interesting.

Response 2.2

As listed in Table 1, these four obfuscators do not contain many intra-procedure and inter-procedure obfuscation techniques. Thus, the main reason for JSimpo’s good performance is the simplicity of these obfuscators..

For generalization ability, we add the following paragraph in Section 5.

The Generalization Ability of JSimpo. Although obfuscators may defeat our approach by introducing variables that have data and control dependencies on other variables of the target program, doing so may take the risk of compromising the correctness of the target program. For instance, making the indicator of control-flow flattening dependent on another variable of the target program may avoid Indicator Detection in Section 2.2.2. However, to automatically achieve such sophisticated obfuscation, the obfuscator needs to conduct symbolic analysis on the target program and get the precise possible values of variables in the target program. Unfortunately, this is particularly difficult for JavaScript. Existing symbolic analysis techniques for JavaScript programs have an accuracy of only 87% [58](Figure-11(b)). Such accuracy cannot guarantee the correctness of the obfuscated program. Therefore, we have tried our best to raise the bar for obfuscators.

As for JSDec, it is specifically designed on several JS obfuscators, which include jsjiami, and BeautifyJS.

However, it fails to deobfuscate programs obfuscated by javascript-obfuscator. Even if we do not select the JSDec as the obfuscator option for JSDec, JSDec fails to deobfuscate the given program. Thus, we make the conclusion that it might contains some specific designs for BeautifyJS. (Similar to Response 2.6)

Comment 2.3

Do de-obfuscated programs need to run, or is the main purpose of de-obfuscation to assist in the detection of malware? If the code does need to be run, I would like to see a research question examining if the code runs before and after de-obfuscation.

Response 2.3

As for the correctness of deobfuscated JS programs, JSimpo should keep the semantics of programs before and after deobfuscation.

We make conservative analysis when conducting deobfuscation. JSimpo might not find the indicators or spurious functions, and it is also possible that JSimpo might fail during slice symbolic execution and dynamic code execution. In these scenarios, JSimpo does not conduct further deobfuscation to keep the semantics of deobfuscated programs. This also explains why JSimpo cannot achieve the tree kernel similarity of 100%. Each step of deobfuscation might be given up.

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.

Another reason is that considering our purpose of deobfuscation, deobfuscated programs with a little different semantics can also be tolerable for malware detection and comprehension. Review 1 also mentions that even obfuscators might change the semantics of programs, so it is possible that JSimpo might change the semantics of programs. We add the following paragraph in Section 3 to show the purpose of JSimpo.

We propose JSimpo to help developers and administrators audit the security of JavsScript applications. For example, cloud server hosts, such as our industrial partner, need to check uploaded JavaScript apps from third-party developers to ensure there are no security risks in the uploaded apps. Web developers can also use JSimpo to examine third-party libraries to avoid supply-chain threats.

Comment 2.4

Please include an analysis of related work re: obfuscation and de-obfuscation in other programming languages and domains.

Response 2.4

We add the following subsection into Section 7 to introduce obfuscation and deobfuscation of other programming languages.

Obfuscation and Deobfuscation of Other Programming Languages. Besides JavaScript, obfuscation is also widely applied in other programming languages, e.g., binary code and Android application packages (APKs). An empirical study shows that binary rewriting and API obfuscation are two widely used obfuscation techniques. Binary rewriting changes a code snippet into a semantically equivalent one that cannot be optimized by current compilers. It might include statement-level and intra-procedure obfuscation. API obfuscation, which is mainly used in APK obfuscation, removes its malicious API import table and rebuilds this table on the fly as part of the unpacking routine, thus avoiding malware detection.

Researchers propose various approaches for binary code deobfuscation. QSynth is a representative one. For a given snippet of obfuscated binary code, it uses dynamic symbolic execution to obtain its input-output correlation, and then synthesizes another program with the same semantics as the deobfuscated code. However, such approaches cannot be directly used in JS deobfuscation as it is difficult to conduct symbolic execution and programming synthesis on JS due to its various language features. As for APK deobfuscation, researchers also propose various approaches. DeGuard is a representative one. It detects the obfuscated malicious API by constructing a dependency graph for a given APK, and uses deep learning approaches to predict its possible features (e.g., APIs). Partial Evaluation.

Comment 2.5

The evaluation would benefit from some code examples showcasing de-obfuscated code to help contextualize the aggregate statistics that are presented. E.g., if the authors showed an example program, how various obfuscators mangle it, and the code recovered by JSimpo and competing de-obfuscators.

Response 2.5

We add an example JS program that is not completely recovered in RQ1 to illustrate how JSimpo recovers an obfuscated program, and then we use this example to illustrate why the deobfuscated programs cannot achieve a tree kernel similarity of 100%.

//The original JS program.

function treeEqual(node1, node2) {

if (!node1 && !node2) {

return true;

}

if (!node1 || !node2) {

return false;

}

return (

node1.val == node2.val &&

treeEqual(node1.left, node2.left) &&

treeEqual(node1.right, node2.right)

);

}

//The program obfuscated by jsjiami and deobfuscated by JSimpo.

function treeEqual(\_0x46668a, \_0x52d93c) {

if (!\_0x46668a && !\_0x52d93c) {

return true;

}

if (!\_0x46668a || !\_0x52d93c) {

return false;

}

return \_0x46668a['val'] == \_0x52d93c['val'] &&

treeEqual(\_0x46668a['left'], \_0x52d93c['left']) &&

treeEqual(\_0x46668a['right'], \_0x52d93c['right']);

}

The preceding listing shows an example of one JS program obfuscated by jsjiami and deobfuscated by JSimpo and its original program. The tree kernel of this program is 81.22%. The main reason for not achieving a tree kernel similarity of 100% is that JS contains many equivalent expressions, e.g., node.val and node[’val’]. Additionally, JSimpo removes the extra bracket in the return expression. The preceding difference changes the structure of syntax trees, thus affecting the tree kernel similarity. We do not change these equivalent expressions for two reasons. First, it is hard to distinguish which expression is used in the original program because obfuscators change both expressions into the same format, e.g., node[’val’]. Second, these equivalent expressions do not increase the complexity of a given program and can be easily understood by security engineers.

Additionally, the changed local variable names (e.g., from node1 to \_0x46668a) cannot be recovered because the original names of local variables are not kept in the obfuscated programs.

Comment 2.6

It is my understanding that JSjiami does some amount of structural obfuscation. If that is the case, I'm curious to know how JSDec managed to do so well on the programs obfuscated by JSjiami. I think the paper would benefit from such an analysis.

Response 2.6

Actually, JSDec is specifically designed on several JS obfuscators, which include jsjiami.

However, it fails to deobfuscate programs obfuscated by javascript-obfuscator. Even if we replace the name of `jsjiami` at the start of obfuscated programs with other string literals, JSDec fails to deobfuscate the given program. Thus, we conclude that it might contain some specific designs for jsjiami.

Comment 2.7

In related work, authors reference "On the (Im)possibility of Obfuscating Programs", stating that "Unfortunately, some negative theoretical results show the impossibility of general obfuscation detection". This is curious to me because the reference actually seems to claim the opposite, that true \_obfuscation\_ is impossible. On its face, this appears to suggest that de-obfuscation should certainly not be impossible. Further clarification here would be welcome.

Response 2.7

Sorry for using an improper citation here. In this paragraph, we want to show that deobfuscation is a NP-hard problem, so it is impossible to design a general deobfuscation approach. We have listed the proper citation [A] in Section 5 (General Deobfuscator). However, it is possible to design a general obfuscator based on two limitations of current obfuscators (in Section 1), which is our main motivation of JSimpo.

[A] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. USA

Comment 2.8

Response 2.8

We add the following paragraph to explain how Lu et. al. conduct inter-procedure deobfuscation, and explain the main difference between JSimpo and their approach.

