Should We Argue for Better Compiler Error Messages?

***Abstract*—Compilers primarily give feedback about problems to the developer through the use of error messages. Unfortunately, developers routinely find these messages to be confusing and unhelpful. In this paper, we postulate that because error messages are intended to convince developers of problems in their code, these messages can be framed as arguments. To understand how error messages should present arguments to developers, and to investigate the applicability of argumentation theory to compiler error messages, we conducted an empirical study of 210 question-answer pairs, tagged as compiler errors, from Stack Overflow.**

Our findings indicate that the argument layout of compiler error messages is significantly different from the argument layout of Stack Overflow accepted answers, suggesting that error message arguments are misaligned with the way in which developers expect arguments to be presented. For example, in contrast to human-authored answers, compiler error messages typically do not provide a rationale for why the problem is a problem. Our long-term goal is to enable compilers to automatically present error messages as arguments. The results of our study provide empirical justification and support towards an underlying argumentation theory to achieve this goal.

I. Introduction

Compilers primarily give feedback about problems to developers through the use of error messages. Despite the intended utility of error messages, the difficulty of comprehending and resolving compiler error message is pervasive. Researchers and practitioners alike have described their output as "cryptic" [1], "difficult to resolve" [1], "not very helpful" [2], "appalling" [3], "unnatural" [4], and "basically impenetrable" [5].

In this paper, we postulate that because error messages are intended to *convince* the developer that a problem exists in their code, these error messages can be framed as arguments and understood in terms of argumentation theory. Through this perspective, the difficulties developers encounter with existing error messages can be explained by the fact that error messages present poor arguments.

To understand how error messages should present arguments to developers, and to investigate the applicability of argumentation theory to compiler error messages, we conducted an empirical study through a popular question-and-answer site, Stack Overflow.[1] From Stack Overflow, we extracted 210 question-answer pairs posted by developers about compiler error messages, across seven different programming languages. For every question-answer pair, we qualitatively coded the compiler error message found within the question, and the human-authored answer, using components from argumentation theory (Section III).

[1]https://www.stackoverflow.com

The results of our study provide empirical support for presenting compiler error messages to developers as arguments. Specifically, we find that:

- The argument layout, or macrostructure, for compiler error messages is significantly different from the argument layout of Stack Overflow accepted answers, suggesting that error message arguments are misaligned with the way in which developers expect arguments to be presented (Section VI-A). An additional qualitative analysis of microstructure identifies how accepted answers support the macrostructure layout, for the domain of compiler error messages (Section VI-C).
- Human-authored answers for compiler error messages use complementary argument layout structures to argue different types of errors (Section VI-B). For issues that have relatively simple resolutions, arguments are presented primarily in terms of the actionable fixes that resolve the defects—without rationale for the underlying cause of the problem. For more difficult problems, human-authors employ argument layouts that model conventional argumentation theory layouts.

Our long-term goal is to enable compilers to automatically present error messages as arguments that are comparable to the human-authored answers found on Stack Overflow. Towards this goal, our study provides the empirical justification and the underlying argumentation theory needed to provide argumentation-based diagnostics in compilers.

II. Motivating Example

To illustrate how confusion with error messages can arise, consider the hypothetical developer, James, who hammers out the following short code snippet into his IDE:[2]

```
enum Color { RED, GREEN, BLUE }
String colorString(Color c) {
    switch(c) {
        case RED: return "red";
        case GREEN: return "green";
        case BLUE: return "blue";
    }
}
```

Essentially, James has created an enumeration consisting of three constants, `RED`, `GREEN`, and `BLUE`—and has handled each

[2]This example is found in "Ensuring completeness of switch statements" from IBM [6]. For brevity, we've also removed the surrounding Java boilerplate.

of the cases exhaustively within a `switch` statement in his `colorString` method.

James is surprised, however, to discover that Eclipse emits an error message:

```
1. ERROR in Example.java (at line 5)
        String colorString(Color c) {
               ^^^^^^^^^^^^^^^^^^^^
This method must return a result of type String
```

This message claims that the code is missing a return statement, and James reasons that there must be some other case which he has not considered—causing the switch statement to fall through. He briefly considers that `null` is the missing case, and therefore adds a `return null` statement to the line just before the indication error location. Unfortunately, James determines that this would never be reached if `c` were in fact `null`, because the switch statement would throw a `NullPointerException` before ever reaching the `return`.

James is perplexed since he can think of no obvious missing `return` condition that could ever execute other than those he has already enumerated. Consequently, James finds it difficult to accept this explanation—particularly since this is a degenerate case of pattern matching over a bounded range (that is, the enumeration), a problem known to be computationally decidable [7]. Perhaps, James concludes, Eclipse simply does not implement this capability in its compilation analysis.

Lacking any additional assistance from the compiler other than the error message, James constructed two possible explanations for why the compiler emitted the error message. In the first explanation, James concluded that the error message was a result of an error on his part—that he had in fact missed some edge case, such as `null`, or perhaps some other case that can occur with enumerations that he must not be aware of. In the second explanation, James concluded that he had in fact exhaustively matched all cases, but that the compiler was simply not sophisticated enough in its compilation analysis to know this. Unfortunately, though both arguments seem plausible, neither are actually correct.

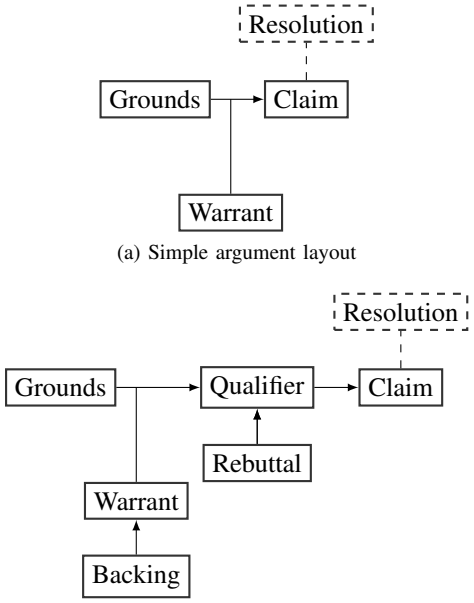
It turns out that the *actual* reason for the compiler error is that the Java language specification requires each `enum` statement without a default case to be considered as incomplete, *even when* it exhaustively covers all of the enumeration constants. Not only did James *not* miss any cases in his switch statement, his conclusion about the Eclipse compiler was also incorrect: Eclipse has the machinery to perform enumeration checking. Thus, using a different compiler would not have helped James to avoid hitting the error.

Imagine how much time, effort, and confusion might have been alleviated if James had been able to ask the compiler to elaborate on its error message, perhaps through a compiler flag:

```
1. ERROR in Example.java (at line 6)
        switch(c) {
               ^
The switch over the enum type Eclipse.Color
exhaustively covers all cases.

However, the Java specification requires that
enum statements without a default case statement
be considered incomplete, even when all cases
are covered.

Thus, the switch statement should provide a
default case, or the method should return a
result of type String for all execution paths.
```



(a) Simple argument layout

(b) Extended argument layout

Fig. 1. A prototypical Toulmin's model of argumentation for (a) simple argumentation layout, and (b) extended argument layout. The possible need for auxiliary steps to convince the other party yields the extended argument layout.

Had the compiler present a better-supported argument, James would likely have more easily understood why the problem was a problem in the first place. Additionally, James would have had the necessary justification, provided by his compiler, for why he needs to address the problem.

III. Background on Argumentation Theory

Argumentation theory provides a lens through which we can evaluate the effectiveness of arguments—a communicative means to convince one party to accept a position from another party [8], [9]. If, as early work by Dean suggests—-that researchers could better understand how computers should "talk" to people by investigating how ordinary people talk to each other—then the same proposition should hold true when these "talks" are presented in the form of an argument [10].

Within argumentation theory, the Toulmin model of argument posits one such model precisely for this style of argument: an informal reasoning model which characterizes everyday arguments, or how arguments occur in practice [11]. Specifically, the Toulmin model of argument is a *macrostructure* model. Macrostructure models examine how components

```
Error:(31, 58) java: incompatible types(C):
bad return type in lambda expression(bc W, G)
  java.lang.String cannot be converted to void(B)
```

Fig. 2. A compiler error message from Java, annotated with argumentation theory constructs. This particular message contains all of the basic argumentation constructs to satisfy Toulmin's argument: `(C)` = Claim, `(bc W)` = implied "because" Warrant, `(G)` = Grounds. It also includes an extended construct, `(B)` = Backing.

combine to support the larger argument rather; in contrast, *microstructure* examines the phrasing and composition of the "sentence-level" statements.

In the simple argument layout (Figure 1a), the components consist first of a *claim*—the assertion, view or judgment to be justified. The second component is *ground*, or data that provides evidence for this claim. The third component is the justification or *warrant*, which acts as a bridge between the grounds and the claim (for example, "[claim] because [data]"). Together, the claim, the grounds, and the warrant provide a simple argument layout.

Toulmin also devised an extended model of argument, to acknowledge the possibility of needing auxiliary steps to the simple argument layout (Figure 1b). In addition to the simple argument layout components, the extended argument layout offers a *rebuttal* when an exception has to be inserted into the argument. The claim may also not be absolute: in this case, a *qualifier* component can temper the claim. Finally, the warrant may also not be immediately accepted by the other party, in which case additional *backing* is needed to support the warrant.

For both models, we made one tweak that is peculiar to programming tasks. We observed in our own Stack Overflow dataset that some accepted answers provided resolutions in their answer. Strictly speaking, a resolution is not an argument, but suggested resolutions appear frequently enough in both compiler error messages and Stack Overflow answers that we felt it important to capture this component within our analysis—they are clearly important to supporting developers, whether they are formally an argument or otherwise.

IV. Methodology

A. Research Questions

In this study, we investigate the following research questions and offer the motivation for each:

RQ1: Are unsatisfactory compiler error messages and satisfactory Stack Overflow answers explainable in terms of argument layout? If compiler error messages and Stack Overflow accepted answers use significantly different argument layout components, this would suggest that *macrostructure* differences in argumentation play an important role in confusion developers face with compiler errors. While some approaches to improving compiler error messages focus on microstructure (for example, "confusing wording" in the message [12], [13]), macrostructure differences emphasize how components combine to support the larger argument rather than the statements themselves. Microstructure improvements may be ineffectual without supporting macrostructure layout. Unfortunately, determining whether the argument layout of the two groups is significant turns out to be more complicated than expected.

TABLE I
Compiler Errors and Warnings Count by Tag

	Question Count[2]			
Tag[1]	Errors[3]	Warnings[4]	Total	% Accepted[5]
C++	3508	421	3929	63%
Java	2078	170	2248	55%
C	1179	286	1465	61%
C#	783	122	905	69%
Objective-C	270	109	379	65%
Swift	246	17	263	56%
Python	211	4	215	53%
Extracted Data[6]	11736	1553	13289	58%

[1] Programming languages are indicated in bold.
[2] Questions may be counted more than once if they have multiple tags, for example, `Java` and `Eclipse`.
[3] Questions tagged as `compiler-errors`.
[4] Questions tagged with `compiler-warnings`, but not `compiler-errors`.
[5] Percentage of questions tagged as `compiler-errors` and `compiler-warnings` that have accepted answers.
[6] Description of extracted data set.

RQ2: What argument layouts are used in compiler error messages and Stack Overflow accepted answers? The answer to this question helps us to understand the types of argument layouts that are used in accepted answers. In other words, toolsmiths can use the design space of argument layout to model and structure automated compiler error messages for developers. