On The Impact of Atoms of Confusion in JavaScript Code

Anonymous

***Abstract*—Evolving legacy code is a challenging task, particularly when the code has been poorly written or uses confuse idioms and language constructs, which might increase maintenance efforts and impose a significant cognitive load on developers. For this reason, researchers have investigated possible sources of confusion in codebases, including the impact of small code patterns (hereafter atoms of confusion) that contribute to misunderstanding the source code written in statically typed languages such as C, C++, and Java. In this work, we investigate whether atoms of confusion identified in statically typed languages also confuse developers of a dynamically typed language. We use JavaScript as a representative example of dynamic programming languages and collect evidence from a mixed-method research effort: a survey, a set of interviews with practitioners, and an activity of mining open source JavaScript repositories (MSR). Our survey and interviews confirm that atoms of confusion lead to code that is hard to understand in JavaScript. Considering ten atom candidates, developers correctly predict the outcome of at least 15% more cases in code snippets where atom candidates are not present, compared to alternative versions that include the atom candidates. For five of these atoms, the difference is statistically significant (accounting for p-value correction). Effect size is large for four atoms and medium for one of them. In addition, our MSR effort reveals that atom candidates are frequent and used intensively in 72 popular open-source JavaScript systems. Four atom candidates appear in 90% of the analyzed projects, and two of them occur more than once for every 100 lines of code in the dataset. Altogether, our findings might help practitioners: (1) better understand the implications of atoms of confusion on understanding JavaScript code, (2) avoid writing code that is unnecessarily difficult to maintain, and (3) design program transformation tools that remove potential sources of misunderstanding in JavaScript code.**

***Index Terms*—code readability, program comprehension, program understanding, atoms of confusion, JavaScript code**

I. Introduction

Developers are often confused while reading unfamiliar code. According to Hermans [1], that confusion can stem from three sources: (i) lack of knowledge, i.e., not knowing what an element in the program does; (ii) lack of information, i.e., not knowing how a program element works; and (iii) lack of processing power, i.e., the inability to combine all the elements in a program and make sense of its execution in one's head. Even small pieces of code can be confusing for developers [2], [3] at first glance.

For instance, consider the pair of JavaScript code snippets in Listing 1 (adapted from NERVJS/TARO). The code snippet on the left-hand side is arguably harder to understand than the one on the right-hand side. The former uses the logical AND operator (`&&`) beyond its lexical meaning [4] to determine the control flow of the program. Furthermore, the expression has a potential side effect, due to the use of the post-increment operator (`s++`). The snippet on the right-hand side uses an `if` statement for control flow and increments the value of variable `s` using an assignment, making it arguably easier to understand. Notwithstanding, confusion is a direct consequence of the experience of the person who is reading the code. A developer who customarily employs the idiom on the left-hand side may find it easier to understand.

Recent work [3], [5], [6] has attempted to elicit and catalog simple code patterns and language constructs that often confuse developers when reading code and for which there are less confusing, functionally-equivalent alternatives. These confusing constructs and patterns are called "atoms of confusion" [3] when it is possible to experimentally ascertain that there is a less confusing alternative (otherwise they are called "atom candidates"). The snippet on the left-hand side of Figure 1 contains two atoms of confusion discussed in previous work [3] focusing on the C language: Logic as Control Flow and Post-increment. Previous work has identified atoms of confusion in two statically-typed languages, C and Java. According to the studies of Gopstein et al. [3], removing the atoms Logic as Control Flow and Post-increment from small C code snippets improved the ability of study participants to predict their outcomes by 41% and 34%, respectively. For small Java code snippets, Langhout and Aniche [6] report improvements of 53% and 46.27%, respectively. These results show that atoms of confusion may be **challenging** for developers using these languages. In addition, a study [7] with 14 large-scale open source projects written in C found that 4.38% of the lines of code have an atom and their presence has a strong correlation with bug fixing commits and long code comments. These results show that atoms of confusion are **prevalent** in large systems. They are also **relevant**, even for experienced software developers.

The goal of this paper is to investigate whether atoms of confusion that were identified in statically typed languages also cause confusion to developers of a dynamically typed language. We leverage JavaScript as a representative example of dynamic programming languages. JavaScript's programming culture differs from that of languages such as C and Java, particularly due to its dynamic capabilities and weaker type system. At the same time, the language is not so different that previously identified atoms cannot be represented in it. We present the results of a mixed-method research effort based on a survey, on interviews, and on a mining software repositories study. First, the survey's instrumment asks the participants to predict the output of small code snippets, with half of them

```
res = properties.reduce((s, b) => {
  b && s++
  return s
}
```

```
res = properties.reduce((s, b) => {
  if(b) {
    s = s + 1
  }
  return s
}
```

Listing 1: Code snippet from NERVJS/TARO project (left-hand side) and an alternative implementation (right-hand size).

containing atom candidates. Subjects were organized according to a Latin Square design [8] to control for the experience of the subjects. To learn more about the misunderstandings in the snippets used in the survey, we also interviewed 15 experienced professional developers. Participants preferred the code snippets without the atoms in 70% of the cases. The frequency with which the atoms appear *in the wild* might reveal cases where developers should be more or less careful when writing JavaScript code. To measure the prevalence of the atoms, we mined popular JavaScript projects from GitHub and found that atoms frequently appear in practice. Our MSR effort reveals that atom candidates are frequent and used intensively in 72 popular open-source JavaScript systems.

Altogether, the main contributions of this paper are:

- A mixed-method research effort—based on a survey, interviews, and mining software repositories—about the impact of atoms of confusion on JavaScript (a dynamic language), controlling for subject experience.
- A comparison of the impact and prevalence of atoms of confusion in JavaScript with what has been previously reported for other languages (C, C++, and Java).
- A library and infrastructure to mine atoms of confusion using code queries (implemented in the .QL language). This library and infrastructure are publicly available and might help researchers and practitioners to automate program analysis at a large scale.

II. RELATED WORK

The concept of program comprehension is central to software maintenance and feature development [9], [10]. The study of program comprehension in the presence of atoms of confusion was introduced by Gopstein et al. [7], who defined them as small code patterns that can verifiably lead to misunderstandings. In addition, for a code pattern to be considered an atom, there must be some alternative pattern or language construct that is functionally equivalent and less likely to cause confusion. One example that occurs in many programming languages is the Change of Literal Encoding. For instance, the C code statement `printf("\%d", 013);` often leads programmers to predict the output to be `13`, even though the correct answer is `11`. This occurs because a leading `0` in a numerical literal indicates that the number is in base `8`, a fact that is not only unknown to less experienced programmers, but misleading even for seasoned developers. After formulating a list of 19 atom candidates, Gopstein et al. [3] were able to identify 15 code idioms and language constructs that present a statistically significant difference in comparative answer correctness when each of the 15 atoms was removed. The least confusing atom produced a 14% boost in prediction accuracy when it was removed, whereas the most confusing one showed a 60% accuracy increase. In a different work, Gopstein et al. [7] presented the results of a comprehensive study on the incidence of atoms of confusion in the wild, considering open-source projects written in C and C++ programming languages.

These previous studies [3], [7] have a great influence in our work. Even though JavaScript is a dynamic language that differs from C in a variety of aspects, it also has a number of constructs in common. In addition, the imperative aspects of the language are syntactically similar to C, e.g., assignments, conditional expressions, `if`-statements, pre and post increments and decrements, among others. On the one hand, this means that some of the atoms of confusion that exist in C programs may also occur in JavaScript code. On the other hand, differences between the languages and the surrounding programming cultures may lead to code patterns that are atoms of confusion in one language not being atoms in the other one. Compared to previous research, in this work we leverage the *Latin square* experimental design to control both participant experience and education better, while investigating the effect of the atom candidates in code comprehension. We present details about our setup in Section III. Differently, previous research works (e.g., [3], [6]) use a *Randomized Partial Counterbalanced Design*, which does not *block* relevant characteristics of the participants.

Oliveira et al. [11] conducted an experiment using an eye-tracker with 30 participants involving three atoms in C. They found that code with atoms of confusion requires more time from participants to predict the output and more visual effort to comprehend. Gopstein et al., in another paper [12] further explored the original atoms by additionally conducting interviews and having programmers discuss among themselves the studied atoms. The authors argue in favor of complementing a quantitative analysis. Indeed, their study revealed findings such as a *"correct evaluation of an atom might not mean that a programmer understood its meaning"*. Here, we also complement the quantitative analysis with interviews.

Langhout and Aniche [6] derived a set of 14 atoms for Java and performed an experiment with 132 students. For seven atom candidates (out of 14), they report that participants are more likely to make mistakes in the confusing version of the code. Castor [4] presented a preliminary catalog of six atom candidates for the Swift programming language. Unlike JavaScript, in Swift, most of the atoms identified by Gopstein et al. [3] are avoided by construction, e.g., it does not have

assignments as values, increment operators, or macros.

Medeiros et al. [5] analyzed 50 open source projects written in the C language with the goal of evaluating 12 code patterns called "misunderstanding patterns" by the authors. Many of these misunderstanding patterns are either atoms of confusion or atom candidates [3]. This study shows that these patterns are prevalent; among these 50 projects, there are more than 109K occurrences of misunderstanding patterns. In order to gauge the relevance of these misunderstanding patterns, the authors sent 35 pull requests removing occurrences of these patterns to randomly-selected open source projects. The authors of the study received feedback for 21 of these pull requests and the maintainers of the projects accepted 8 of them (22.86%).

III. Methodology

The main goal of this research is to investigate the impact of atoms of confusion on JavaScript code comprehension. As such, in this paper we answer the following research questions:

(RQ1) What is the impact of atoms of confusion on the comprehension of JavaScript code?

(RQ2) Do JavaScript developers identify atoms of confusion as contributing to program misunderstanding?

(RQ3) What is the frequency of occurrence of atoms of confusion in practice (i.e., in open-source JavaScript projects)?

(castor note) This one will probably be removed.

By answering these research questions, we can either generalize or refute the findings about atoms of confusion already discussed in the literature (goal of research questions (RQ1) and (RQ2)).

(castor note) The following pertains to the third RQ.

In addition, answering the third question allows us to enrich existing catalogs about atoms of confusion and discuss how often they occur in practice

(castor note) This should probably be removed.

. Answering these research questions also lays the foundations for the implementation of tools that can automatically transform code into cleaner versions. We conduct a mixed-methods study to answer these questions. It includes two independently designed and conducted experiments: a repeated measures/within subjects study and one using a Latin square, counterbalancing experimental design to control for subject experience. By using two different experiments, designed by different researchers, with different designs, and conducted with different participants, we hope to rigorously verify the impact of atoms of confusion in code comprehension activities. These two experiments explore the impact of atom candidates on understanding JavaScript code (goal of research question RQ1). They investigate whether or not programs that contain atom candidates tend to produce more misunderstanding for programmers trying to predict their outcome.We also conduct a set of interviews with developers to contrast the quantitative results of the experiments with their preferences and opinions (thus addressing RQ2).

A. Repeated Measures Study

(castor note) We must have introduced the atoms before.

The first experiment we perform uses a repeated measures, or within-subject, design [13]. This experiment is inspired by the work of Gopstein et. al [3], but targeting JavaScript instead of the C language, and including some methodological differences. It covers all the atom candidates analyzed in the work of Gopstein et al. that can be transposed to JavaScript and adds candidates that are specific to JavaScript. Overall, we select 24 atom candidates in the repeated measures study.

Experimental design: In this experiment, the control variable consists of a tiny program containing a single atom candidate and the treatment is a functionally-equivalent version that does not contain the atom. The main dependent variable is whether subjects are able to correctly determine the output of the programs. The other dependent variable is the time required to correctly vs. incorrectly determine the outputs, independently of the presence of atoms. We use the same null hypothesis as Gopstein et al.: *"code from both control and treatment groups can be hand-evaluated with equal accuracy"* and the alternate hypothesis that *"the existence of an atom of confusion causes more errors than other code in hand-evaluated outputs."*. For each atom candidate, we built six programs, three versions including the atom candidate and three versions not including it.

The design of the repeated measures study considers two treatments: the presence or absence of atom candidates within the programs. To identify atom candidates, we start out by selecting every atom candidate from Gopstein's study that can also happen in JavaScript. That excludes atoms related to pointers and preprocessor directives. Furthermore, we conduct an informal search of programming forums, solutions to Code Golf [1] problems, code style guidelines, and expert knowledge to identify new atom candidates. Overall, the repeated measures analyzes 24 atoms, 15 from the original study and 9 that are new and specific to JavaScript. Table **??** presents the complete list.

For each atom candidate, we write six short programs, three including and three not including the atom candidate. The former are the control for the study, or the **obfuscated** versions of the programs. The alternative, atom-free, versions of the programs are the treatment, or **clean**, versions. Each subject is exposed to one obfuscated and one clean version of each atom, totalling 48 programs, and should determine their outputs. As mentioned before, we measure answer correctness and the time to answer questions correctly vs. incorrectly. We control for learning effects [14] in three manners. First, by having multiple obfuscated and clean versions for each atom and presenting only one of each per subject. Second, by presenting the programs in a random order. Third, by presenting obfuscated and clean versions corresponding to the same atom candidate with at least 11 other programs in between, in accordance to the original experimental protocol [3].

[1] https://codegolf.stackexchange.com/

Study instrument: The study is conducted by means of a web application. As part of this effort, we carried out an informal pilot whose main objectives were (i) to spot bugs in the application and in the data collection mechanism; (ii) to gain feedback from respondents about the user experience of the application; and (iii) to formulate an estimate about how long answering the survey would take on average. Based on the feedback of the participants of the pilot, we present only one obfuscated and one clean version of each atom candidate, to reduce to total time of the study and potentially increase participation.

We organize the study in three sections. In the first one we present instructions and also a check button whose checking means users agree that all collected data will be used anonymously and solely for research purposes. The instructions explain how the survey works and asks them to dedicate their attention to it. We stress to participants the importance of not using any aids during the survey, such as online or console interpreters.

The second section presents the programs, one at a time. For each one, there is a text box where the answer should be written and a "Next" button. If the subject leaves the text box empty and presses "Next", this is treated as a wrong answer. To reduce the likelihood of a subject attempting to execute the programs, the web application verifies whether the subject tries to change tabs or windows and presents a pop-up message if that is the case. Furthermore, it presents the programs as images, instead of text, to demotivate respondents from resorting to external resources by copying and pasting the code into an interpreter. Upon submitting an answer for a particular program, the subject is automatically led to a similar page, containing the next one.

We do not provide feedback about the time subjects take to answer each question. We also do not tell them whether their answers are correct or not. This aims at avoiding introducing bias for future respondents. Since we posted the survey on social media platforms, possible threats could have arisen if we gave respondents instant feedback.

Study audience: We posted the survey on the authors' social media platforms, on Brazilian CS departments' mailing lists, and on two programming subreddits. We explained our research purposes and asked developers to take the survey only if they have some familiarity with JavaScript. We collected 70 responses.

B. Latin Square Study Setting

This experiment uses a different experimental design that makes it possible to control for the experience of the subjects. It employs a subset of the atom candidates used in the repeated measures study. More specifically, we select a set of 9+1 atom candidates: nine previously-validated atoms of confusion for the C language [3] that also exist in JavaScript programs plus one atom candidate that is specific to the JavaScript language (Automatic Semicolon Insertion).

Experimental design: The design of our second study blocks two variables (subject experience and the programs) and considers two treatments: the presence or absence of atom candidates within the programs. To achieve such a design goal, controlling the effect of experience and individual programs, we resort to the *Latin Square Design* [8]. Using this design we create a 2 x 2 matrix in which each row represents a subject and each column indicates the set of programs. The design of each square (a replica) is such that no treatment is repeated in the same row or column. For example, considering that we have a set of 10 code samples $s_0, s_1, ..., s_{10}$, if a given subject (P1) is asked to predict the output of the code samples $s_0, s_1, ..., s_5$ that contain atom candidates, then, when answering questions about clean programs, they will only be presented with clean versions of the code samples $s_6, s_7, ..., s_{10}$. Furthermore, a given subject (P2), who constitutes the second row of our example square, will be asked questions about the clean versions for $s_1, s_2, ..., s_5$, and will answer questions about obfuscated programs for $s_6, s_7, ..., s_{10}$. By doing that, we guarantee that all versions of the programs are contained within each square, and that each configuration occurs only once within a square. Figure 1 offers a visual representation of the concept.