Lu et. al. also propose an approach for inter-procedure deobfuscation. They collect the dynamic execution trace to generate the dynamic Control-Flow Graph of a given JS program. Then they slice and execute the code snippets added by obfuscators to recover the original program. The main difference between JSimpo and their approach is that JSimpo uses static analysis instead of dynamic execution to find those code snippets added by obfuscators, thus avoiding executing the original JS program, which highly depends on its original execution environment.

## Reviewer: 3

Comment 3.1

How general/fundamental are the indicators that JSimpo uses to identify control-flow flattening? The text in section 3.2/3.2.1 indicates that fairly simple heuristics are used to identify state-indicator variables, but the example program in Fig. 3 appears trivially modifiable to avoid this heuristic (and it does not immediately appear obvious why an obfuscator could not make such modifications and confuse JSimpo in an automated/generalizable way). Can you clarify exactly how "unavoidable" these indicators are? If there are specific, fundamental constraints that prevent obfuscators from evading detection of control-flow flattening, they should be explained clearly here.

Response 3.1

We add the following paragraph in Section 3.2.1 to show the generalizable of our rules for finding indicators.

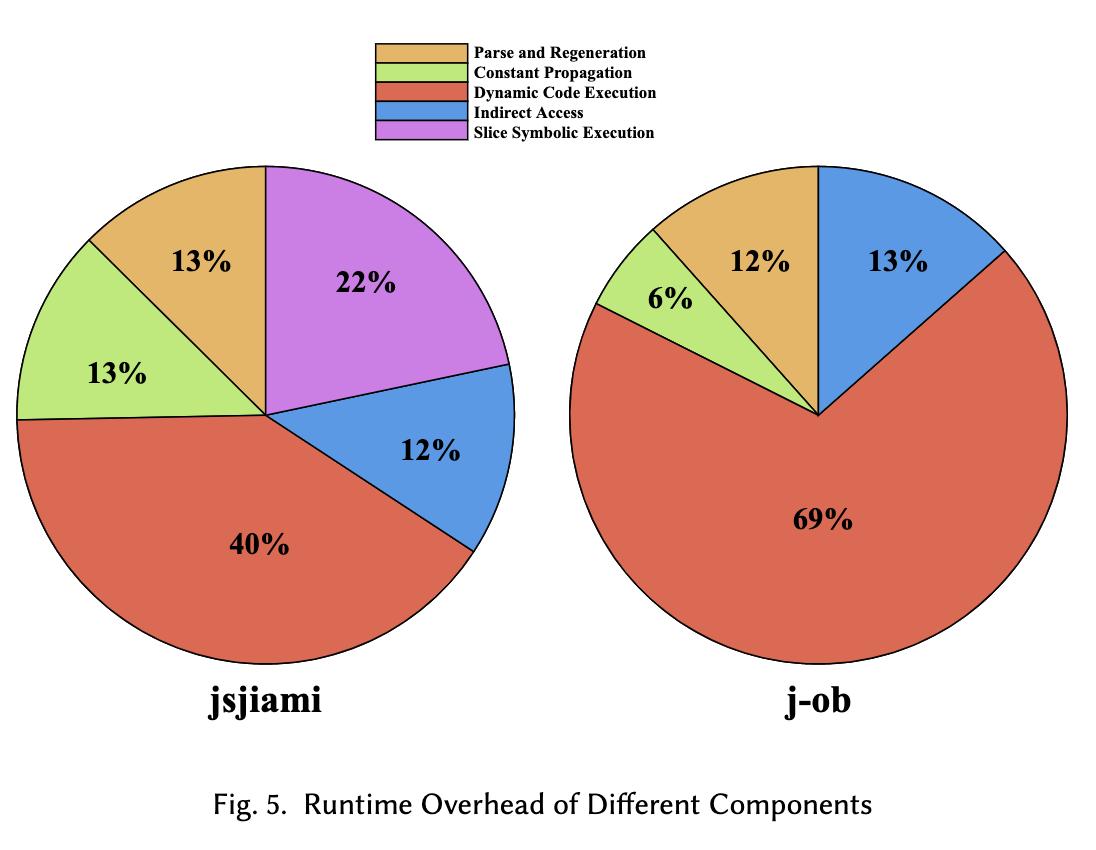
The preceding two rules are generalizable enough based on two reasonable assumptions. First, obfuscators should know the assignment and jump order of the indicators (i.e., the deterministic control-flows and data-flows). Otherwise, they cannot keep the real execution order of the obfuscated programs, thus failing to keep the semantics of the original programs. Second, obfuscators do not conduct heavy analysis on the original programs for effectiveness due to JavaScript's various language features. Thus, the control-flow and data-flow of indicators might be independent of those flows of the original programs. Considering the indicators' deterministic and independent control-flows and data-flows, we design the preceding two rules to find out the indicators added by obfuscators.

