Noname manuscript No.
(will be inserted by the editor)

Investigating Misunderstanding Code Patterns in C Open-Source Software Projects

**Flávio Medeiros · Gabriel Lima ·
Guilherme Amaral · Sven Apel ·
Christian Kästner · Márcio Ribeiro ·
Rohit Gheyi**

Received: date / Accepted: date

Abstract Maintenance consumes 40% to 80% of software development costs. So, it is essential to write source code that is easy to understand to reduce the costs with maintenance. Improving code understanding is important because developers often mistake the meaning of code, and misjudge the program behavior, which can lead to errors. There are patterns in source code, such as *operator precedence*, and *comma operator*, that have been shown to influence code understanding negatively. Despite initial results, these patterns have not been evaluated in a real-world setting, though. Thus, it is not clear whether developers agree that the patterns studied by researchers can cause substantial misunderstandings in real-world practice. To better understand the relevance of misunderstanding patterns, we applied a mixed research method approach, by performing repository mining and a survey with developers, to evaluate misunderstanding patterns in 50 C open-source projects, including *Apache*, *OpenSSL*, and *Python*. Overall, we found more than 109K occurrences of the 12 patterns in practice. Our study shows that according to developers only some patterns considered previously by researchers may cause misunderstandings. Our results complement previous studies by taking the perception of developers into account.

Keywords Misunderstanding Patterns · Repository Mining · Survey

Flávio Medeiros and Gabriel Lima
Federal Institute of Alagoas (IFAL), Maceió, Alagoas, Brazil

Guilherme Amaral and Márcio Ribeiro
Federal University of Alagoas (UFAL), Maceió, Alagoas, Brazil

Christian Kästner
Carnegie Mellon University (CMU), Pittsburgh, Pennsylvania, USA

Sven Apel
Universität Passau, Passau, Germany

Rohit Gheyi
Federal University of Campina Grande (UFCG), Paraíba, Brazil

1 Introduction

Software maintenance is the modification of a software product after delivery to add new functionalities, correct faults, improve design or performance, or adapt programs to different hardware (ISO/IEC/IEEE, 2006). Maintenance consumes 40% to 80% of the software development costs (Glass, 2001). So, writing code that is easy to understand and to change (Buse and Weimer, 2008; Pahal and Chillar, 2017) is essential to reduce costs and time to market (Jha et al, 2016). In open-source projects, in which often many developers contribute to the code base, it is necessary to pay special attention to code understanding and standards to keep the code easier to review, debug, and to find bugs, as described in the guidelines for contributors[1] of the project *Curl*, as well as in a number of research studies (Beller et al, 2014; Rigby et al, 2008; Stamelos et al, 2002).

Even when developers take care of the code base, they often misunderstand the meaning of source code, and misjudge a program's true behavior (Gopstein et al, 2017). This happens even when considering small and isolated patterns in the source code, which can still lead to significant runtime errors. Thus, it is not only important to define a proper high-level architecture of the system (Fowler et al, 1999; Gamma et al, 1995), but also to avoid certain code patterns that influence code understanding negatively. Gopstein et al (2017) discussed a number of bugs related to small and isolated code patterns that caused losses of millions of dollars, such as Apple's goto fail SSL bug (Bland, 2014), the Ariane 5 floating point overflow (Dowson, 1997), and the cascading network failure of AT&T (Burke, 1995).

In the past, researchers have evaluated such misunderstanding patterns in C programs by performing controlled experiments with programmers (Gopstein et al, 2017; Malaquias et al, 2017; Schulze et al, 2013). A *misunderstanding pattern* is a small code excerpt that can influence code understanding negatively. These studies showed that certain code patterns influence code understanding negatively. However, the participants of the experiment of (Gopstein et al, 2017), for instance, were undergraduate students with at least three months of experience, which might not be sufficient to understand complex programming language concepts in detail.

To obtain a better understanding about the relevance of misunderstanding patterns in practice, we applied a mixed research method approach (Creswell and Clark, 2011; Easterbrook et al, 2008), by applying software repository mining, and by conducting a survey with software developers, with the goal of evaluating misunderstanding code patterns in 50 real-world open-source projects, such as *Apache*, *OpenSSL*, and *Python*. We aim at answering the following research questions:

– **RQ1.** What is the frequency of occurrences of misunderstanding patterns in open-source projects?

[1] `https://github.com/curl/curl/blob/master/docs/CODE_STYLE.md`

- **RQ2.** Do developers of open-source projects agree that misunderstanding patterns influence code understanding negatively?
- **RQ3.** What are the guidelines that open-source projects provide to avoid misunderstanding code patterns?
- **RQ4.** Do developers of open-source projects accept pull requests to remove instances of misunderstanding patterns?

Our study considers 12 misunderstanding patterns, which we collected by analyzing the results of previous work and by studying guidelines for contributors of open-source projects. We selected the patterns *dangling else* and *initializations in conditions* from the guidelines of open-source projects, and the other misunderstanding patterns from Gopstein et al (2017). We found occurrences of the majority of patterns studied in this work. In the presence of misunderstanding patterns, developers might misjudge the program behavior, and introduce runtime errors accidentally. Overall, we found more than 109 thousand occurrences of 11 out of 12 patterns considered in our study. However, we found no occurrences of pattern *reversed subscript*, which is criticized in prior work (Gopstein et al, 2017), considering the 50 C projects.

By means of a survey with 97 developers, we learned that most developers agree that the presence of 6 out of 12 misunderstanding patterns can influence code understanding negatively. The reason is that the 6 misunderstanding patterns require developers to know specific programming language concepts in advance to understand how the patterns work. For instance, there is no way to know for sure how a boolean expression is going to be evaluated without knowing exactly the operator precedence rules.

Furthermore, we found that the majority of guidelines for code contributors in open-source projects, address code style, pull request information, and bug report instructions. The guidelines instruct developers to use specific tools or mailing lists for sending pull requests, and bug reports, and they describe very specific code style issues, such as to use spaces instead of tabs for indentation, and to include spaces before and after operators. Only a few projects provide guidelines regarding code understanding, such as *Curl*, *Librdkafka*, *OpenSSL*, and *Reactos*, which guide developers to avoid certain misunderstanding patterns.

To learn about the relevance of misunderstanding patterns, we submitted 35 pull requests to remove instances of misunderstanding patterns that we found in open-source projects. We received feedback for 21 pull requests, and developers accepted 8 pull requests (38%). Despite this being a clear sign that removing misunderstanding pattern is beneficial, we learned that developers do not like to change code that is working to improve only code style issues without fixing bug, or adding new functionalities, which might be one of the reasons for the rather low acceptance rate. A replication package for this study is available at our supplementary *Website*[2] and *Zenodo* (Medeiros et al, 2018b).

The key contributions of this article are:

[2] `http://cpsoftware.com.br/patterns/index.html`

– A better understanding regarding which code patterns may influence code understanding negatively, based on a study that triangulates the results of a survey, repository mining, and pull request submissions, considering real-world settings (Section 4);
– A set of patterns that influence code understanding negatively about which most developers of open-source projects agree (Section 4.3);
– A dataset of 50 C projects, showing that certain misunderstanding patterns occur frequently in practice (Section 4.3).

The remainder of this article is organized as follows. Section 2 discusses a real bug related to the use of a misunderstand pattern. In Section 3, we introduce the misunderstanding patterns studied in this article. Section 4 presents the settings and results of the empirical study, which we performed to better understand misunderstanding code patterns. In Section 5, we list several code guidelines for practitioners deduced from our empirical study. Section 6 presents a literature review of related work, and Section 7 summarises the article.

2 Motivating Example

Developers often make mistakes when trying to understand small, and isolated parts of the source code, which can lead to errors. For instance, *OpenH264*[3] is a project implementing a codec library that supports H.264 encoding and decoding. In Figure 1 (a), we present an excerpt of *OpenH264's* source code. It contains a runtime error arising from the misunderstanding pattern *dangling else*, which we discuss in Section 3. Notice that there is an `else` statement at Line 9. This statement is supposed to belong to the `if` statement that starts at Line 6. However, there is no curly bracket, so the `else` statement actually belongs to the `if` statement that starts at Line 7. The reason is that `else` statements in C belong to the innermost `if` statement when there is no curly bracket. During code reviews, the developers of *OpenH264* fixed the error that we discussed here by adding curly brackets,[4] as we can see in Figure 1 (b).

```
1. while (iAnyMbLeftInPartition > 0) {
2.   int32_t iSliceSize    = 0;
3.   int32_t iPayloadSize = 0;
4.
5.   if (iSliceIdx >= pSliceCtx) {
6.     if (pCtx->iActiveThreads == 1)
7.       if (DynSlice (pCtx))
8.         return MEMALLOCERR;
9.     else if (iSlice >= pSliceCtx) {
10.      return MEMALLOCERR;
11.    }
12.  }
13.}
```
(a)

```
1. while (iAnyMbLeftInPartition > 0) {
2.   int32_t iSliceSize    = 0;
3.   int32_t iPayloadSize = 0;
4.
5.   if (iSliceIdx >= pSliceCtx) {
6.     if (pCtx->iActiveThreads == 1) {
7.       if (DynSlice (pCtx))
8.         return MEMALLOCERR;
9.     } else if (iSlice >= pSliceCtx) {
10.      return MEMALLOCERR;
11.    }
12.  }
13.}
```
(b)

Fig. 1: (a) A code snippet of *OpenH264* with an error arising from the misunderstanding pattern *dangling else*, (b) a solution to fix the error by adding curly brackets.

[3] http://www.openh264.org/
[4] https://rbcommons.com/s/OpenH264/r/465/diff/1#0

Many coding standards and style guides recommend to use curly brackets to avoid the misunderstanding pattern *dangling else*. The *Mozilla* coding style guide,[5] for instance, suggests developers to always include brackets, even in single-line blocks of `if`/`else` statements. Other authors (Cannon et al, 2000; Darnell and Margolis, 1996; Scott, 2000) stated that if you have a nested mixture of `if`/`else` statements, especially with misleading indentation, even expert developers may introduce errors.

Previous studies (Dijkstra, 1968; Elgot, 1976; Gopstein et al, 2017; Marshall and Webber, 2000; Wulf and Shaw, 1973) refer to misunderstanding patterns that may cause runtime errors similar to the one that we discuss here. However, little effort has been put into understanding the relevance of these patterns using real-world settings. Most prior work performs controlled experiments with students (Gopstein et al, 2017; Malaquias et al, 2017; Schulze et al, 2013), with at least three months of experience (Gopstein et al, 2017), which might not be sufficient to understand details regarding programming language concepts. In this article, we study misunderstanding patterns by using a corpus of 50 C open-source projects (see Table 2), on which we performed an empirical study to better understand such patterns, as we discuss in Section 4.

3 Misunderstanding Patterns in Source Code

In this section, we describe the misunderstanding patterns that we considered in our study. We collected these patterns by analyzing the results of different studies (Creswell and Clark, 2011; Easterbrook et al, 2008), including a review of previous work, and by studying the guidelines for contributors provided by 36 open-source projects of our corpus, and the *Mozilla*, *Google*,[6] and *Linux*[7] coding style guides.

We include only patterns that we can detect syntactically without using semantic information. For instance, Gopstein et al (2017) present a pattern related to the use of the same variable for different purposes. To detect it, we need semantic information. So, we have not considered this pattern in our study. Next, we present one example of each pattern included in our study, but our tool is also able to detect variations of the patterns. The complete list of the variations is available at our *Website*.

In Pattern 1, on the left side, we present the misunderstanding pattern *dangling else*. It is not obvious that the `else` statement starting at Line 4 is part of the `if` statement that starts at Line 2, except for the indentation that may change during maintenance tasks. There is an implicit concept in this code snippet, and developers need to know that the `else` clause belongs to closest `if` statement. On the right side, we show one possible way to remove this pattern, in which there are no doubts regarding the respective `if`/`else` statement.

[5] `https://developer.mozilla.org/docs/Mozilla/Developer_guide/Coding_Style`
[6] `https://github.com/google/styleguide`
[7] `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`

Pattern 1 ⟨*Dangling Else*⟩

```
1. if (cond_1)
2.   if (cond_2)
3.     stmt_1
4.   else
5.     stmt_2
```

```
1. if (cond_1) {
2.   if (cond_2) {
3.     stmt_1
4.   } else {
5.     stmt_2
6.   }
7. }
```

We present the misunderstanding pattern *initialization in conditions* in Pattern 2, on the left side. Note that variable `var_1` is initialized and compared to the result of a function in Line 1. Again, developers need to know that an assignment returns the value of the expression on the left, that is, the value of `var_1`, in this case. On the right side, we present a possibility to remove this misunderstanding pattern by separating the initialization from the `if` statement condition.

Pattern 2 ⟨*Initialization in Conditions*⟩

```
1. if ((var_1 = func_1()) == 0) {
2.   stmt_1
3. }
```

```
1. var_1 = func_1();
2. if (var_1 == 0) {
3.   stmt_1
4. }
```

In Pattern 3, we illustrate the pattern *logic as control flow*. As we can see on the left side, the call to `func_1` is performed only if `var_1` is `true` or different from `0`. The reason is that C evaluates the second argument of the logical `&&` operator only when the first one evaluates to `true`. On the right side of Pattern 3, we use an `if` statement to make this situation clearer.

Pattern 3 ⟨*Logic as Control Flow*⟩

```
1. var_1 && func_1();
```

```
1. if (var_1) {
2.   func_1();
3. }
```

We illustrate the pattern *conditional operator* on the left side of Pattern 4. In this pattern, we have an assignment that depends on the value of a variable. We can make this clearer on the right side of Pattern 4 by showing explicitly that we will change the value of `var_1` only if `var_2` is equal to `3`.

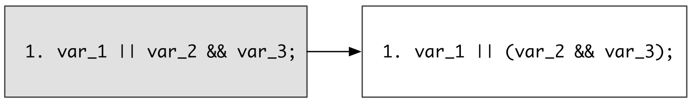
Pattern 4 ⟨*Conditional Operator*⟩

```
1. var_1 = (var_2 == 3) ? 2 : var_1;
```

```
1. if (var_2 == 3) {
2.   var_1 = 2;
3. }
```

In Pattern 5, left side, we show the pattern *operator precedence*. As we can see, it is not clear that the second part of the expression is going to be evaluated first, as operator `&&` has precedence over operator `||`. On the right side, we can make this clearer by adding parentheses.

Pattern 5 ⟨*Operator Precedence*⟩

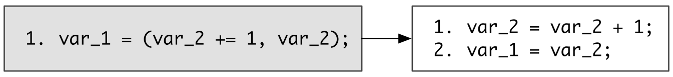


```
var_1 || var_2 && var_3;
```

```
var_1 || (var_2 && var_3);
```

We present the pattern *comma operator* on the left side of Pattern 6. In this pattern, the statements inside the parentheses are executed in sequential order. For instance, as we can see, `var_2` will have its value summed by one, followed by the assignment of `var_2` to `var_1`. On the right side of Pattern 6, we make the order explicit by separating the statements in different lines.

Pattern 6 ⟨*Comma Operator*⟩

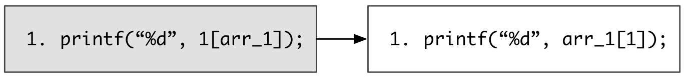


```
var_1 = (var_2 += 1, var_2);
```

```
var_2 = var_2 + 1;
var_1 = var_2;
```

On the left side of Pattern 7, we present the pattern *reversed subscript*. In this pattern, we access an integer array (`arr_1`) and use `printf` to output the value stored at the second position of the array. Notice that this is not the common way of reading values from arrays, but still valid in C. On the right side of Pattern 7, we show the common way of reading array values used by the majority of developers.

Pattern 7 ⟨*Reversed Subscript*⟩

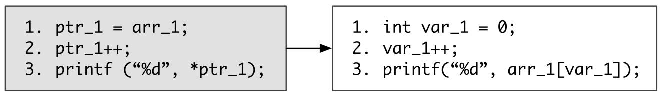


```
printf(“%d”, 1[arr_1]);
```

```
printf(“%d”, arr_1[1]);
```

On the left side of Pattern 8, we present the pattern *pointer arithmetic*. Notice that there is a pointer (`ptr_1`) that receives the address of an integer array (`arr_1`). Instead of using pointers, we can use an integer variable to represent the array index, as we show on the right side of Pattern 8.

Pattern 8 ⟨*Pointer Arithmetic*⟩



```
ptr_1 = arr_1;
ptr_1++;
printf (“%d”, *ptr_1);
```

```
int var_1 = 0;
var_1++;
printf(“%d”, arr_1[var_1]);
```

In Pattern 9, we present the pattern *multiple initializations at the same line*. As we can see on the left side, `var_1` and `var_2` are both initialized in Line 1. On the right side, we separate the initializations in different lines of the source code to make clear that both are initialized. Notice that we may extend this pattern to other types of statements, not only initializations.

Pattern 9 ⟨*Multiple Initializations at the Same Line*⟩

```
1. int var_1 = 10, var_2 = 20;
```

```
1. int var_1 = 10;
2. int var_2 = 20;
```

On the left side of Pattern 10, we present the pattern *assignment as value*. Notice that, in this pattern, we have variable `var_1` receiving the result of an assignment. In this case, we assign `10` to variable `var_2`, which is the value of the left expression. As a consequence, the value `10` is going to be assigned to `var_1` too. On the right side of Pattern 10, we separate the assignments in different lines of the source code.

Pattern 10 ⟨*Assignment as Value*⟩

```
1. var_1 = var_2 = 10;
```

```
1. var_1 = 10;
2. var_2 = 10;
```

In Pattern 11, on the left side, we present the pattern *post-increment*. As we can see, first `var_1` receives the value of `var_2`, and then `var_2` is incremented. On the right side of Pattern 11, we make the sequence of executions explicit. The implicit concept here is that variable `var_2` is incremented, but only after the assignment of its value to `var_1`.

Pattern 11 ⟨*Post-Increment*⟩

```
1. var_1 = var_2++;
```

```
1. var_1 = var_2;
2. var_2 = var_2 + 1;
```

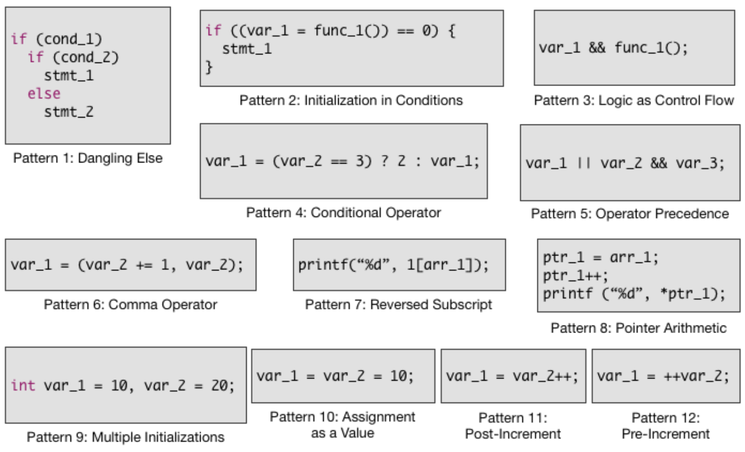
In Pattern 12, on the left side, we illustrate the pattern *pre-increment*. First, `var_2` is incremented, and then `var_1` receives the value of `var_2`. On the right side of Pattern 12, we again make the sequence of executions explicit. The implicit concept here is that variable `var_2` is incremented, but before the assignment of its value to `var_1`.

Pattern 12 ⟨*Pre-Increment*⟩

```
1. var_1 = ++var_2;
```

```
1. var_2 = var_2 + 1;
2. var_1 = var_2;
```

In Figure 2, we present a summary of all patterns considered in our study. Notice that Figure 2 show only the misunderstanding pattern, not presenting the alternatives to remove each pattern.



```
if (cond_1)
  if (cond_2)
    stmt_1
  else
    stmt_2
```

```
if ((var_1 = func_1()) == 0) {
  stmt_1
}
```

Pattern 2: Initialization in Conditions

```
var_1 && func_1();
```

Pattern 3: Logic as Control Flow

```
var_1 = (var_2 == 3) ? 2 : var_1;
```

Pattern 4: Conditional Operator

```
var_1 || var_2 && var_3;
```

Pattern 5: Operator Precedence

```
var_1 = (var_2 += 1, var_2);
```

Pattern 6: Comma Operator

```
printf("%d", 1[arr_1]);
```

Pattern 7: Reversed Subscript

```
ptr_1 = arr_1;
ptr_1++;
printf ("%d", *ptr_1);
```

Pattern 8: Pointer Arithmetic

```
int var_1 = 10, var_2 = 20;
```

Pattern 9: Multiple Initializations

```
var_1 = var_2 = 10;
```

Pattern 10: Assignment as a Value

```
var_1 = var_2++;
```

Pattern 11: Post-Increment

```
var_1 = ++var_2;
```

Pattern 12: Pre-Increment

Fig. 2: Summary of all patterns considered in our study.

4 Study Settings and Results

In this section, we present the settings of our empirical study, which considers a corpus of 50 C open-source projects. The goal of our empirical study is to analyze C projects with respect to evaluating the relevance of our 12 misunderstanding patterns in practice.

4.1 Research Questions

Specifically, we address the following research questions:

– **RQ1.** What is the frequency of occurrences of misunderstanding patterns in open-source projects?
– **RQ2.** Do developers of open-source projects agree that misunderstanding patterns influence code understanding negatively?
– **RQ3.** What are the guidelines that open-source projects provide to avoid misunderstanding code patterns?
– **RQ4.** Do developers of open-source projects accept pull requests to remove instances of misunderstanding patterns?

4.2 Study Setup

To answer these research questions, we selected a corpus of 50 C open-source projects, including *Apache*, *OpenSSL*, and *Python*. Our corpus contains projects from different domains, such as operating systems, Web servers, text editors, security libraries, and databases. For the selection of subject projects, we used *GHTorrent* (Gousios, 2013) with the goal of identifying active projects by sorting them based on the number of stars and pull requests on *GitHub*. In addition, we considered only projects that use C as the primary language of the project. That is, we considered column "language" in *GHTorrent*. The dataset we used was from March 23, 2018. We list all subject projects in Table 2.

For **RQ1**, we conducted a static analysis to count the number of occurrences of misunderstanding code patterns in the subject projects. We implemented a Java tool that searches for occurrences of misunderstanding patterns based on *SrcML*,[8] a tool that generates XML files from source code. Our tool is available at the supplementary Website with an example showing how our tool operates.

To answer **RQ2**, we conducted a developer survey to learn about the relevance of misunderstanding patterns based on the perception of developers. In Appendix A, we present our survey, which includes 12 questions about the misunderstanding patterns (one per pattern), and three additional questions related to the experience of developers, and free text boxes to get additional misunderstanding pattern candidates from developers. To recruit participants, we collected relevant information about developers by mining the software repositories of the projects in our corpus. To select developers to participate in the survey, we measured code churn metric to select the most active developers from each project. Then, we arbitrarily selected a number of developers from the active ones to send emails asking them to fill our survey. Overall, we sent emails to 701 developers, of which 97 (14%) developers completed the survey.

For **RQ3**, we studied the guidelines for code contributors provided by 36 projects of our corpus to identify guidelines with respect to misunderstanding patterns. We searched for guidelines by performing a manual analysis of the respective project repositories. We considered a project to contain guidelines for contributors when there is at least one file, or a specific section, focusing on providing guidelines information. In most projects, we found these guidelines in the `contributing.md` file at the root of the repository. To some projects, we searched the guidelines in the complete software repository.

To answer **RQ4**, we submitted 35 pull requests, arbitrarily selected from pattern instances that we found, to the respective software projects suggesting developers to remove misunderstanding patterns; we counted the number of patches accepted.

4.3 Results and Discussion

In this section, we present the results regarding our study.

RQ1: What is the frequency of occurrences of misunderstanding patterns in open-source projects?

To answer this question, we measured the following metrics, as presented in Table 1: the percentage of projects with occurrences of misunderstanding

[8] `http://www.srcml.org/`

Table 1: Percentage of projects with occurrences, occurrences per thousand lines of code, and the frequency category.

Pattern	**Projects**	**Occurrence/KLOC**	**Category**
Multiple Initializations	100%	66.67	highly used
Conditional Operator	98%	12.05	commonly used
Initialization in Conditions	11%	6.41	commonly used
Assignment as Value	94%	3.02	commonly used
Pointer Arithmetic	74%	0.87	little used
Post Increment	78%	0.57	little used
Pre Increment	80%	0.48	little used
Operator Precedence	54%	0.27	little used
Dangling Else	40%	0.09	little used
Logic as Control Flow	50%	0.12	little used
Comma Operator	0.02%	0.003	little used
Reversed Subscript	0%	–	not used

patterns; and the number of occurrences of the patterns per thousand lines of code. Then, we classified the patterns into categories based on their frequencies. For this, we used the following rules: (1) not used: no occurrence found; (2) little used: at least 1 occurrence to 1.000 occurrences; (3) commonly used: for patterns with more than 1.000 and less than 10.000 occurrences; and (4) highly used: more than 10.000 occurrences.

The patterns *conditional operator*, *multiple initializations*, and *assignment as value*, are used in the majority of open-source projects. We found the pattern *multiple initializations* in all projects analyzed in our study. In contrast, we did not find occurrences of the pattern *reversed subscript.*

Regarding the number of occurrences per thousand lines of code, the pattern *multiple initializations* is definitely the most occurring in practice; we found more than 66 occurrences of this pattern per thousand lines of code. In a previous study, Gopstein et al (2018) found that the patterns analyzed in their work appeared on average once every 23 lines. In our study, the patterns appeared on average once every 11 lines of code.

We classified the pattern *multiple initializations* as highly used in practice, 3 patterns as commonly used, 7 patterns as little used, and 1 misunderstanding pattern as not used in practice (see Table 1). In Figure 3, we present the numbers of occurrences for the patterns considered in our study. We do not show pattern *reversed subscript*, for which we did not find occurrences. In addition, we omitted pattern *comma operator* (only 4 occurrences), and patterns *conditional operator* and *multiple initializations* (high numbers of occurrences). For instance, we found almost 80K occurrences of pattern *multiple initializations*, and *conditional operator*, of which we found almost 15K occurrences.

To study the current state of existing tools with regards to warn developers about misunderstanding patterns, we checked which patterns the tools detect. Existing tools can detect only two patterns. The *Gcc* compiler, the *Clang*

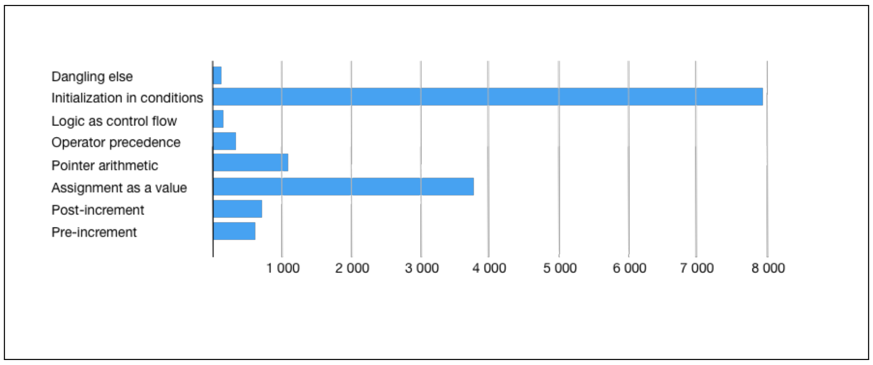


Fig. 3: Frequency of occurrences of the patterns.

static analyzer,[9] and *PVS Studio*[10] are able to warn developers about the use of patterns *dangling else*, and *operator precedence*. All the other patterns are not detected by these tools.

Summary

The majority of patterns analyzed in our study (92%) are used in practice by real developers of open-source C projects. The pattern reversed subscript does not occur in practice, and the patterns dangling else, comma operator, and logic as control flow occur only rarely.

RQ2: Do developers of open-source projects agree that misunderstanding patterns influence code understanding negatively?

As the core of the survey, we asked developers 12 questions (each in *Likert item*) to evaluate how positive or negative the influence of using a particular misunderstanding pattern is. We used the following options as possible answer: *totally positive*, *positive*, *neither positive or negative*, *negative*, or *totally negative*. At the end of the survey, we added a question about years of experience, and two open text boxes for additional comments.

Analyzing the answers of 97 software developers, we found that most developers agree that the use of 6 out of 12 of our patterns (50%) might cause misunderstandings. In Figure 4, we present the results of the survey. The patterns *comma operator* and *reversed subscript* are the ones that most developers agree on causing misunderstandings, more than 90% of developers that filled the survey. This is in line with our results for RQ1, as software developers do not use these patterns in practice.

For the misunderstanding patterns *dangling else*, *initialization in conditions*, *logic as control flow*, and *operator precedence*, our results show that more than 60% of developers agree that they may influence code understanding negatively. Three out of these four patterns are classified as little used in RQ1,

[9] `https://clang-analyzer.llvm.org/`
[10] `https://www.viva64.com/en/pvs-studio/`

Table 2: Overview of the subject projects

Project	**Domain**	**LOC**	**Dev**	**Commits**	**Guides**
Apache	Web Server	201 032	36	30 536	
Cinder	C++ Library	140 731	100	8425	✓
Citus	Database	65 409	28	1661	✓
Cleanflight	Controller Firmware	420 295	262	10 391	✓
Cmake	Build Tool	223 007	570	38 541	✓
Cmus	Music Player	3730	92	2122	✓
Collectd	Statistic Library	94 520	320	9852	✓
Contiki	Operating System	253 667	160	12 325	✓
Ctags	Tags Implementation	76 874	94	5761	
Curl	Command Line Tool	108 802	392	22 794	✓
Dmd	Compiler	77 051	159	18 511	✓
Edk2	Firmware	1 493 867	144	23 168	✓
FFmpeg	Video Tool	868 058	931	89 831	✓
FreeRDP	Remote Desktop	247 437	201	11 599	
Git	Code Mirror	174 715	1163	49 935	
Glfw	Open GL Library	26 154	104	3555	✓
Grpc	RPC Framework	14 266	306	30 434	✓
Hiredis	Database	3902	79	591	
Irssi	Chat Client	53 612	72	5524	✓
Jansson	JSON Tool	6827	54	847	
JohnTheRipper	Password Cracker	231 211	86	14 320	
Krb5	Security Library	262 323	58	19 304	✓
Libpng	Image Library	51 987	23	3199	
Librdkafka	C++ Library	45 127	93	2444	✓
Libssh2	SSH Library	27 377	71	1907	
Libuv	I/O Library	49 634	289	3952	✓
Libwebsockets	Websocket Library	53 412	152	2380	
Lxc	Linux Containers	52 770	279	6251	✓
Mongo	Database	484 903	327	40 354	✓
Mpv	Video Player	115 403	240	46 085	✓
OpenSSL	SSL Library	273 284	313	21 259	✓
Phpredis	Database	12 482	77	1912	
Poco	C++ Library	317 911	171	4724	✓
Premake-core	Premake	148 105	77	3057	
Python	Compiler	288 002	112	561	
Qmk_Firmware	Controller Firmware	148 160	489	7074	✓
Radare2	Reverse Engineering	421 778	444	17 051	✓
Reactos	Operating System	4 042 406	73	70 684	✓
Redis	Database	86 358	243	6528	✓
RetroArch	Libretro API	396 618	236	41 940	✓
Riot-os	Operating System	152 294	171	3150	✓
S2n	Security Library	18 664	61	1888	✓
Silver Searcher	Search Tool	3840	179	1968	✓
Statsite	Administration Tool	15 083	61	705	
Stb	C++ Library	14 948	99	1442	✓
Swift Corelibs	I18n Tool	77 994	220	3252	✓
Syslog-ng	Log Daemon	89 242	72	6866	✓
Systemd	System Manager	297 896	894	31 825	✓
Tvheadend	Streaming Server	126 041	196	9916	✓
Weechat	Chat Client	167 572	84	8427	✓

Table 3: Correlation between perception of misunderstanding and how often the patterns are used in practice.

Pattern	Frequency Category	Negative Perception
Comma Operator	not used	100%
Reversed Subscript	not used	91.75%
Dangling Else	little used	71.73%
Logic as Control Flow	little used	88.66%
Operator Precedence	little used	80.40%
Pointer Arithmetic	little used	39.18%
Post Increment	little used	31.96%
Pre Increment	little used	21.65%
Initialization in Conditions	commonly used	76.29%
Conditional Operator	commonly used	8.25%
Assignment as Value	commonly used	8.25%
Multiple Initializations	highly used	12.37%

the exception is pattern *initialization in conditions*, which is commonly used in practice by developers of open-source projects.

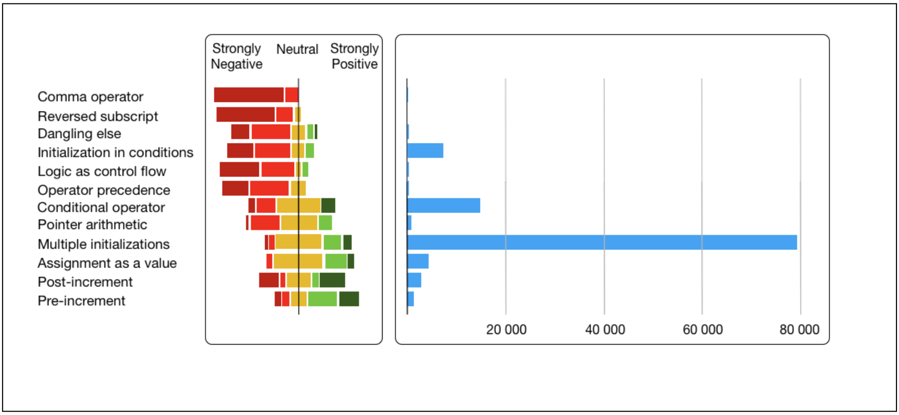


Fig. 4: Results of the survey.

To investigate the relationship between perception of misunderstanding (RQ2), and how often the patterns are used in practice (RQ1), we calculate *Spearman's* correlation coefficient. Overall, we found a strong correlation: $rho = 0.93$. In Table 3, we show the data regarding the category of occurrences for each pattern, and the percentage of developers that agreed that the patterns influence code understanding negatively. As we can see, the majority of patterns that are not used or little used are perceived as negative by developers. However, this does not hold for pattern *initialization in conditions*, which is perceived as negative by 76.29% of developers, but which is commonly used in practice.

According to the majority of developers, the four patterns, *conditional operator*, *pointer arithmetic*, *multiple initializations*, *assignment as a value*, are neither negative or positive. The patterns *post-increment* and *pre-increment*, most developers state that they do not cause misunderstandings.

According to the results of Gopstein et al (2017), the misunderstanding patterns *logic as control flow*, *conditional operator*, *operator precedence*, *comma operator*, *assignment as value*, *post-increment*, *pre-increment*, and *reversed subscript*, influence code understanding negatively. The results of Gopstein et al (2017) differ from our results, as we did not find that the patterns *conditional operator*, *assignment as value*, *pre-increment*, and *post-increment*, influence code understanding negatively based on the answers of developers. Our results are in line with the results of Gopstein et al (2017) with regards to the pattern *pointer arithmetic*, in which both studies could not conclude that this pattern influences code understanding negatively.

Summary

The majority of developers agree that 6 out of 12 patterns (50%) considered in our study may cause misunderstandings in practice. For most patterns that developers do not agree to negatively influence understanding, the majority of developers is neutral.

RQ3: What are the guidelines that open-source projects provide for developers regarding misunderstanding code patterns?

The majority of open-source projects analyzed in our study (72%) provide development guidelines for contributors; see column "Guides" in Table 2. We studied these code guidelines with the goal of identifying rules relevant to misunderstanding patterns. This way, we learned that the majority of the guidelines focus mainly on code style issues, and pull request and bug reporting information.

Still, we found a few guidelines suggesting developers to avoid four patterns considered in our study. The guidelines of *Curl* suggest developers to avoid the pattern *initialization in conditions.* In fact, we found only 9 occurrences of this pattern in *Curl*, while this pattern is very frequent in other projects, such as *OpenSSL*, and *Reactos.* In addition, the *Curl* project also suggests to avoid multiple statements in the same line, as is the case in pattern *multiple initializations.* However, we found 175 occurrences of this pattern in *Curl.* The guidelines of *OpenSSL* guide developers to avoid using nested `if` statements with `else` branches without brackets, that is, the pattern *dangling else.* In *OpenSSL*, we found no occurrences of this pattern, and 40% of the projects that we analyzed contain occurrences of this pattern. In *Librdkafka*, the guidelines mention explicitly to add parentheses to avoid operator precedence problems, supporting the pattern *operator precedence.* However, we found 26 occurrences of the pattern in the *Librdkafka* project, which also appears in 54% of the projects analyzed with similar numbers of occurrences.

Despite the small number of code guidelines targeting misunderstanding patterns, they do not seem to help avoiding the patterns in practice. Among the four patterns mentioned in the guidelines, more than 60% of developers agree that the patterns *dangling else*, *operator precedence*, and *initializations* *in conditions* influence code understanding negatively. In contrast, the pattern *multiple initializations* is not perceived as negative by most developers.

The guidelines of the other subject projects also suggest code style issues, but focus more on fine-grained issues, such as the use of spaces for indentation instead of tabs, and white spaces around operators. The Google, Linux, and Mozilla guides also focus on fine-grained issues, and not on the patterns analyzed in our study, except for pattern *dangling else*, which is mentioned in Mozilla's guide. Table 4 presents some guidelines that we found by analyzing our corpus of open-source projects. In some projects, such as *Dmd* and *Grpc*, the guidelines suggest developers to avoid submitting pull requests that do not fix issues, such as a bug or warning. In other words, these projects guide developers to avoid submitting pull requests to change only the code style (e.g., removing a misunderstanding pattern). It should be done only when fixing other issues in the code base.

In general, there is not a set of specific development guidelines to improve understanding in open-source projects. However, some projects claim to address code understanding by providing guidelines to code style and argue that the use of a common code style makes the code base easier to review, debug, and figure out why things go wrong, such as in *Curl*. In *OpenSSL*, the guidelines say that coding style is all about readability and maintainability, which they check by using available tools, such as the *Clang* analyzer.

Summary

There are only few guidelines for contributors that are specific to misunderstanding patterns. Open-source projects argue to tackle understanding by using a common set of guidelines for code style that do not focus on patterns, but on fine-grained issues, such as the use of spaces for indentation instead of tabs, and white spaces around operators.

RQ4: Do developers of open-source projects accept pull requests to remove misunderstanding code patterns?

We submitted 35 pull requests, arbitrarily selected, to open-source projects, of which we received responses to 26 pull requests. We ignored 5 of our own pull requests because developers mentioned that they were from third-party dependencies or deprecated code. We did not submit pull requests for projects involved in the survey to avoid bias. Notice that it could influence developers to accept or reject pull requests after seen the patterns in the survey. As we discussed in the subject selection (Section 4.2), we selected active projects by sorting C projects based on the number of pull requests with the purpose of receiving fast feedback from developers.

We submitted the 35 pull requests manually. So, this is the first reason for why we were not able to submit hundreds of pull requests. In addition, and more importantly, developers of open-source projects do not seem to like to receive pull requests that change only style without fixing problems. Thus, as

Table 4: The guidelines that open-source projects provide for contributors.

Guideline	**Projects**	#
Use C99 style for comments	*Collectd*, *Krb5*, *Libuv*, *Librdkafka*, *Lxc*, *Mpv*, *OpenSSL*, *RetroArch*, *s2n*, and *Weechat*	10
Use spaces for indentation	*Collectd*, *Curl*, *Edk2*, *Krb5*, *Libuv*, *Mapserver*, *Mongo*, *OpenSSL*, *s2n*, and *Weechat*	9
Use consistent names for variables and functions	*CleanFlight*, *Curl*, *FFmpeg*, *Librdkafka*, *Lxc*, *Libuv*, *Poco*, *OpenSSL*, and *Weechat*	9
Use bracket for every block even with a single statement	*Contiki*, *Cmus*, *Edk2*, *Lxc*, *Micropython*, and *Radare2*	6
Do not write long lines with more than 79 columns	*Curl*, *Cmus*, *Krb5*, *OpenSSL*, and *Weechat*	5
Use a tool to check code style, such as *Clang*	*Curl*, *Mapserver*, and *Mongo*	4
No brackets for blocks with one statement	*Curl*, *Krb5*, *Librdkafka*, and *OpenSSL*	4
Avoid warnings in all major platforms	*Curl*, *OpenSSL*, and *Python*	3
Use space around binary operators	*Curl*, *Mpv*, and *OpenSSL*	3
Define macros, typedefs and enums in uppercase	*Openssl*, *Linux*	2
Keep function return type at a single line alone	*Krb5*, *Mruby*	2
Do not write long functions	*CleanFlight*, and *OpenSSL*	2
Open bracket at next line of function definition	*Curl*, and *Krb5*	2
Use C89 style for comments	*Mruby*, and *Poco*	2
Always use parentheses in evaluations	*Weechat*	1
Avoid global variables	*CleanFlight*	1
Avoid checking platforms and operating systems in `#ifdefs`	*Curl*	1
Do not change auto generated code	*Grpc*	1
Do not check conditions (true, false, or null)	*Curl*	1
Do not include assignments in conditions	*Curl*	1
Do not use space after unary operators	*OpenSSL*	1
Do not use space before parentheses	*Curl*	1
Do not use spaces at the end of lines	*Krb5*	1
Else branch starts at the next line of a closing bracket	*Curl*	1
Nested compound statements must have brackets	*OpenSSL*	1
Never write multiple statements on the same line	*Curl*	1
Opening bracket at the same line of the statement	*Curl*	1
Use column alignment when breaking statements in multiple lines	*Curl*	1
Use one space for preprocessor directives indentation	*Openssl*	1
Use one white space after command keywords	*Krb5*	1
Use brackets to make operator precedence explicit	*Librdkafka*	1
Use tabs for indentation	*Radare2*	1

our pull request study was not bringing benefits to the projects, we decided to avoid submitting more pull requests.

Before submitting pull requests, we studied the guidelines of the project, when available, to avoid introducing code style issues. For the projects that we submitted pull request to, *John the Ripper* and *FreeRDP* do not provide guidelines. According to our experience, developers tend to reject patches because of very small code style problems (Medeiros et al, 2013, 2015b, 2018a). However, developers accepted 8 (38%) out of 21 pull requests considered in our study that we received feedback for. We submitted the 8 pull requests accepted to the following projects: *Curl*, *Grpc*, *John the Ripper*, *Map Ready*, *Mruby*, *Poco*, *RetroArch*, and *Systemd*.

Specifically, the developers of the 8 open-source projects accepted pull requests to remove the following misunderstanding patterns: 2 pull requests to remove *dangling else*, 2 pull requests to remove *operator precedence*, 2 pull requests to remove *conditional operator*, 1 pull request to remove *assignment as value*, and 1 pull request to remove *post-increment*. The patterns *dangling else* and *operator precedence* are perceived as negative by developers according to the results of our survey. However, developers also accepted pull requests for the patterns *conditional operator*, *assignment as value*, and *post-increment*, which are not perceived as negative according to the survey. Notice that we did not submit pull requests to remove the pattern *reversed subscript* because we did not find occurrences of this patterns in practice.

To compare the acceptance rate of our study to the acceptance rate of pull requests submitted by developers, we analyzed the number of pull requests accepted and rejected for the projects that we submitted pull requests. Overall, the acceptance rate of the open-source projects considering all pull requests closed, on average, is 29%, lower than the acceptance rate of our study (38%). The details about this study is available at the supplementary Website.

Developers rejected 13 (62%) out of 21 pull requests that we submitted to open-source projects for which we received feedback. In most cases, developers said that the original version of the code is also readable, that the pattern is commonly used in practice, or that they prefer to handle such issues by following standard guidelines, such as the *Google Code Style* guides. In Table 5, we present the patches rejected and the reasons raised by developers for rejection.

In some projects, such as *Dmd* and *Grpc*, the guidelines suggest developers to avoid submitting pull requests that do not fix issues, such as a bug or warning. In other words, these projects guide developers to avoid submitting pull requests to change only the code style. It should be done only when fixing other issues in the code base. This might be a possible reason for the high percentage of rejected patches. It is important to note that developers rejected patches even to remove patterns that more than 60% of software developers agree on to influence code understanding negatively, such as the pull requests submitted to *Irssi*, *Dmd*, and *Libgit2*.

We also submitted pull requests to projects suggesting developers to change the guidelines. Here, we submitted 5 pull requests and received feedback from

Table 5: Patches rejected by developers.

Project	**Pattern**	**Reason exposed by developers**
Dmd	*Conditional Operator* (P4)	Chaining ternary expressions is a common pattern in some code bases.
FreeRDP	*Init. in Condition* (P2)	It is not really necessary and more or less preference.
Irssi	*Dangling Else* (P1)	All style-related changes handled in a specific pull request.
Libgit2	*Operator Precedence* (P5)	I don't think this is much more readable. I definitely prefer the brevity.
Open TX	*Assign. as Value* (P10)	The original code doesn't seem unreadable.
Ossec Hids	*Cond. Operator* (P4)	Both versions seem easy enough to read.
Machine Kit	*Operator Precedence* (P5)	That is a very commonly used C assignments.
MapReady	*Pre Increment* (P11)	I prefer the ++ style of incrementing myself, it's pretty standard C at this point.
Mpv	*Operator Precedence* (P5)	The short-circuiting behavior is already pretty common knowledge.
Radare	*Logic as Control Flow* (P3)	It is clear and valid for their coding style.

4 pull requests. Many developers agreed with our guidelines by making a number of positive comments, saying that they agree with most guidelines, that the guidelines may improve maintainability, and that the guidelines help to avoid compiler warnings. For instance, a developer from project *Cleanflight* mentioned that "*[he] fully agrees with the first two patterns: dangling else, and logic as control flow*".

We also received negative feedback for some misunderstanding patterns, including the *Cleanflight* project, such as for the pattern *operator precedence*, in which developers mentioned that it is better to use parentheses only when they are required, such as the sentences of a developer saying that "*brackets are there to alter the normal operator precedence, and when seeing a bracket you should be able to assume that the operator precedence has been altered. Extra brackets make the code less readable and less understandable"*. However, we also received opposite opinions at the same pull request, such as a developers from *Cleanflight* mentioning that "*[he] prefers to have parenthesis always, to [him] it makes simpler to read*".

Furthermore, some developers mentioned that the project follows the *Google Code* guides, and that the contributor's guidelines are not the best place to include an exhaustive list of misunderstanding patterns. A developer from the *Libuv* project mentioned that "*[he] dislikes the direction this pull request is suggesting. [He does not] think [that] an exhaustive list of good coding practices is really appropriate for a file like contributing.md"*. Overall, three pull requests were rejected, and one is under review.

Summary

Developers accepted 8 out of 21 pull requests (38%) for which we received feedback. For the others, developers stated mostly that the current version of the code is fine and they resist to change the code, as it is working without errors.

4.4 Threats to Validity

Next, we discuss potential threats to validity of our study, considering the opinion of developers, occurrences of misunderstanding patterns in practice, analysis of contributors guidelines, and pull request submission.

Regarding the occurrences of patterns in practice, we used *SrcML*,[11] which uses heuristics that may fail in source code with undisciplined preprocessor directives (Liebig et al, 2011). To minimize this threat, we did not include projects with high numbers of undisciplined preprocessor directives (Liebig et al, 2011), and the majority of projects included in our study do not use the C preprocessor heavily. To minimize this threat, we performed a small experiment to test our tool. We detected 17 patterns manually, from 6 projects. Initially, the tool was able to detect 13 patterns. In addition, we included 50 patterns detected by the tool published at the study of Gopstein et al (2017), and our tool detected 44 patterns. When performing this experiment, we detected some problems in the implementation and fixed the issues. The current version of our tool is able to detect all 67 patterns considered in the experiment. It is important to notice that the numbers reported in this article have been generated by using the current version of our tool.

We found that the majority of developers (52%) have less than five years of experience with C. A couple of developers sent us additional comments saying that *"experienced programmers should be able to understand all of the given code patterns immediately"* and *"if a programmer has any difficulties to understand even a single of the given patterns, he probably has only a couple of years of experience with C"*. To address this threat, we analyzed the results of the survey by separating the developers into two different groups: (1) less than five years of experience; and (2) developers with more than five years of experience. Still, we found that the results are very similar, minimizing this threat to validity.

In our pull requests, we submitted modifications that do not add new functionalities, nor fix bugs or warnings. As many projects guide developers to avoid submitting pull requests addressing code style issues only, we might have a bias in our study. To understand this threat, we collected information about the reasons that made developers reject our pull requests, by discussing with developers on the pull request via the *GitHub* infrastructure.

[11] `http://www.srcml.org/`

5 Guidance for Practitioners

Our study has several implications for practitioners that use the C language in practice as well as for researchers interested in program comprehension, as we discuss next.

We found that the most developers (at least 60%) do not recommend the use of the following patterns in practice: *dangling else*, *initialization in conditions*, *logic as control flow*, *operator precedence*, *comma operator*, and *reversed subscript*. Thus, it may be advisable to avoid the use of these patterns. Furthermore, it could be useful to include information about them in the code guidelines of open-source projects, as we already found for four patterns: *Curl* suggests developers to avoid *initialization in conditions*, and *multiple initializations*; *OpenSSL*, and *Reactos* guide developers to avoid *dangling else*; and *Librdkafka* mentions explicitly to avoid the pattern *operator precedence*. Our study provides a first step to develop guidelines grounded in research data and taking into account developer preferences and acceptance.

We found many open-source projects guiding developers to avoid pull requests that only make stylistic improvements of source code without fixing bugs or warnings. It would be interesting to include tools to check certain patterns in new pull requests, so that we could avoid new occurrences of these misunderstanding patterns. Furthermore, there might be a need to develop tools, integrated with the current IDE used by developers of C open-source projects, to avoid developers to add new occurrences of misunderstanding patterns as well as to remove existing occurrences. External tools that detect guideline violations automatically (such as misunderstanding patterns) and propose fixes (e.g., refactorings) can likely have a larger impact on practice, and may simplify the work of developers. Refactoring tools may also suggest to remove misunderstanding patterns when developers are fixing bugs and warnings.

Researchers might use the results of our study to make their tools more attractive to developers by taking their perspective and needs into account. As we discussed in previous work (Medeiros et al, 2015a), developers are not aware of research tools. Thus, researchers should not only take the perception of developers into account, but they also need to interact somehow with practice, through pull requests to open-source projects, for example, to make developers aware of their tools. Moreover, our results motivate further research to study automated refactorings to remove misunderstandings.

6 Related Work

There has been research on misunderstanding patterns since at least the late sixties. Dijkstra published a study discussing the problems of using `go to` statements (Dijkstra, 1968). According to his study, `go to` statements should be abolished from all high-level programming languages, as it becomes terribly hard to understand the sequence of execution of programs in the presence of `go to` statements. A follow-up study discusses several programming language taboos, including the `go to` statement, and agreed with the problems of having explicit control transfers using `go to` statements (Marshall and Webber, 2000). In a recent study, Nagappan et al (2015) have studied the use of `go to` statements empirically, and the results show that developers are still using these statements, but developers limit themselves to use `go to` in certain constructs, avoiding unrestricted use as discussed by Dijkstra (1968).

Researchers also discussed the problems of other patterns and language constructs, such as global mutable state. For example, researchers argued to avoid global variables and use local ones to guarantee that variables are starting from known values in every path of execution. Marshall and Webber (2000) mention that the standard advice to notice developers today is that global variables are bad and you should not use it. Wulf and Shaw (1973) claimed that global variables are a major contributing factor in programs which are difficult to understand, and should be abolition from modern programming languages. Another example are magic numbers, which are constants used in the code, such as array sizes, character positions, and so on (Kernighan and Pike, 1999). According to researchers, every magic number should have a name for its own to ease understanding.

A topic frequently related to understanding is the discipline of preprocessor directives. Several studies criticized the use of the C preprocessor regarding its lack of separation of concerns, and code obfuscation, which make maintenance, and program comprehension difficult. Spencer and Collyer (1992) argue that developers normally tend to use the preprocessor to workaround problems instead of dealing with portability in the right way, that is, planning in advance and structuring the code accordingly.

Ernst et al (2002) presented an empirical study on the C preprocessor by analyzing 26 packages comprising 1.4 MLOC. They found that most C preprocessor usage follows simple patterns. The researchers also discussed about the undisciplined use of the C preprocessor and its problems, such as that it makes the program more difficult to understand. Baxter (1992) proposed DMS, a source-code transformation tool for C and C++. In a more recent work (Baxter and Mehlich, 2001), the authors used the DMS tool and emphasized the problem of using unstructured directives. Garrido and Johnson (2003) developed the CRefactory, a refactoring tool for C to remove certain patterns of preprocessor usage to minimize the problems related to code understanding. Liebig et al (2011) analyzed 40 systems and suggested that developers can introduce subtle syntax errors when using undisciplined directives. The authors found that the undisciplined use of the preprocessor corresponds to 15.6% of the total number of directives.

Developers sometimes refer to the excessive use of preprocessor directives as the "`#ifdef` hell" (Lohmann et al, 2006). A specific practice that has been discussed is the undisciplined use of preprocessor directives, that is, conditional compilation directives that do not align with the syntactic code structure. Undisciplined use of preprocessor directives has been related to error proneness, as we discussed in our previous studies (Kästner et al, 2011;

Medeiros et al, 2013, 2016, 2018a), decrease of code understanding and code maintainability (Baxter and Mehlich, 2001; Ernst et al, 2002), and limitations in tool support, as discussed in a previous study (Padioleau, 2009).

In our previous work, we performed some studies by using different methods, such as surveys (Medeiros et al, 2015a, 2018a), interviews (Medeiros et al, 2015a), repository mining (Medeiros et al, 2013, 2016, 2018a), and controlled experiments (Feigenspan et al, 2013), to analyze the problems of undisciplined preprocessor use taking the perception of developers into account. Here, we use a similar approach but considering other aspects of the C language.

Gopstein et al (2017) presented a set of misunderstanding patterns of C programs. They performed two controlled experiments, on a sample composed largely by students, that showed that the set of patterns analyzed increased significantly the misunderstanding rates. They considered patterns such as *pre-increment*, *if ternary initializations*, and *macro operator precedence*. We included 10 misunderstanding patterns from Gopstein et al (2017) in our study with the goal of evaluating them in a real-world setting, involving developers from open-source projects. In a recent study, Gopstein et al (2018) used a corpus of 14 open-source projects to measure the prevalence misunderstanding patterns. Their results showed that the patterns analyzed appear on average once every 23 lines. In addition, their study showed that there is a strong correlation between misunderstanding patterns and commits to fix bugs. In our study, we found an occurrence of misunderstanding pattern every 11 lines of code on average.

Fowler et al (1999) defined a set of bad code smells, which are structures in the code that we can refactor to improve the design of existing systems. The work of Fowler et al. focuses more on high-level design issues, such as architectural code smells, while our work considers more fine-grained patterns. Other researchers used obfuscation techniques to make the source code more difficult to understand with the purpose of protecting intellectual properties by hindering reverse engineering attacks. Herzberg and Pinter (1987) present protocols that enable software protection, without causing substantial overhead in distribution and maintenance, based on DES and RSA. Collberg et al (1997) reviewed various techniques to obfuscate code and proposed a code obfuscation strategy based on obfuscating transformations.

To summarize, there are several studies discussing the problems and the importance of code understanding for software evolution and maintenance. Here, we complement previous work by evaluating a set of misunderstanding patterns by taking the perception of real developers into account.

7 Concluding Remarks

In this article, we discuss a mixed-method study on misunderstanding patterns, including repository mining and analysis, and a survey with developers. We considered 50 C open-source projects and showed that the majority of the misunderstanding patterns, taken from a previous study (Gopstein et al, 2017), are commonly used in practice.

In our survey, we found that developers agreed that the use of 6 out of the 12 patterns influence code understanding negatively. For the other 6 patterns, most developers are indifferent.

By analyzing the guidelines of the 50 subject projects, we found that most guidelines do not address code understanding. They focus rather on information about how to structure pull requests, which tools developers should use, and how to report bugs. Furthermore, the majority of the guidelines regarding code style address fine-grained issues, such as the use of spaces for indentation instead of tabs, use of spaces before and after operators, and not to write long lines with more than 80 columns.

To understand the relevance of misunderstanding patterns, we submitted 35 pull requests to remove misunderstanding patterns in open-source projects, for which we received feedback for 21 pull requests, and developers accepted 8 (38%) pull requests. Despite this low acceptance rate, our results suggest that developers tend not to accept pull requests that do not fix errors and warnings, and developers tend to resist to change source code that is working.

In future work, we are planing to extend our study to other programming languages with the purpose of identifying new misunderstanding patterns that developers should avoid in practice. Furthermore, we intend to define and evaluate refactorings to remove misunderstanding patterns automatically.

Acknowledgement

We would like to thank Dan Gopstein for the useful feedback regarding our study. Apel's work has been supported by the German Research Foundation (AP 206/6). This work was funded by CNPq (308380/2016-9, 477943/2013-6, 460883/2014-3, 465614/2014-0, 306610/2013-2, 307190/2015-3, and also CNPq 409335/2016-9), FAPEAL (PPG 14/2016), and CAPES grants (175956 and 117875).

References

Baxter I, Mehlich M (2001) Preprocessor conditional removal by simple partial evaluation. In: Procedings of the Working Conference on Reverse Engineering, IEEE, WCRE, pp 281–290

Baxter ID (1992) Design maintenance systems. Commun ACM 35(4):73–89

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the Working Conference on Mining Software Repositories, ACM, pp 202–211

Bland M (2014) Finding more than one worm in the apple. Commun ACM 57(7):58–64

Burke D (1995) All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse. California Polytechnic State University

Buse RP, Weimer WR (2008) A metric for software readability. In: Proceedings of the International Symposium on Software Testing and Analysis, ACM, pp 121–130

Cannon LW, Elliott RA, Kirchhoff LW, Miller JH, Milner JM, Mitze RW, Schan EP, Whittington NO, Spencer H, Brader M, Cannon LW, Elliott RA, Kirchhoff LW, Miller JH, Milner JM, Mitze RW, Schan EP, Whittington NO, Spencer H, Brader M (2000) Recommended C style and coding standards

Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland

Creswell JW, Clark VLP (2011) Designing and Conducting Mixed Methods Research. SAGE Publications

Darnell PA, Margolis PE (1996) C: A Software Engineering Approach. Springer

Dijkstra EW (1968) Go to statement considered harmful. Commun ACM 11(3):147–148

Dowson M (1997) The Ariane 5 software failure. SIGSOFT Softw Eng Notes 22(2):84–93

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting Empirical Methods for Software Engineering Research, Springer, pp 285–311

Elgot CC (1976) Structured programming with and without go to statements. IEEE Transactions on Software Engineering SE-2(1):41–54

Ernst M, Badros G, Notkin D (2002) An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering 28(12):1146–1170

Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2013) Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering 18(4):699–745

Fowler M, Beck K, Brant J, Opdyke W, Roberts D, Gamma E (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley

Garrido A, Johnson R (2003) Refactoring C with conditional compilation. In: Proceedings of the IEEE International Conference on Automated Software Engineering, IEEE, pp 323–326

Glass RL (2001) Frequently forgotten fundamental facts about software engineering. IEEE Softw 18(3):112–111

Gopstein D, Iannacone J, Yan Y, DeLong L, Zhuang Y, Yeh MKC, Cappos J (2017) Understanding misunderstandings in source code. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, ESEC/FSE 2017, pp 129–139

Gopstein D, Zhou H, Frankl P, Cappos J (2018) Prevalence of confusing code in software projects: Atoms of confusion in the wild. In: Proceedings of the Working Conference on Mining Software Repositories, ACM

Gousios G (2013) The GHTorent dataset and tool suite. In: Proceedings of the Working Conference on Mining Software Repositories, IEEE Press, pp

233–236

Herzberg A, Pinter SS (1987) Public protection of software. ACM Trans Comput Syst 5(4):371–393

ISO/IEC/IEEE (2006) Iso/iec/ieee international standard for software engineering - software life cycle processes - maintenance. Std 14764-2006 pp 1–58

Jha MM, Vilardell RMF, Narayan J (2016) Scaling agile scrum software development: Providing agility and quality to platform development by reducing time to market. In: 2016 IEEE 11th International Conference on Global Software Engineering (ICGSE), pp 84–88

Kästner C, Giarrusso P, Rendel T, Erdweg S, Ostermann K, Berger T (2011) Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the Object-Oriented Programming Systems Languages and Applications, ACM, pp 805–824

Kernighan BW, Pike R (1999) The Practice of Programming. Addison-Wesley

Liebig J, Kästner C, Apel S (2011) Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In: Proceedings of the International Conference on Aspect-Oriented Software Development, ACM, pp 191–202

Lohmann D, Scheler F, Tartler R, Spinczyk O, Schröder-Preikschat W (2006) A quantitative analysis of aspects in the eCos kernel. In: Proceedings of the European Conference on Computer Systems, ACM, pp 191–204

Malaquias R, Ribeiro M, Bonifácio R, Monteiro E, Medeiros F, Garcia A, Gheyi R (2017) The discipline of preprocessor-based annotations does #ifdef TAG N'T #endif matter. In: Proceedings of the International Conference on Program Comprehension, IEEE Press, pp 297–307

Marshall L, Webber J (2000) Gotos considered harmful and other programmers taboos. In: Proceedings of the Workshop of the Psychology of Programming Interest Group, PPIG, pp 171–180

Medeiros F, Ribeiro M, Gheyi R (2013) Investigating preprocessor-based syntax errors. In: Proceedings of the International Conference on Generative Programming: Concepts & Experiences, ACM, pp 75–84

Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R (2015a) The Love/Hate Relationship with the C Preprocessor: An Interview Study. In: European Conference on Object-Oriented Programming (ECOOP), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Leibniz International Proceedings in Informatics (LIPIcs), vol 37, pp 495–518

Medeiros F, Rodrigues I, Ribeiro M, Teixeira L, Gheyi R (2015b) An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In: Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, ACM, pp 35–44

Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S (2016) A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the International Conference on Software Engineering, ACM, pp 643–654

Medeiros F, Ribeiro M, Gheyi R, Apel S, Kastner C, Ferreira B, Carvalho L, Fonseca B (2018a) Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. IEEE Transactions on Software Engineering 44(5):453–469

Medeiros F, Silva G, Amaral G, Apel S, Kästner C, Ribeiro M, Gheyi R (2018b) Investigating Misunderstanding Code Patterns in C Open-Source Software Projects (Replication Package). DOI 10.5281/zenodo.1461534, URL `https://doi.org/10.5281/zenodo.1461534`

Nagappan M, Robbes R, Kamei Y, Tanter E, McIntosh S, Mockus A, Hassan AE (2015) An empirical study of goto in C code from GitHub repositories. In: Proceedings of the Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, pp 404–414

Padioleau Y (2009) Parsing C/C++ code without pre-processing. In: Proceedings of the International Conference on Compiler Construction, Springer-Verlag, pp 109–125

Pahal A, Chillar RS (2017) Code readability: A review of metrics for software quality. International Journal of Computer Trends and Technology 46(1):1–58

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: A case study of the Apache server. In: Proceedings of the International Conference on Software Engineering, ACM, pp 541–550

Schulze S, Liebig J, Siegmund J, Apel S (2013) Does the discipline of preprocessor annotations matter? a controlled experiment. In: Proceedings of the International Conference on Generative Programming: Concepts and Experiences, ACM, pp 65–74

Scott ML (2000) Programming Language Pragmatics. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Spencer H, Collyer G (1992) #ifdef considered harmful, or portability experience with C News. In: USENIX Summer Technical Conference, pp 185–197

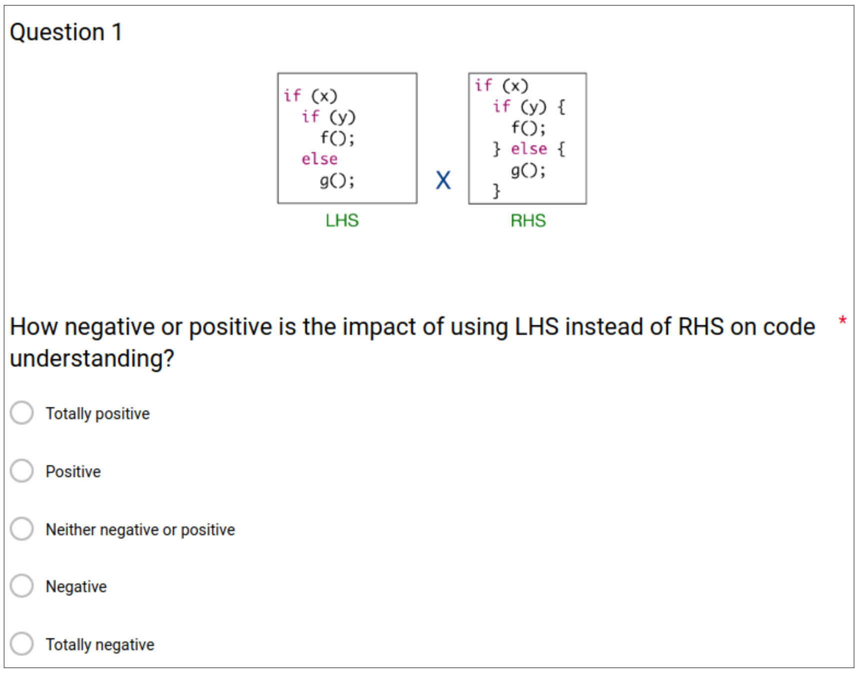
Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. Information Systems Journal 12(1):43–60

Wulf W, Shaw M (1973) Global variable considered harmful. SIGPLAN Not 8(2):28–34

Appendix A Survey with Developers

We are investigating specific C constructions (code patterns) in the source code. This survey presents some code patterns and ask you about their influence in terms of understanding the source code. For each question we will present the code patterns at the Left-Hand Side (LHS) and an alternative on the Right-Hand Side (RHS).

You should be able to answer our survey in around 10-15 minutes. We will use your answers to understand the practical use of code patterns and develop supporting tools. We really appreciate your help. Thanks!



Question 1

```
if (x)
  if (y)
    f();
  else
    g();
```

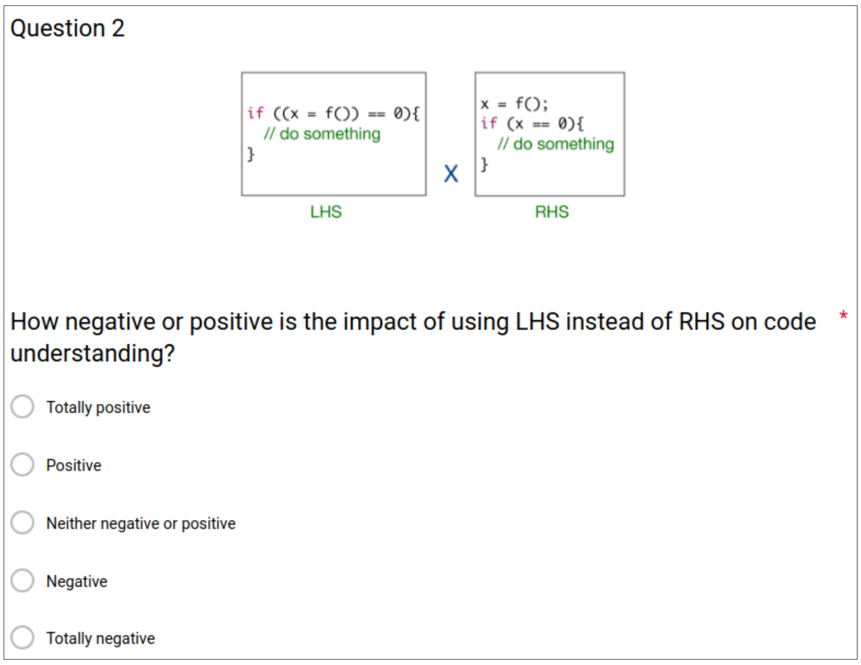
X

```
if (x)
  if (y) {
    f();
  } else {
    g();
  }
```

LHS RHS

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- Totally positive
- Positive
- Neither negative or positive
- Negative
- Totally negative



Question 2

```
if ((x = f()) == 0){
  // do something
}
```

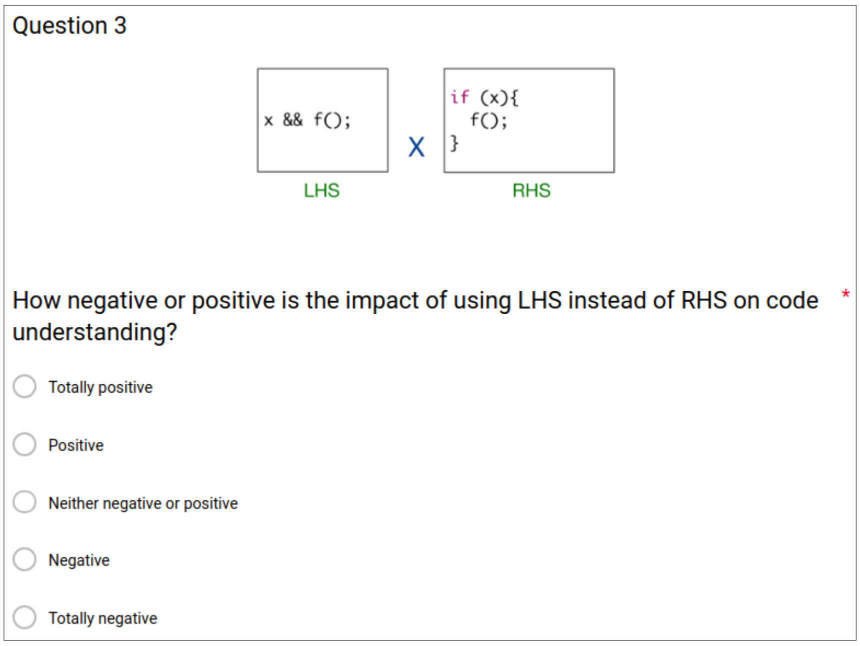
X

```
x = f();
if (x == 0){
  // do something
}
```

LHS RHS

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- Totally positive
- Positive
- Neither negative or positive
- Negative
- Totally negative



Question 3

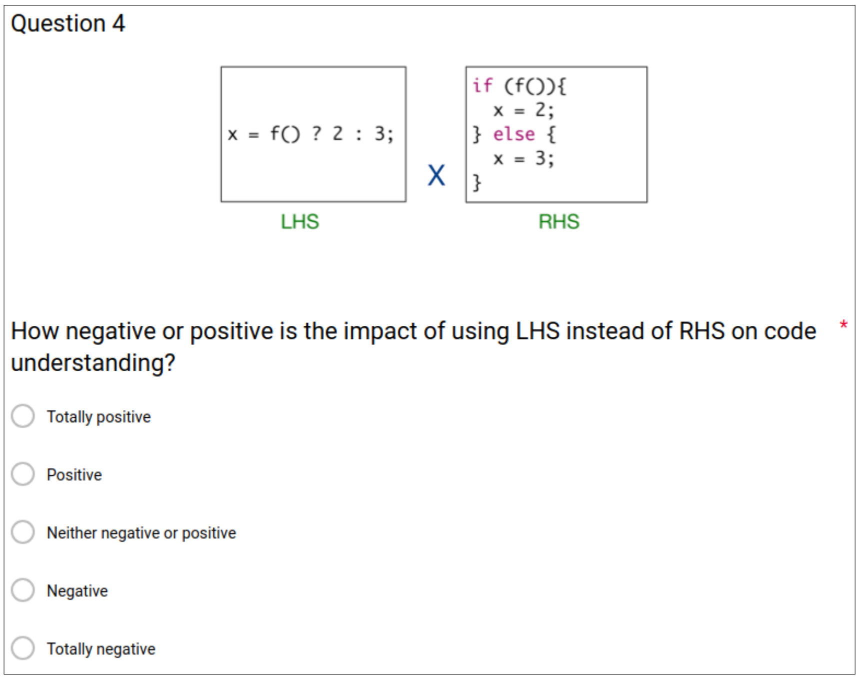
LHS
```c
x && f();
```

X

RHS
```c
if (x){
  f();
}
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- Totally positive
- Positive
- Neither negative or positive
- Negative
- Totally negative



Question 4

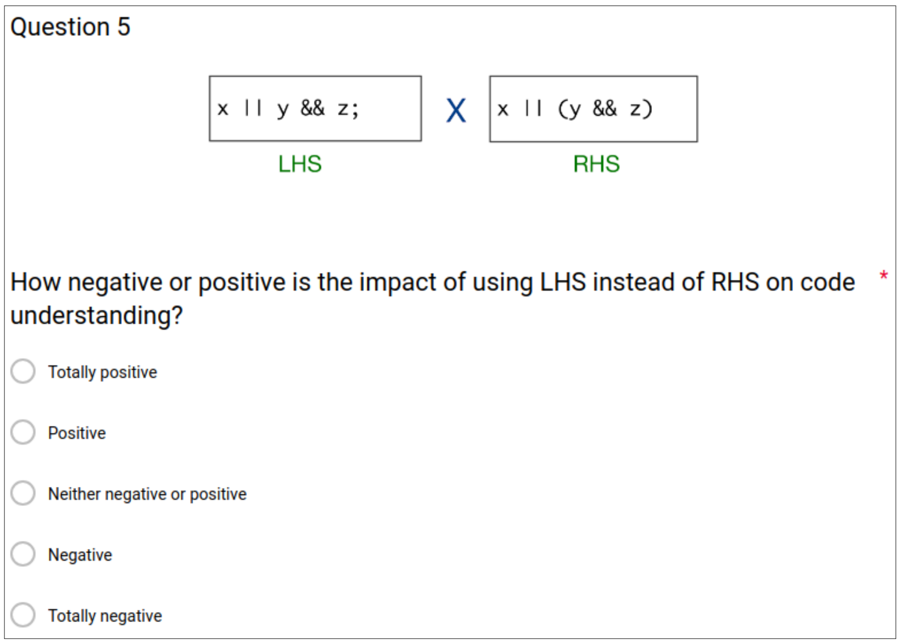
LHS
```c
x = f() ? 2 : 3;
```

X

RHS
```c
if (f()){
  x = 2;
} else {
  x = 3;
}
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- Totally positive
- Positive
- Neither negative or positive
- Negative
- Totally negative



Question 5

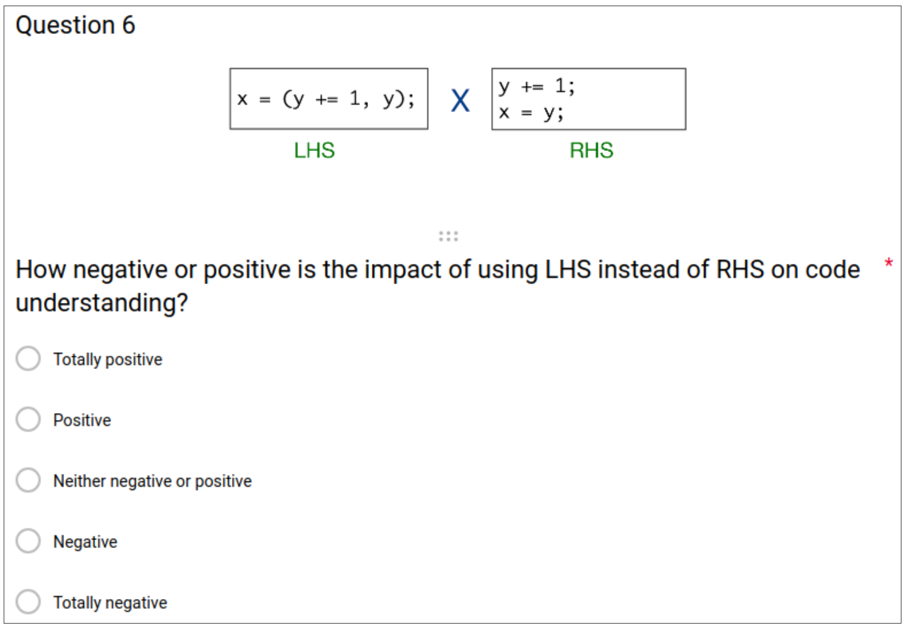
```
x || y && z;
```

X

```
x || (y && z)
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ○ Totally positive
- ○ Positive
- ○ Neither negative or positive
- ○ Negative
- ○ Totally negative



Question 6

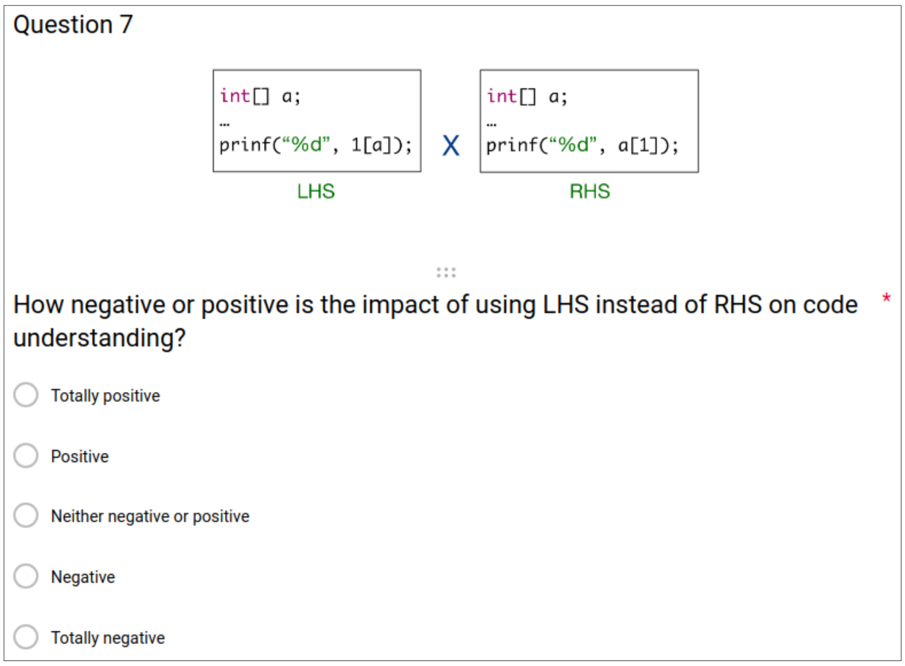
```
x = (y += 1, y);
```

X

```
y += 1;
x = y;
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ○ Totally positive
- ○ Positive
- ○ Neither negative or positive
- ○ Negative
- ○ Totally negative



Question 7

LHS

```
int[] a;
…
prinf(“%d”, 1[a]);
```

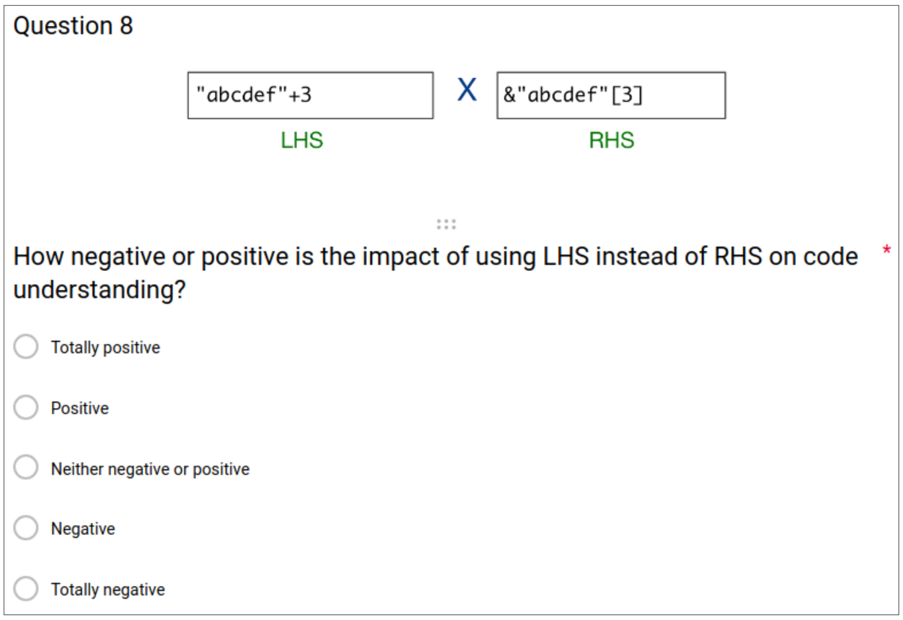
X

RHS

```
int[] a;
…
prinf(“%d”, a[1]);
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ◯ Totally positive
- ◯ Positive
- ◯ Neither negative or positive
- ◯ Negative
- ◯ Totally negative



Question 8

LHS

```
"abcdef"+3
```

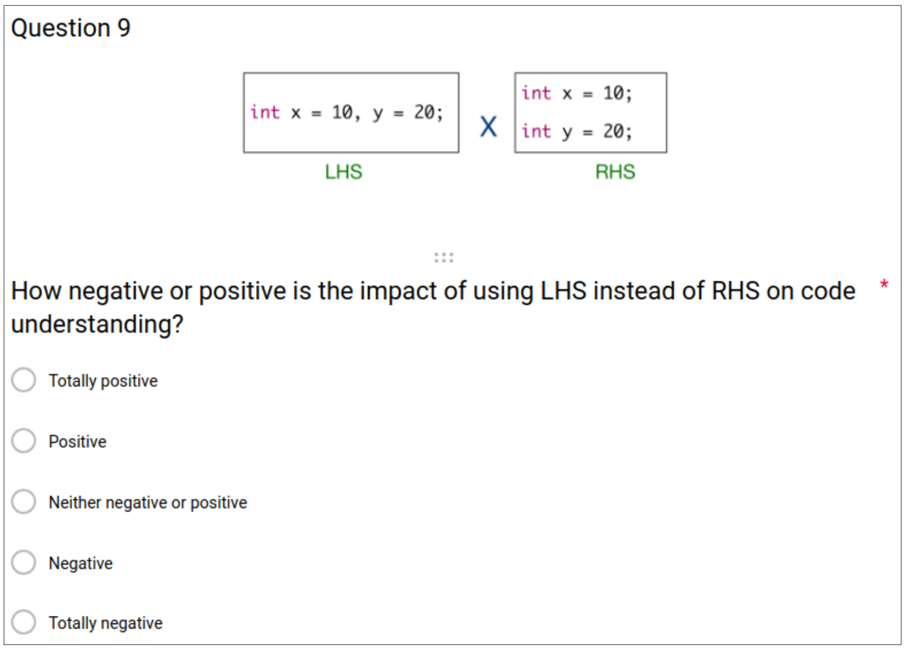
X

RHS

```
&"abcdef"[3]
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ◯ Totally positive
- ◯ Positive
- ◯ Neither negative or positive
- ◯ Negative
- ◯ Totally negative



Question 9

```
int x = 10, y = 20;
```

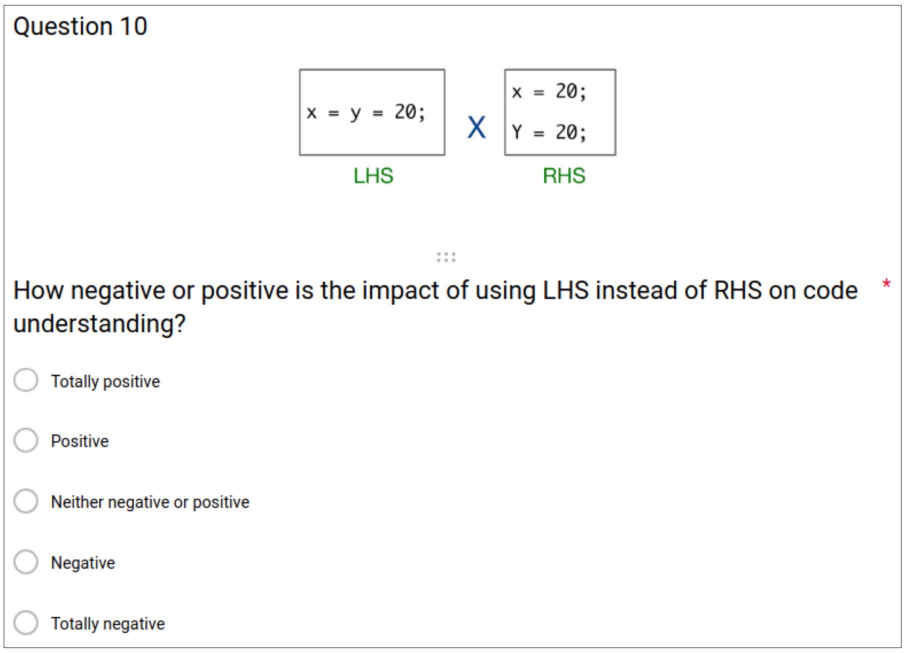
X

```
int x = 10;
int y = 20;
```

LHS RHS

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ◯ Totally positive
- ◯ Positive
- ◯ Neither negative or positive
- ◯ Negative
- ◯ Totally negative



Question 10

```
x = y = 20;
```

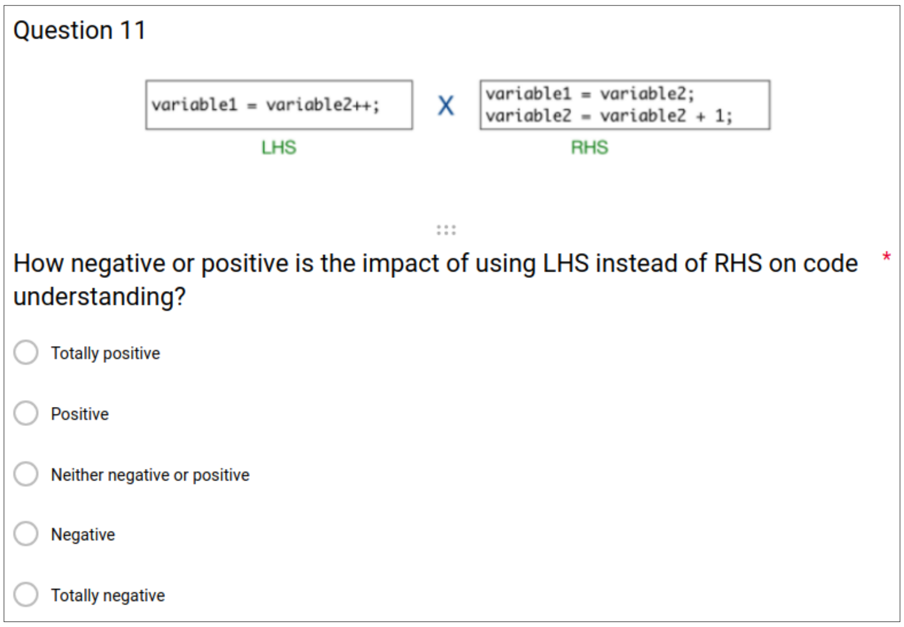
X

```
x = 20;
Y = 20;
```

LHS RHS

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

- ◯ Totally positive
- ◯ Positive
- ◯ Neither negative or positive
- ◯ Negative
- ◯ Totally negative



Question 11

LHS:

```
variable1 = variable2++;
```

X

RHS:

```
variable1 = variable2;
variable2 = variable2 + 1;
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

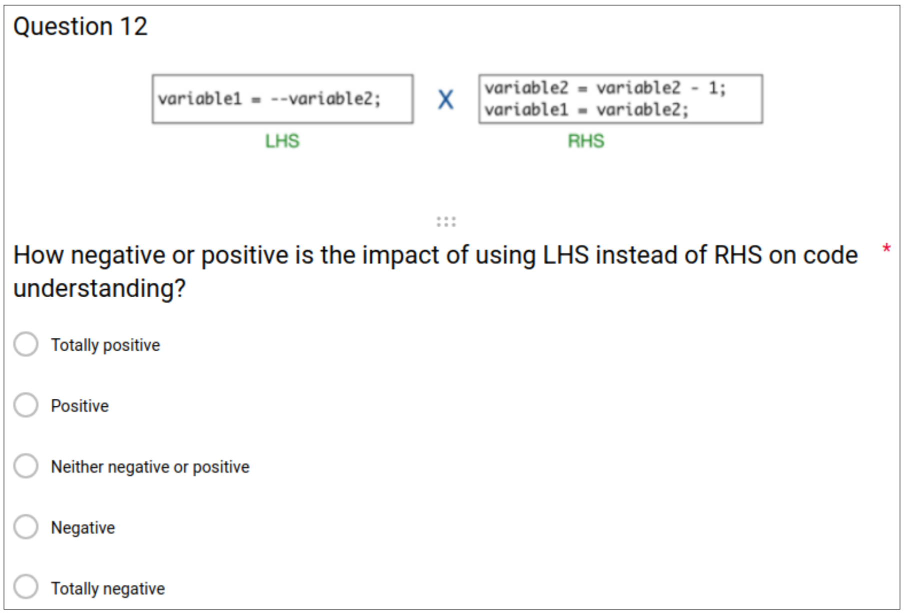
○ Totally positive

○ Positive

○ Neither negative or positive

○ Negative

○ Totally negative



Question 12

LHS:

```
variable1 = --variable2;
```

X

RHS:

```
variable2 = variable2 - 1;
variable1 = variable2;
```

How negative or positive is the impact of using LHS instead of RHS on code understanding? *

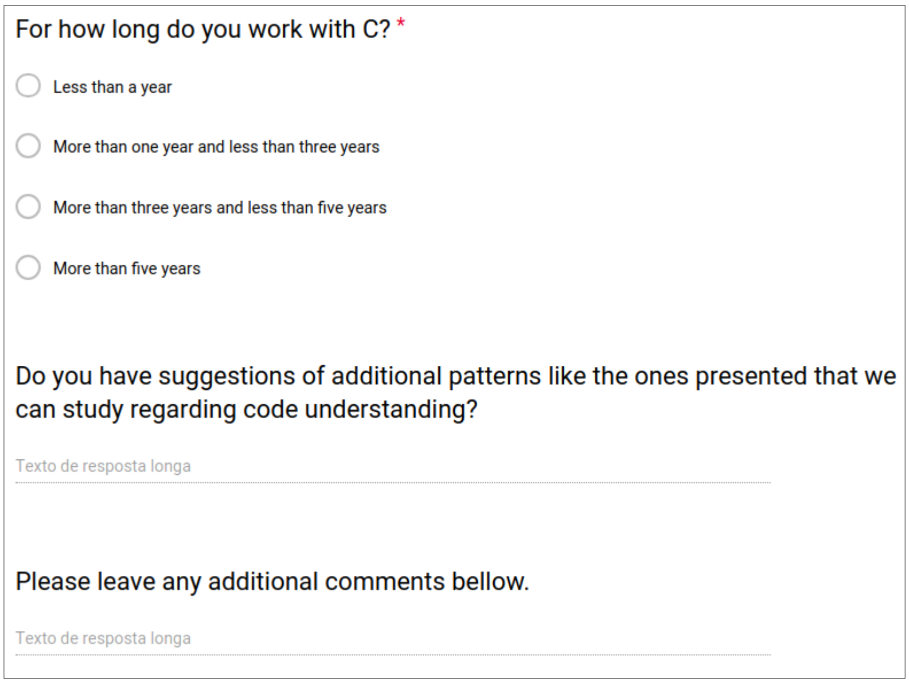
○ Totally positive

○ Positive

○ Neither negative or positive

○ Negative

○ Totally negative



For how long do you work with C? *

- ◯ Less than a year
- ◯ More than one year and less than three years
- ◯ More than three years and less than five years
- ◯ More than five years

Do you have suggestions of additional patterns like the ones presented that we can study regarding code understanding?

Texto de resposta longa

Please leave any additional comments bellow.

Texto de resposta longa