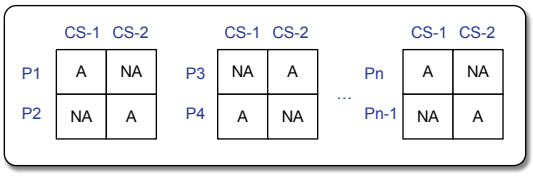

Fig. 1: Latin square design. Each "square" corresponds to a replica in our study. Each replica comprises two participants (square rows, e.g., P1 and P2) and two sets of programs (CS-1 and CS-2). We randomly apply the treatments (atom or non-atom code) to the cells of the squares.

For each of the 10 selected atom candidates, we write one short program containing the atom. As in the repeated measures study, we call them the obfuscated versions of the programs. We also write corresponding short programs without the atom candidates and call them the clean versions of the programs. Overall, the experiment employed 20 programs, 10 obfuscated and 10 clean. In order to reduce the cognitive effort, each subject was asked to predict the output of 10 programs, five obfuscated and five clean. The order in which they are presented is randomized. By doing this, we seek to minimize the chances of subjects being aware that the current listing they are analyzing contains (or not) atoms of confusion. That is, each subject should indicate what would be the outcomes of the programs, some of them having atoms of confusion (while other programs do not). Since participants are not exposed to obfuscated and clean versions corresponding to the same atom candidate, learning effect is not possible. We measure answer correctness and the total time each participant requires to participate in the experiment.

Study instrument: We implemented our survey as a web application and carried out a pilot. Undergraduate students and professional colleagues took the pilot survey. Some users

reported layout defects, and many reported that the landing page did not explain the survey well enough. We also spotted minor issues with our routines to create and populate the latin squares.

We organize the survey in three sections. The first section aims to characterize the subjects, asking their age, education level, and programming experience. We also include a check button, whose checking means users agree that all collected data will be used solely for research purposes. In the second section, we present to the participants a small set of instructions, where we explain how the survey works and ask them to dedicate their attention to it. We stress to participants the importance of not using any aids during the survey, such as online or console interpreters. For each question page, we kept track of whether or not the subjects switch windows.

The last section of the survey presents a sequence of ten questions, each containing a program. For each question, there is a text box where the answer should be written. There is also an "I do not know" button, which, when clicked, leads the subject to the next question. In our setting, "I do not know" is treated as a wrong answer. The programs are also presented as images copied from a text editor. Upon submitting their answer for a particular question, the subjects are automatically led to a similar page, containing another program. Similarly to the repeated measures study, we do not provide feedback about time and correctness to the subjects.

Study audience: We posted the survey on a JavaScript Reddit channel.[2] We explained our research purposes and asked developers of any level of expertise to take the survey. Within twelve hours, we collected more than 150 answers, populating more than 70 replicas of the Latin Squares. We collected significant data on time taken and discrepancies in answer correctness between obfuscated and clean versions of the programs. Some inconsistencies arose while building the squares, for instance, when a user quit in the middle of the survey. We discarded from our study all squares that contained incomplete rows (a common approach for data imputation).

C. Interview Study Setting

To complement the two experiments, we perform semi-structured interviews with professional JavaScript developers, aiming to identify their perceptions regarding programs containing atoms of confusion (research question RQ2). We also ask each participant if they know of any other JavaScript-specific construct or idiom that they regard as being likely to make the code hard to understand.In this section we detail the protocol we followed to conduct the interviews and to analyze the results.

Participant Selection: We invite the participants of the interviews using a snowballing technique. That is, starting from our network of contacts, we invite an initial set of candidates to take part in our survey. From this initial list, we ask for an indication of additional candidates. Our main selection criterion is that all participants should have been working with JavaScript in their daily professional activities. We invited a total of 17 developers, and 15 of them agreed to participate.

Interview Process: We conduct semi-structured interviews using web conferencing software. We record all the interviews with the consent of the participants. On average, the interviews last 26.29 minutes, with the shortest one lasting 14.59 minutes and the longest one 43.06 minutes. Two of the researchers conduct the interviews, and a third one listens to all the recordings to cross-validate the collected data. The interviews have three main parts. In the first one, we ask the developers the following demographic information: name, email, gender, level of education, current job position, JavaScript experience in years, and other programming languages they have worked with. Table I summarizes this demographic information.

TABLE I: Demographic information of the participants

ID	Education	JS Experience	Other Languages
P1	BSc Degree	9 years	Java, PHP, C, Go
P2	HS Degree	3 years	Python, Go, Dart, Lua, C++, C#
P3	BSc Degree	4 years	Java, C
P4	Undergraduate	3 years	Python, C, C++, Java, Go
P5	Master Student	3 years	Python, SQL
P6	Bsc Degree	15 years	Java, PHP, C, Python , Ruby, C#
P7	BSc Degree	6 years	C, C++, Java, Assembly, Kotlin
P8	PhD Degree	2 years	Java, Python
P9	BSc Degree	5 years	PHP
P10	Master Student	4 years	Java, C# and Python
P11	Master Student	4 years	Java, Erlang, C#, Cobol
P12	Master Degree	1 year	C
P13	Master Degree	13 years	Java, PHP
P14	Master Student	3 years	C, Python, Ruby
P15	BSc degree	2 years	Java, PHP

In the second part of the interview our aim is to allow the subjects to describe their JavaScript experience, as well as to allow them to reveal any JavaScript constructs they regard as innately confusing. This allows us to identify potential atom candidates that are more specific to the JavaScript language. Examples of questions we explore in this section include: *Does JavaScript favor developers to produce code that is hard to understand?* and *Do you regard any particular construct or idiom of the language as especially confusing?*

In the third part of the interview, participants are shown pairs of programs that were used in the first study, where each pair contains a clean and an obfuscated version of the same atom candidate. The participants are asked to evaluate which version of the code is easier to understand. To avoid introducing bias in the answers, the interviewers do not explain that one of the versions in each pair contains the atom under investigation. Subjects are just presented the pairs of programs and allowed to take the necessary time to decide on the most readable one.

Interview Analysis: We first transcribe each interview and then examine the broad distribution of the answers. Our goal is to build an initial understanding of the participants' perceptions with regards to the challenges to understand JavaScript code in general and JavaScript code with atoms of confusion in particular. We follow-up with an open-coding procedure,

[2]https://www.reddit.com/r/javascript/

highlighting the main themes and quoting the answers of the participants. We present these results in Section VI.

D. Mining JavaScript Software Repositories Setting

To understand how often the analyzed atom candidates appear in open source projects, and thus answer our fourth research question (*What is the frequency of occurrence of atoms of confusion in practice?*), we mined a set of GitHub open source repositories. To this end, we first collected the most popular GitHub repositories that are primarily written in JavaScript. We measured popularity using the project's stargazers. This metric, available through the GitHub API, represents the number of stars a project received from users of the platform. The same metric has been used in a number of previous studies as a proxy to estimate a project's popularity [15], [16]. We then selected the top 100 most popular repositories and removed projects that did not reach the first quartile of the distribution of lines of code.

After filtering out JavaScript project candidates, in the second step we built a curated dataset comprising the top 72 repositories. Examples of projects in this dataset include REACT, NODE JS, and ANGULARJS. Table II presents some statistics about the projects we consider in our research. The size of the projects range from small ones (5543 lines of code) to complex systems with more than 1 MLOC. All projects in our dataset have at least 1244 forks and at least 23 672 stars. We automated all the steps to filter, clone, and collect the statistics from the repositories using Python scripts.

We mined the occurrence of atom candidates from the repositories in our curated dataset using source code queries that we wrote using the CodeQL language [17]. CodeQL is an object-oriented variant of the Datalog language [18], and currently supports researchers and practitioners to query the source code of systems written in different languages (such as C++, Java, and JavaScript). We also automate the process of running the queries and exporting the results to a format that simplifies our analysis (and also the reproduction of this study). Finally, we computed some descriptive statistics to measure the prevalence of atoms of confusion in practice.

TABLE II: Some descriptive statistics about the projects used in the MSR study

	Min.	Median	Mean	Max.
Lines of Code	5543	36 161.5	111 432.33	1 278 405
Num. of Forks	1244	6078	8906.24	68 849
Num. of Contributors	6	285.5	533.44	4047
Num. of Stars	23 672	34 990	46 919.51	310 935

IV. RESULTS OF THE REPEATED MEASURES STUDY

In this study, we investigate whether 24 atom candidates impact readability. We examine how the performance of the participants differ when predicting the outputs of the clean and obfuscated versions of small code snippets (Section IV-A). Based on those results we establish which candidates can actually be considered atoms of confusion. In addition, similarly to Gopstein et al. [3], we verify whether there is a trade-off between correctness and time irrespective of the presence of atoms; we check if participants who correctly predict the outputs of the programs tend to do so more slowly than the ones that make incorrect predictions (Section IV-B).

TABLE III: Summary of the correctness analysis.

Atom	Confusing	Clean	Δ(%)
Type Conversion	5.0	27.0	+440.0
Change of Literal Encoding	9.0	36.0	+300.0
Comma Operator	8.0	29.0	+262.0
Object Destructuring	15.0	42.0	+180.0
Assignment As Value	18.0	43.0	+139.0
Pre-Increment	13.0	28.0	+115.0
Post-Increment	8.0	17.0	+112.0
Array Destructuring	15.0	28.0	+87.0
Indentation no Braces	30.0	50.0	+67.0
Omitted Curly Braces	35.0	51.0	+46.0
Object Spread	21.0	30.0	+43.0
Array Spread	22.0	31.0	+41.0
Automatic Semicolon Insertion	15.0	21.0	+40.0
Infix Operator Precedence	47.0	53.0	+13.0
Arrow Function	54.0	58.0	+7.0
Implicit Predicate	41.0	44.0	+7.0
Indentation With Braces	53.0	55.0	+4.0
Conditional Operator	60.0	62.0	+3.0
Constant Variables	59.0	61.0	+3.0
Dead Unreachable Repeated	63.0	64.0	+2.0
Repurposed Variables	23.0	23.0	0.0
Arithmetic as Logic	52.0	47.0	-10.0
Logic as Control Flow	33.0	28.0	-15.0
Property Access	51.0	42.0	-18.0

This study had 70 participants. Out of these, 67 have received university-level training in programming, 4 hold a PhD degree, 7 hold a master's degree, and 11 hold a bachelor's degree. The median programming experience of the participants is four years (mean 6.5 years) and the median experience with JavaScript programming is one year (mean 2.5 years).

A. Correctness Analysis

Exploratory Data Analysis. : For each of the 24 atom candidates, the participants predicted the outputs of two versions, one clean (control) and one obfuscated (treatment), as per the definition of a repeated measures design [8]. Thus, each participant examined 48 code snippets. We also measured the time required by the participants to provide their answers. For most cases, participants made more mistakes when attempting to predict the outcomes of the obfuscated versions.

Table III presents the number of correct predictions for the clean and obfuscated versions of each atom candidate, as well as the percentage difference. In only three cases the participants predicted results correctly more often for the obfuscated versions, Arithmetic as Logic, Logic as Control Flow, and Property Access. The former two were empirically validated as atoms of confusion in the study of Gopstein and colleagues [3]. For seven of the atom candidates, the number of correct predictions for the clean versions is more than twice the corresponding number for the obfuscated versions. Some of these atoms are common programming patterns in JavaScript programs, such as pre- and post-increments and object destructuring. The difference was even greater for Type Conversion (+440%) and Change of Literal Encoding (+300%).

Statistical analysis: Since this study uses a within-subject design, the data is paired and dependent. Our goal is to verify whether there is a significant difference in the performance of the same participants when examining clean and obfuscated versions of programs. The analyzed data is also dichotomous, since each participant either correctly predicts the output of a program or does not. Considering this scenario, we employ McNemar's test [19], which is aimed at *"judging the significance of the difference between correlated proportions"*.

The "McNemar test" column of Table IV indicates the p-values for the tests. These p-values are compared against an alpha of 0.0021, after applying Bonferroni correction (0.05/24). One of the atom candidates, Constant Variables, is omitted from the table because, due to the very low number of mistakes, it is not possible to perform the test. The table shows that eight of the 24 analyzed atom candidates can be considered atoms of confusion based on the results of our study, Object Destructuring, Type Conversion, Change of Literal Encoding, Comma Operator, Indentation No Braces, Assignment as Value, Pre-Increment, and Ommitted Curly Braces. Among these eight, only Object Destructuring is JavaScript-specific. The other seven atoms have been previously reported for the C language [3] and also for Java [6] (with the exception of Comma Operator). For some candidates that have been previously identified as atoms, there was not a statistically significant difference. This is the case for Repurposed Variables, Logic as Control Flow, Implicit Predicate, and Conditional Operator.

We leverage the odds ratio (OR) as a measure of effect size, to quantify the meaningfulness [20] of these results. The odds ratio is an appropriate measure of effect size for scenarios where McNemar's test is applicable [21]. For example, odds ratio for Type Conversion is 12. This suggests that the odds of a participant correctly predicting the output of a clean version while incorrectly predicting the output of the corresponding obfuscated version is 12 times higher than the odds of correctly predicting the output of the obfuscated version and incorrectly predicting the output of the correct one.

According to the thresholds established by Chen and colleagues [22] (OR = 1.68 small, 3.47 medium, and 6.71 large), for most of the atoms where there was a statistically significant difference the odds ratio can be considered high. The value of the odds ratio for Object Destructuring is ∞ because no participant correctly guessed the output for the obfuscated version while incorrectly guessing the output for the clean one, which results in a denominator of 0. Omitted Curly Braces exhibits an OR of 6.3, which is borderline between medium and high [22]. Array Destructuring and Post-Increment are two atom candidates that deserve a special note. The difference in the performance of participants examining clean and obfuscated versions of these atoms was not statistically significant, after correction. Notwithstanding, effect sizes were large and medium, respectively. This is a result that hints at practical relevance in spite of the p-values [20]. Further investigation is required for these two atom candidates.

TABLE IV: Hypotheses Testing (correctness). Asterisks (*) indicate a statistically significant difference. Using Bonferroni correction, a p-value is considered statistically significant if it is lower than 0.0021.

Atom candidate	McNemar test	odds ratio
Object Destructuring	$<$ **0.0001***	∞
Type Conversion	$<$ **0.0001***	12.0
Change of Literal Encoding	$<$ **0.0001***	28.0
Comma Operator	$<$ **0.0001***	11.5
Indentation No Braces	$<$ **0.0001***	11.0
Assignment As Value	$<$ **0.0001***	9.3
Pre-Increment	**0.0003***	16.0
Omitted Curly Braces	**0.0009***	6.3
Array Destructuring	0.0023	7.5
Post-Increment	0.0225	5.5
Array Spread	0.0490	3.3
Object Spread	0.0636	2.8
Property Access	0.0636	2.8
Infix Operator Precedence	0.1796	2.5
Arrow Function	0.2891	3.0
Arithmetic As Logic	0.3323	1.8
Automated Semicolon Insertion	0.3915	1.4
Logic As Control Flow	0.4421	1.5
Implicit Predicate	0.6900	1.3
Indentation With Braces	0.7266	1.7
Conditional Operator	0.7905	1.3
Dead Unreachable Repeated	1.0000	1.5
Repurposed Variables	1.0000	1.0

B. Time Analysis

Exploratory Data Analysis: In this section we investigate whether there is a trade-off between correctness and time required to predict program outputs, similarly to the analysis performed by Gopstein et al. [3]. For this analysis, we consider only whether participants correctly predict the outputs of programs or not, independently of the presence or absence of atoms, and the time required to do so. The null hypothesis we investigate is that there is no significant difference in time between correct and incorrect predictions. Intuitively, we would expect correct responses to take longer, on the basis that participants would spend more time carefully analyzing the programs before providing an answer.

Table V presents the mean and median times required by participants to provide answers. The data is based on the 1648 correct and 1572 incorrect answers collected in the study. Contrary to our intuition, the mean time required by participants to provide an incorrect answer is 25% greater than the time required to provide a correct one and the median time is 20%. greater. The standard deviations suggest that there is great variation in the times required by different participants, for different code snippets. Notwithstanding, the standard deviation for incorrect answers is 62.58% greater when compared to the same statistic for correct answers.

Statistical Analysis: We employ the non-parametric Mann-Whitney U test, also known as Wilcoxon's rank-sum test, to verify the aforementioned null hypothesis, i.e., to check whether the times for correct and incorrect answers can be considered identical. That test produces a p-value of **0.00000046**, which shows that the difference is statistically significant and

TABLE V: Time in seconds required by the participants to correctly and incorrectly predict the outcomes of the programs.

	Mean	Median	Std. Dev.
Correct	35.44	24.36	38.43
Incorrect	44.50	29.29	62.48

we can reject the null hypothesis. To gauge the magnitude of that difference we employ Cliff's Delta as a measure of effect size. This produces a result of **0.1027**, which suggests that, although participants tend to provide correct predictions more quickly, they do that only slightly so.

V. Results of the Latin-Square Study

In the latin-square study, we estimate the impact of atoms considering two perspectives: *misunderstanding rate* (number of wrong answers) and *time* (how long to provide a correct answer). In particular, in the time analysis, differently from the repeated measures study, we compare the time required by participants to submit a correct response when analyzing code snippets with and without atom candidates. This study is based on the responses of 140 participants (a total of 70 replicas). All participants had taken at least some university course or hold a bachelor degree or equivalent. In addition, 21 participants hold a master's degree and three a doctorate degree. Considering the programming experience of our respondents, 19% have more than ten years of programming experience, 37% have between four and ten years of experience, 37% have between one and four years of experience, and 7% have less than one year of programming experience. Accordingly, we characterize the effect of atoms of confusion in JavaScript code taking into account the perceptions of both novice and experienced developers.

A. Misunderstanding Rate Analysis

(castor note) We need to be uniform about terminology. Are we going with "misunderstanding rate analysis"? The problem is that it is not clear to me to which rate this refers.

Exploratory Data Analysis: As mentioned in Section III-B, each participant in this study evaluated ten code snippets, from which five were in their confusing versions, whilst the other five contained clean versions of the code snippets (i.e., without the atom candidates). As in the repeated measures study, the participants should provide the expected outcomes of the code snippets. Differently from that study, in this one each participant only predicts the outcome of one version of each code snippet, either clean or obfuscated. As discussed, we collected information about *correctness* (whether the participant correctly predicted the program's output) and *time*. Table VI and Figure 2 summarize the results of the survey w.r.t. correctness.

Considering Table VI, the clean versions of six code snippets present at least a 15% improvement in answer correctness when compared with the confusing versions. In particular, the presence of the *Comma Operator* atom exhibits the highest impact on misunderstanding. Frequently used constructs and idioms, such as *Post Increment* and *Omitted Curly Braces* (see Section VII), also result in many mistakes. The boxplot of Figure 2 shows a non-negligible decrease in the average number of incorrect answers when observing the clean versions of the code snippets. Also, the sample of responses with no atoms had almost no dispersion, which supports the argument that the non-confusing versions of the code snippets are easier to evaluate correctly.

TABLE VI: Summary of the correctness analysis

Atom	Confusing	Clean	Δ(%)
Comma Operator	28	65	+132
Automatic Semicolon Insertion	32	68	+112
Post-Increment	48	64	+33
Omitted Curly Braces	47	58	+23
Assignment as Value	56	68	+21
Implicit Predicate	58	68	+17
Logic as Control Flow	41	48	+17
Conditional Operator	60	66	+10
Pre-Increment	50	53	+6
Arithmetic as Logic	64	63	-2

Statistical analysis: We first use the *Pearson's Chi-squared test* to investigate if there is a statistically significant difference in the frequency of correct and incorrect answers—due to the versions of the code snippets (confusing and clean code). The p-values for these tests are reported in the "Chi-square test" column of Table VIII. The results indicate that for five atom candidates (Comma Operator, Automatic Semicolon Insertion, Post-Increment, Assignment as Value, Implicit Predicate) the confusing versions of the code snippets have a negative impact on code understanding (p-value < 0.05). This result holds even after applying the Benjamini-Hochberg correction with a false discovery rate of 5%.

We measure the effect size of the clean version of the code snippets into the answers' correctness using the *Odds Ratio* (OR). The results are reported in the "Odds Ratio" column of Table VIII. In the table, we also report the confidence interval (CI) for the OR in the "CI" column. Although many of the intervals are wide, for six of the atom candidates the lower bound of the confidence interval is greater than or equal to 1. This indicates that, at a 95% confidence level,

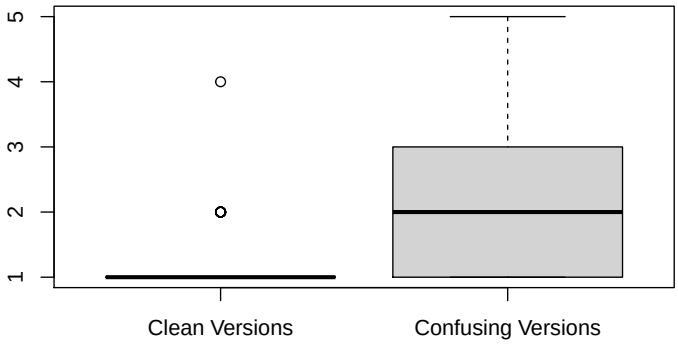

Fig. 2: Number of wrong answers of each subject.

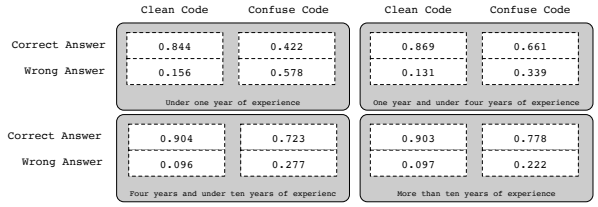

Fig. 3: Impact of atoms of confusion on the correctness (over the four participant experience groups)

a developer is likely to commit less mistakes when using the clean versions. For four of the atoms, Comma Operator, Automatic Semicolon Insertion, Assignment as Value, and Implicit Predicate, effect size can be considered large. For Implicit Predicate, it is medium. For instance, when comparing clean and confusing code snippets pertaining to the Comma Operator atom candidate, we observed an *Odds Ratio* of 19.02. This means that the odds of a correct answer are 19.02 times higher when interpreting the clean version—in comparison with the corresponding confusing version of the code snippet. Furthermore, with a 95% confidence level, the odds ratio is between 6.6 and 68.22, i.e., although the error margin is wide, it is strongly in favor of the clean version. If there is a significant likelihood of a developer committing a mistake when analyzing a clean version, the lower bound of the CI should be a number between 0 and 1 (since the OR is a ratio). This is the case for the four atom candidates in the lower part of the table.

Finally, we also use a *Binomial Generalized Logistic Regression* analysis to investigate if either *participant education* or *participant experience* impact correctness. The findings suggest that *participant experience* impacts on the results related to correctness (Figure 3). Even though the presence of atoms of confusion reduces the number of correct answers for all *experience groups*, this effect is not uniform. For instance, novice developers (under one year of experience) provide a higher number of wrong answers for the confusing version of the code (57.8%), while developers with more than four years of experience provide more than 70% of correct answers even when evaluating the confusing version of the code snippets. It is also important to note that, independently of the *experience group*, developers tend to provide more than 84% of correct answers while predicting the outputs for the clean versions of the code. We replicate the *Pearson's Chi-squared test* for each experience group and find that the statistical difference is less significant for those developers with more than ten years of experience. Differently from *participant experience*, the factor *participant education* does not significantly change the correctness of the answers.

B. Time Analysis

Exploratory Data Analysis: Table VII shows the average time necessary for the participants to give a correct answer about the expected outcomes of a code snippet, considering both confusing and clean versions. We do not consider cases where the participants made mistakes. Seven atom candidates required more time from participants to predict a correct response. For these seven atom candidates, developers take at least 5.90% less time on average to find a correct answer when considering the clean version of a code snippet. In the extreme case (atom candidate Comma Operator), the participants take 76.23% less time on average to find the correct answer for the clean version of the code snippets. Not all atom candidates, though, require less time for the participants to predict the answer. In fact, for the Arithmetic as Logic and the Pre-Increment atoms, the time to give a correct answer increased 29.06% and 38.19% for the clean versions—we did not confirm these atom candidates as atoms of confusion in the previous section.

TABLE VII: Time in seconds to submit a correct answer

Atom	Confusing	Clean	Δ(%)
Comma Operator	87.67	20.84	-76.23
Automatic Semicolon Insertion	46.08	22.04	-52.17
Post-Increment	28.70	25.67	-10.56
Omitted Curly Braces	48.85	30.00	-38.58
Assignment as Value	52.47	48.95	-6.71
Implicit Predicate	36.24	24.01	-33.75
Logic as Control Flow	108.94	51.07	-53.12
Conditional Operator	41.80	39.34	-5.90
Pre-Increment	30.71	42.45	38.19
Arithmetic as Logic	28.82	37.20	29.06

Statistical analysis: We use the non-parametric *Mann-Whitney U test* to investigate the null hypothesis that developers spend the same amount of time to correctly predict the outcome of clean and confusing versions of a code snippet. The results of Table VII suggest that we should refute the null hypotheses for five atom candidates (Comma Operator, Implicit Predicate, Logic as Control Flow, Pre-Increment, and Arithmetic as Logic), after applying the Benjamini-Hochberg correction with a false discovery rate of 5%. For the first three, the analysis suggests that the participants need more time to predict the outcome of the code snippet in the confusing code version. For the atom candidates Arithmetic as Logic and Pre-Increment, the participants take less time to predict the outcome of the code snippets in the confusing version. We also computed the effect size using Cliff's Delta statistic. In the table, negative effect sizes suggest that it took less time to correctly predict the output of clean versions of the snippets. Considering the thresholds established by Romano et al. [23] ($0.147 < |d| < 0.33$ small, $|d| < 0.474$ medium, otherwise large), we found a large effect for the Comma Operator atom candidate; a medium effect size for Logic as Control Flow, and small effect sizes for Automatic Semicolon Insertion, Implicit Predicate, Arithmetic as Logic, Post-, and Pre-Increment.

VI. Results of the Interview Study

In this section we present the results of the interviews with the practitioners. We contrast the findings presented in Section III-D with the opinion of software developers about their preferences regarding the confusing or clean versions of the code snippets we used in our experiment. For the

TABLE VIII: Hypotheses Testing (misunderstanding and time). Asterisks (*) indicate a statistically significant difference.

Atom	Chi-square test	Odds Ratio (misunderstanding)	CI	Mann-Whitey U test (workload)	Cliff's Delta
Comma Operator	1.1727e-10*	19.02	(6.60, 68.22)	5.8249e-08*	-0.53
Automatic Semicolon Insertion	5.8352e-11*	39.33	(9.21, 356.62)	0.09802	-0.16
Post-Increment	0.0015279*	4.83	(1.73, 15.74)	0.12006	0.15
Omitted Curly Braces	0.050962	2.35	(1.00, 5.77)	0.92529	-0.01
Assignment as Value	0.0034775*	8.39	(1.81, 79.12)	0.82681	0.02
Implicit Predicate	0.01123*	6.95	(1.46, 66.56)	0.0029039*	-0.29
Logic as Control Flow	0.292	1.54	(0.73, 3.28)	7.5862e-05*	-0.39
Conditional Operator	0.15896	2.73	(0.74, 12.57)	0.24073	-0.12
Pre-Increment	0.70147	1.25	(0.55, 2.85)	0.0062981*	0.27
Arithmetic as Logic	1	0.84	(0.22, 3.12)	0.01723*	0.23

interview study, we only considered atoms that were analyzed in both the repeated measures (Section IV) and the latin square (Section V) studies.

A total of 15 practitioners took part of the interviews. We collected information regarding programming experience, familiarity with JavaScript, and their opinion about the 10 atom candidates we explored in the studies of the previous sections. We first presented them pairs of clean and confusing code snippets and then asked them to discuss their preferences towards one version. We did not indicate in any way whether any version was assumed to be confusing or not.

TABLE IX: Summary of participants' preferences for code snippets *with* and *without* atom candidates. The participants were only presented the code snippets, without any indication about whether one was confusing or not. Participants were also allowed to choose Neutral when they thought both sides were equally readable.

	PREFERENCE (%)		
Atom	Confusing	Clean	Neutral
Comma Operator	0	100	0
Automatic Semicolon Insertion	0	80	0
Post-Increment	20	73.33	6.67
Omitted Curly Braces	0	100	0
Assignment as Value	20	60	20
Implicit Predicate	20	73.33	6.67
Logic as Control Flow	20	60	20
Conditional Operator	60	26.67	13.3
Pre-Increment	40	46.67	13.33
Arithmetic as Logic	0	93.33	6.67
OVERALL	18	71.33	8.64

The Participants' perceptions of the atom candidates: Supporting the results of the studies of Sections IV and V, Table IX shows that for eight out of the 10 scenarios surveyed, the majority of the respondents prefer the version of the code without the atom candidate. In no case the *neutral* ratio was higher than the option for the clean version. Only for the Conditional Operator atom candidate the participants prefer the confusing version, instead of the clean one. Figure 4 shows the code snippets for the corresponding confusing and clean versions. For this atom candidate, some participants who preferred the left-hand side version (confusing) still believed that the right-hand side version (clean) was more readable. The following quotes were extracted from the transcripts with two interviewees:

> *"I prefer to write [code using the left-hand side version], but I think [the right-hand side version] is easier to read, especially for newer programmers".*

> *"When I am programming, I write code with the conditional operator, [...], but, to be honest, I still think that [the code using the right-hand side version] is easier to understand".*

The Pre-Increment atom candidate also caused such divide in one of the interviewees, who regarded the clean version as simpler to understand, but would still opt to write code with the atom. In contrast, one of the participants found the version with the atom candidate more elegant, but recognized it was less readable, and was willing to sacrifice elegance for readability. In the repeated measures study, we found a significant difference in favor of the clean version, with a large effect size in its favor. In the latin square study, no difference could be observed in terms of correctness. However, there was a significant difference in terms of the time required predict the output correctly in favor of the obfuscated version. This highlights how factors such as the experience of the participants and the selection of code snippets can have considerable impact on the results of studies about code readability [24].

(castor note) Should the last sentence above be in the discussion?

When analyzing the Logic as Control Flow atom candidate, one of the interviewees gave an example of personal experience that might motivate one to avoid writing code using this construct:

> *"This one is interesting, because I have written code that looks like the left-hand side [confusing version], and my colleagues complained that it was difficult to understand. Nowadays I prefer to write code using the version on the right-hand side [clean version]".*

None of the two studies found a significant difference in correctness between clean and obfuscated versions of snippets

```
let config = {size: 3, isActive: false};
const_config = config.isActive === true
             ? config
             : {size: 10};
console.log(_config.size);
```

Listing 2: *Left-hand side* (using the *Conditional Operator* atom).

```
let config = {size: 3, isActive: false}
let _config;
if(config.isActive === true) {
  _config = config;
}
else{
   _config = {size: 10};
}
console.log(_config.size);
```

Listing 3: *Right-hand side* (without the atom).

Fig. 4: Example of a pair of code snippets used in the interview pertaining to the *Conditional Operator* atom candidate

related to this atom candidate. However, in the latin square study the participants analyzing clean versions of the snippets required significantly less time.

For two of the atom candidates, the interviewees were unanimous in their preference for the clean version: Comma Operator and Omitted Curly Braces. Not coincidentally, in both studies there were significant differences in correctness in favor of the clean versions for both atoms. Regarding the first one, we could often notice during the interviews that the *left-hand side* (with the atom of confusion) caused significant confusion among the participants. One remark about the *Comma Operator* atom of confusion is listed below:

> *"The code in [the left-hand side version] is unlikely to be understood unless the programmer knows C or C++".*

As for the atom candidate Omitted Curly Braces, one of the interviewees mentioned that, although they understand why one would opt not to use braces for simple if-then-else statements, they still advised against it, on grounds that:

> *"[I prefer the right-hand side version of the code . . .] If I want to see well-written, easily understandable code, then I also have to do my job. Therefore I believe that, since I do not know who is on the other end maintaining this code, and it could be any person with any level of expertise, then I try to write readable, easy-to-understand code".*

Confusion in JavaScript Code: The final remarks drawn from the interviews are related to potentially confusing constructs suggested by the participants and their perspectives on JavaScript as a language. From the latter, we can also uncover some other forms in which the language itself might contribute to writing confusing code.

One of the participants mentioned that the use of JavaScript's prototype-based inheritance can make it difficult to understand code, particularly when involving deep prototype chains. When asked about particular JavaScript constructs or patterns that can make code difficult to understand, three participants cited the callback pattern—potentially leading to several levels of nested function calls—as extremely difficult to assimilate. One of the respondents stated:

```
let inc = (x) => x + 1;
```

Listing 4: Example of arrow function.

```
let inc = function(x) {
  return x + 1
}
```

Listing 5: Alternative version without arrow function.

> *"Nested callbacks are very confusing. Even writing them can be confusing, let alone understanding them."*

Two other developers implicitly touched upon the callback pattern, mentioning that it can be difficult to understand *asynchronous* programming in JavaScript. We also found other idioms that are JavaScript atom candidates, including *property access* via array subscription (`V1['P1']` instead of `V1.P1`) and *arrow functions* (see listings 4 and 5). The former was investigated in the repeated measures study but we did not find a significant difference between clean and obfuscated versions. As future work, we aim at investigating whether or not these atom candidates are more likely to introduce confusion in JavaScript code.

VII. Results of the MSR Effort

In this section we present the results of our mining software repository effort, whose goal is to reveal how prevalent atom candidates are in open source JavaScript projects. We mined 72 open source JavaScript repositories to understand how often atoms of confusion arise in real software. Similarly to previous studies [5], [7], which investigate the prevalence of atoms of confusion in open source C and C++ projects, we found that atoms of confusion frequently arise in JavaScript open source systems. For instance, the six most frequently found atoms occur in at least 80% of the projects. Considering the extremes, atom candidates Conditional Operator and Implicit Predicate were found in all repositories we mined, while Comma Operator occurred in only 15% of them (as seen in Table X).

Considering all JavaScript projects in our dataset, we found a total of 364 873 atom candidates, though four atoms are responsible for 92.97% of this total: Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces. The first two, based on the results of Section III-D, can be

TABLE X: Summary of atom candidate occurrences in our dataset.

Atom	Projects	Occurr./KLOC
Implicit Predicate	100%	19.89
Conditional Operator	100%	10.16
Omitted Curly Braces	91.67%	6.61
Post Increment	90.28%	5.62
Pre Increment	83.33%	1.04
Assignment as Value	81.94%	1.02
Logic as Control Flow	56.94%	0.95
Automatic Semicolon Insertion	33.33%	0.13
Arithmetic as Logic	23.61%	0.02
Comma Operator	15.28%	0.03

considered atoms of confusion. The remaining atom candidates comprise 25 621 occurrences combined, 7.03% of the total number of occurrences. The Comma Operator and Arithmetic as Logic atoms are the ones that arise less frequently (0.001% in total with 232 and 165 occurrences, respectively).

We calculated the frequency of each atom per KLOC. The four atoms mentioned in the previous paragraph, Implicit Predicate, Post Increment, Conditional Operator, and Omitted Curly Braces, occur frequently in the analyzed projects. All of them have more than 5 occurrences per KLOC on average. In particular, two atom candidates occur very frequently, Implicit Predicate (19.89 occurrences per KLOC) and Conditional Operator (10.16 occurrences per KLOC). On the other hand, Arithmetic as Logic and Comma Operator occur less than once for every 50 KLOC.

The results of our survey and interviews suggest that the Conditional Operator does not contribute significantly as a source of misunderstanding (increasing the number of wrong answers in 9% of the cases, according to our survey). Nonetheless, the Post-Increment Expression and Automatic Semicolon Insertion atoms are listed in the top four sources of misunderstanding (Table VI), and also frequently appear in open source systems. As such, refactoring existing systems to avoid Post-Increment and Automatic Semicolon Insertion might improve the readabiliy of large amounts of code in JavaScript projects.

VIII. Discussion

Comparing the results of the latin square study to previous research, three atoms of confusion that Gopstein et al. [3] confirmed for C and C++ do not lead to a statistically significant impact on correctness in our study: Logic as Control Flow, Conditional Operator, and Pre-Increment. Our results support the Gopstein et al. findings for the remaining atoms. The work of Langhout and Aniche [6] investigates six atom candidates for Java that we also explore here for JavaScript. Among these atom candidates, we achieve similar conclusions to their study for four atom candidates: Post Increment, Conditional Operator, Arithmetic as Logic, and Omitted Curly Braces—both studies confirmed only the former as an atom of confusion. Our findings diverged from the work of Langhout and Aniche for two atoms: Pre Increment and Logic as Control Flow.

(castor note) Which atoms? The weight of experience. Statistical significance vs. practical relevance. Most of them, with the exception of Comma Operator and Object Destructuring, have also been observed for Java programs [6].
In addition, many of the coding idioms that have been previously identified as atoms of confusion in previous work [3], [6], such as Logic as Control Flow
Differences among the two studies, for example, post increment was not significant in the first one but it was significant in the second.

Our work has several implications. First, it adds external validity to the work of Gopstein et al. [3], which investigates the impact of atom candidates on understanding C,C++ code. That is, similarly to their work, the atom candidates Comma Operator, Post/Pre Increment, Omitted Curly Braces, Assignment as Value, Implicit Predicate, and Logic as Control Flow seem to make JavaScript code harder to understand. For five of them, the difference is statistically significant, with a large effect size for four atoms. Our results also refute the hypothesis that Arithmetic as Logic is an atom of confusion (i.e., a source of misunderstanding). In comparison to the original work of Gopstein et al. [3], our study led to some differences in the effect size of the atom candidates. Altogether, we answer our first research question.

> **Answer to RQ1:** The first study (survey) gives evidence that some of the atom candidates in C and C++ programs that also exist in JavaScript correspond to a source of misunderstanding in JavaScript code.

The results of the interview study complement the understanding of atoms of confusion because the participants make clear the existence of a trade-off between code comprehension and other quality attributes. For instance, most of the participants prefer the version of the code with the Conditional Operator, even though they agree that its use might contribute to the misunderstanding of JavaScript code, particularly when novices are maintaining the codebase. The participants of the interview study also mentioned other possible sources of misunderstanding in JavaScript, including the use of prototype-based inheritance and nested call-backs (as discussed in Section VI). Other JavaScript atom candidates include Object Destructuring, Array Spread, Object Spread, and Type Conversion. In summary, the results of the second study (interviews) allow us to answer the second research question.

> **Answer to RQ2:** The qualitative analysis of the interviews supports the results of the first study, since JavaScript developers most often agree that atoms of confusion compromise source code understanding.

The results of the third study (mining open source JavaScript repositories) provides evidence that, although atoms of confusion compromise program comprehension, they frequently

appear in open source JavaScript projects. In particular, seven, out of 10 atoms considered in our study, appear in more than 50% of the 72 projects we analyzed. Furthermore, at least two of them are used intensively, more than once for every 200 lines of code. In summary, the third study allows us to answer the third research question.

Answer to RQ3: The MSR study reveals that the several atom candidates explored in our research appear frequently in practice, and cleaning up the use of Post-Increment/Decrement and the Automatic Semicolon Insertion might improve the readabiliy of JavaScript code substantially.

IX. Threats to Validity

Conclusion validity. In the context of our survey study, we apply different non-parametric statistical tests appropriate for the cases where data was categorical (correctness, Chi-square test of independence) and continuous (time, Mann-Whitney U test). Furthermore, besides reporting p-values, we also report effect size measures appropriate for each scenario (odds-ratio and Cliff's delta) and apply a p-value correction technique to avoid the multiple testing problem. Finally, it could be argued that the size of the samples is insufficient to make conclusions for some of the atoms, a common problem in Empirical Software Engineering. We estimate the sample size for each one of the atom candidates, considering that each one had a different number of samples (Table VI). To that end, we use the ϕ measure of effect size for each atom (based on the Chi-square statistic), the standard α coefficient of 0.05, and set the expected statistical power to 0.8, as usually employed in the literature [20]. We find out that the sample size is sufficient for all but one of the atoms where we had a statistically significant result for correctness, Implicit Predicate.

Construct validity. We use correctness as a proxy for program comprehension and predicted outcomes of small programs as a measure of correctness. As discussed elsewhere [25], this approach is a test of the developers' ability to trace programs. Although this is a common approach in studies about atoms of confusion [3], [6], [11], other measures of correctness could have been employed, potentially yielding different results. As a complement to correctness in the survey, we have measured the time it took for the participants to correctly predict the outcome of the code snippets (either with or without atom candidates). Furthermore, we have asked the interviewees about their preferences when comparing confusing and clean versions of the programs.

Internal validity. Since the survey was conducted online with unknown participants, we have no way of confirming their levels of education and experience. We mitigate this threat by, in the data analysis, explicitly accounting for programmer experience and using an experimental design that allows us to isolate the impact of the treatment from factors such as experience and formal education level. Also, we did not have a way to prevent respondents from cheating, such as running the code on an interpreter, or consulting other people. We partially mitigate this threat by presenting images with source code, instead of text. This creates an obstacle for participants to run the code while taking the survey.

External validity. Our results suggest that the selected idioms and code constructs may lead to confusion for small code snippets, but it is not clear if that result extrapolates to other scenarios. Even though it is likely that in larger code bases the confusion induced by these constructs may be even greater, the existence of additional context may mitigate this effect. Another possible threat to the generalizability of our results, in particular for the survey and interviews, lies in the fact that the analyzed atoms may rarely occur in real JavaScript code bases. To mitigate that threat, we have analyzed 72 popular open source JavaScript repositories and found out that most of them are common, occurring at least once per 1,000 lines of code. Only Arithmetic as Logic and Comma Operator occur less frequently than once per 10,000 lines of code.

X. Conclusions

This paper reports the results of a mixed-method research effort that allowed us to better understand the impact of atoms of confusion in JavaScript code. First, we conducted a survey with 140 JavaScript developers, asking them to predict the output of some code snippets (with and without atoms). We provide evidence that five atom candidates significantly impact program comprehension activities and can be considered atoms of confusion. Our efforts have two implications for practice. First, we present evidence that not all atoms of confusion validated to statically typed languages (such as C, C++, or Java) led to a statistically significant impact on program understanding for JavaScript (a dynamically typed language). Besides that, our findings might be used to alert developers to avoid writing JavaScript code with certain atoms of confusion (e.g., Comma Operator, Automatic Semicolon Insertion, Post Increment/Decrement, Assignment as Value, and Implicit Predicate). Finally, our results might help tool developers to create program transformation tools for removing atoms that frequently appear in software. Our research also pointed out additional JavaScript constructs and idioms that might introduce misunderstanding. These include nested callbacks, *property access*, and *arrow functions*. As future work, we intend to reproduce our survey to validate whether or not these additional *atom candidates* truly affect the understanding of JavaScript code.

References

[1] F. Hermans, *The Programmer's Brain: What every programmer needs to know about cognition*. Manning, 2021.

[2] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms: what really affects code complexity?" in *Proceedings of the 25th International Conference on Program Comprehension, ICPC*. Buenos Aires, Argentina: IEEE Computer Society, May 2017, pp. 66–76.

[3] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K. Yeh, and J. Cappos, "Understanding misunderstandings in source code," in *ESEC/SIGSOFT FSE*. https://doi.org/10.1145/3106237.3106264: ACM, 2017, pp. 129–139.

[4] F. Castor, "Identifying confusing code in swift programs," *In Proceedings of the VI CBSoft Workshop on Visualization, Evolution, and Maintenance*, vol. 1, no. 6, pp. 1–8, 2018.

[5] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi, "An investigation of misunderstanding code patterns in C open-source software projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1693–1726, 2019.

[6] C. Langhout and M. Aniche, "Atoms of confusion in java," in *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '21, New York, NY, USA, 2021, to appear.

[7] D. Gopstein, H. H. Zhou, P. G. Frankl, and J. Cappos, "Prevalence of confusing code in software projects: atoms of confusion in the wild," in *MSR*. https://doi.org/10.1145/3196398.3196432: ACM, 2018, pp. 281–291.

[8] E. George, W. G. Hunter, and J. S. Hunter, *Statistics for experimenters: Design, innovation, and discovery*. Wiley, 2005.

[9] S. R. Tilley, D. B. Smith, and S. Paul, "Towards a framework for program understanding," in *WPC*. DOI: 10.1109/WPC.1996.501117: IEEE Computer Society, 1996, p. 19.

[10] A. B. O'Hare and E. W. Troan, "Re-analyzer: From source code to structured analysis," *IBM Systems Journal*, vol. 33, no. 1, pp. 110–130, 1994.

[11] B. de Oliveira, M. Ribeiro, J. A. S. da Costa, R. Gheyi, G. Amaral, R. de Mello, A. Oliveira, A. Garcia, R. Bonifácio, and B. Fonseca, "Atoms of confusion: The eyes do not lie," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, ser. SBES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 243–252. [Online]. Available: https://doi.org/10.1145/3422392.3422437

[12] D. Gopstein, A.-L. Fayard, S. Apel, and J. Cappos, "Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion," ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 605–616. [Online]. Available: https://doi.org/10.1145/3368089.3409714

[13] H. J. Keselman, J. Algina, and R. K. Kowalchuk, "The analysis of repeated measures designs: A review," *British Journal of Mathematical and Statistical Psychology*, vol. 54, no. 1, pp. 1–20, 2001.

[14] J. H. Neely, *Semantic priming effects in visual word recognition: A selective review of current findings and theories*. Lawrence Erlbaum Associates, Inc., 1991, p. 264–336.

[15] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: A benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 90–101.

[16] E. D. Canedo, R. Bonifácio, M. V. Okimoto, A. Serebrenik, G. Pinto, and E. Monteiro, "Work practices and perceptions from women core developers in OSS communities," in *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*. ACM, 2020, pp. 26:1–26:11. [Online]. Available: https://doi.org/10.1145/3382494.3410682

[17] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble, ".ql: Object-oriented queries made easy," in *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235. Springer, 2007, pp. 78–133. [Online]. Available: https://doi.org/10.1007/978-3-540-88643-3_3

[18] O. Rodriguez-Prieto and F. Ortin, "An efficient and scalable platform for java source code analysis using overlaid graph representations (support material website)," 2020.

[19] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, pp. 153—-157, 1947.

[20] P. D. Ellis, *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, January 2010.

[21] S. S. Mangiafico, *Summary and Analysis of Extension Program Evaluation in R*. New Brunswick: Rutgers Cooperative Extension, 2016, available at rcompanion.org/handbook/.

[22] H. Chen, P. Cohen, and S. Chen, "How big is a big odds ratio? interpreting the magnitudes of odds ratios in epidemiological studies," *Communications in Statistics - Simulation and Computation*, vol. 39, no. 4, pp. 860–864, 2010.

[23] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *Proceedings of the Annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–3.

[24] D. G. Feitelson, "Considerations and pitfalls in controlled experiments on code comprehension," in *29th IEEE/ACM International Conference on Program Comprehension*. Madrid, Spain: IEEE, May 2021, pp. 106–117.

[25] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. Adelaide, Australia: IEEE, 2020, pp. 348–359.