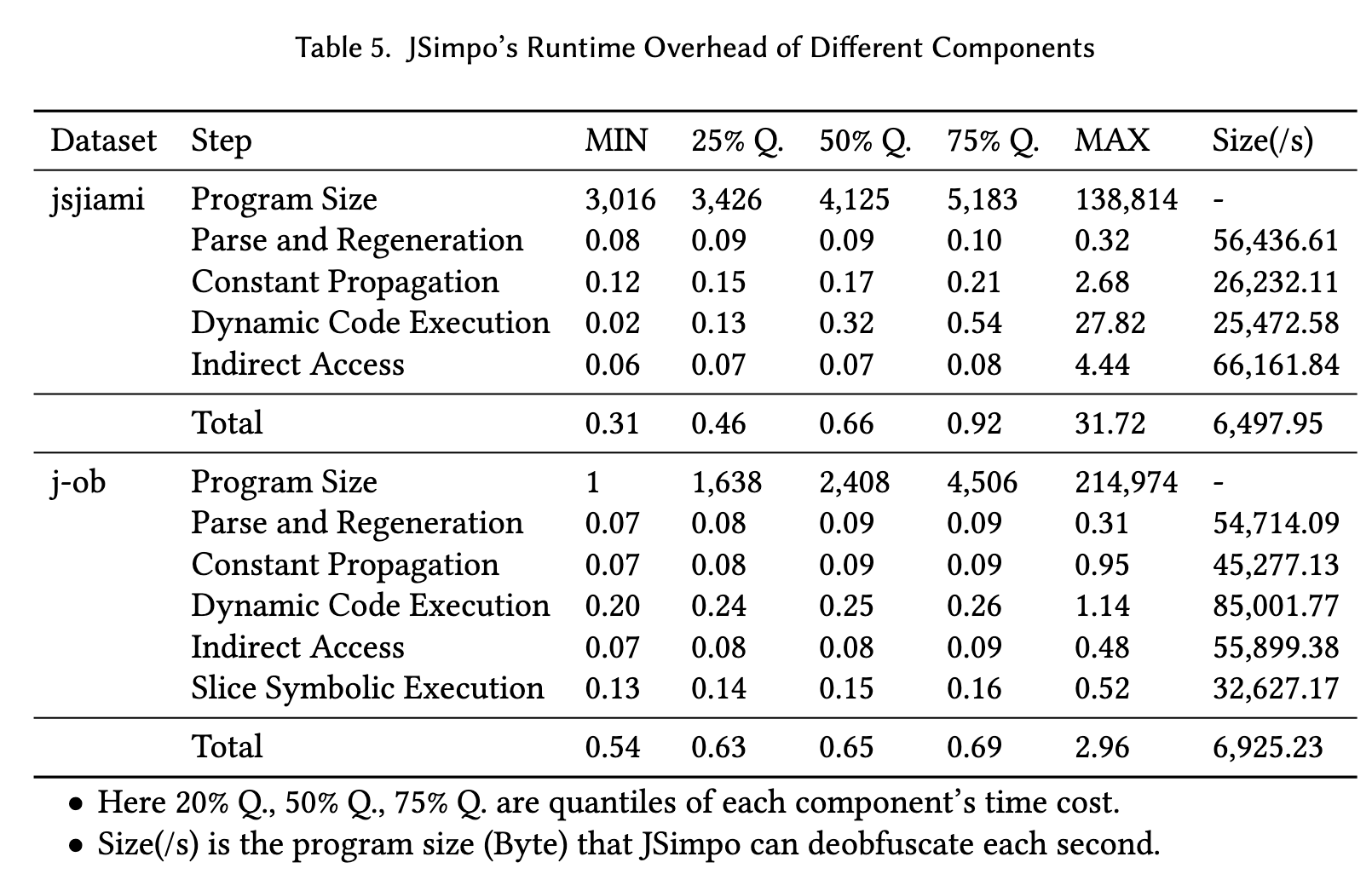
Comment 3.2

The quantification of JSimpo's runtime performance is very vague. The median/average processing time, even when compared to a worst-case time, is not a very useful metric given the extreme variability of JavaScript code size and complexity in the wild. Expressing the performance in terms of program size (e.g., average lines/second, etc.) or some other parameter would give the reader a better basis for estimating real-world performance at scale. At the very least, visualizing the performance data over the entire dataset would give the reader more understanding of the range of performance possible.

Response 3.2

We replace the pie graph of JSimpo’s runtime overhead with a new table to show its runtime overhead. We use 0, 25, 50, 70, and 100 quantiles to show each component’s runtime overhead in detail. Additionally, we add the last column to illustrate the program size that JSimpo can deobfuscate per minute, which is defined as the average number of (program size / each component’s time consumption).





## Reviewer: 4

Comment 4.1

I would ask the authors to better motivate the usage of a tool like JSimpo in practice. The experimental results are executed on OS datasets, which is fine as long as research is concerned. However, are there practical needs for such tools in practice? What can it be useful for and by whom? Why would I want to de-obfuscate a JS program, if obfuscation was used in the first place to as a defence mechanism?

Response 4.1

We add a paragraph in Section 3 to show the purpose of JSimpo, which focuses on only malware detection and comprehension.

We propose JSimpo to help developers and administrators audit the security of JavsScript applications. For example, cloud server hosts, such as our industrial partner, need to check uploaded JavaScript apps from third-party developers to ensure there are no security risks in the uploaded apps. Web developers can also use JSimpo to examine third-party libraries to avoid supply-chain threats.

Comment 4.2

The nature and selection of datasets is unclear. For the Complex-JS programs, what makes them complex? How can a file with one function be considered complex? What is the distribution of functions across these files? Moreover, what prevents programs in Normal-JS from being also "complex", if no selection criteria were used? This whole part should be explained better.

Response 4.2

We add a further explanation about the difference between the two datasets.

The average size of programs in Complex-JS is 2 times larger than those in Normal-JS. Additionally, obfuscators tend to apply many obfuscation techniques, such as control-flow flattening, to JS programs with function definitions.

Actually, we want to construct a relatively larger dataset as Complex-JS. If the reviewers are unsatisfied with the construction procedure of Complex-JS. The results of Normal-JS can also be convincing to show JSimpo’s effectiveness.

Comment 4.3

Concerning the usage of tree kernels for clone detection, concerning citing the recent work by Corazza et al. (ESEM 2021).

Response 4.3

We add the essential citation of recent work by Corazza et al. (ESEM 2021).

Comment 4.4

Table 2: why could 100% not be achieved?

Response 4.4:

We add an example JS program that is not completely recovered in RQ1 to illustrate how JSimpo recovers an obfuscated program, and discuss why the deobfuscated programs cannot achieve a tree kernel similarity of 100%.

//The original JS program.

function treeEqual(node1, node2) {

if (!node1 && !node2) {

return true;

}

if (!node1 || !node2) {

return false;

}

return (

node1.val == node2.val &&

treeEqual(node1.left, node2.left) &&

treeEqual(node1.right, node2.right)

);

}

//The program obfuscated by jsjiami and deobfuscated by JSimpo.

function treeEqual(\_0x46668a, \_0x52d93c) {

if (!\_0x46668a && !\_0x52d93c) {

return true;

}

if (!\_0x46668a || !\_0x52d93c) {

return false;

}

return \_0x46668a['val'] == \_0x52d93c['val'] &&

treeEqual(\_0x46668a['left'], \_0x52d93c['left']) &&

treeEqual(\_0x46668a['right'], \_0x52d93c['right']);

}

The preceding listing shows an example of one JS program obfuscated by jsjiami and deobfuscated by JSimpo and its original program. The tree kernel of this program is 81.22%. The main reason for not achieving a tree kernel similarity of 100% is that JS contains many equivalent expressions, e.g., node.val and node[’val’]. Additionally, JSimpo removes the extra bracket in the return expression. The preceding difference changes the structure of syntax trees, thus affecting the tree kernel similarity. We do not change these equivalent expressions for two reasons. First, it is hard to distinguish which expression is used in the original program because obfuscators change both expressions into the same format, e.g., node[’val’]. Second, these equivalent expressions do not increase the complexity of a given program and can be easily understood by security engineers.

Additionally, the changed local variable names (e.g., from node1 to \_0x46668a) cannot be recovered because the original names of local variables are not kept in the obfuscated programs.

Comment 4.5

Figure 4: I would run statistical tests to assess whether the differences are actually significant

Response 4.5

Thanks for your suggestion.

We conduct the Kruskal-Wallis test to see whether there exists a significant difference between programs deobfuscated by JSimpo and programs deobfuscated by other tools.

The results of programs obfuscated by jsjiami are listed as follows:

-------------------

Statistic: 212.8614741392281

P-value: 6.439419837426507e-45

There is a significant difference among the groups.

The results of programs obfuscated by javascript-obfuscator are listed as follows:

Kruskal-Wallis Test

-------------------

Statistic: 814.2683435730697

P-value: 6.232736364375678e-175

There is a significant difference among the groups.

Comment 4.6

Structural similarity provides good results. What about semantics? Is it retained before/after obfuscation/de-obfuscation?

Response 4.6 (Similar to Response 2.3)

JSimpo should keep the semantics of programs before and after deobfuscation.

We make conservative analysis when conducting deobfuscation. JSimpo might not find the indicators or spurious functions, and it is also possible that JSimpo might fail during slice symbolic execution and dynamic code execution. In these scenarios, JSimpo does not conduct further deobfuscation to keep the semantics of deobfuscated programs. This also explains why JSimpo cannot achieve the tree kernel similarity of 100%. Each step of deobfuscation might be given up.

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.

Another reason is that considering our purpose of deobfuscation (listed in Response 4.1), deobfuscated programs with a little different semantics can also be tolerable for malware detection and comprehension.

Additionally, review 1 also mentions that even obfuscators might change the semantics of programs, so it is possible and unavoidable that JSimpo might change the semantics of programs.

Comment 4.7

I don't get the purpose of studying the obfuscation generalization. What is the difference between RQ1 and RQ2?

Response 4.7

In RQ2, we select four other obfuscators to show JSimpo’s generalization ability while these tools are not considered when designing JSimpo.

Comment 4.8 (Miner Comments)

I'd merge the first two bullets of the claimed contributions, as they are the same

Response 4.8

Thanks for your suggestion. However, the first bullet raises the awareness of structural deobfuscation and proposes our approach, JSimpo, which is a high-level contribution, and the second bullet explains the two main techniques of JSimpo, which is a detailed contribution.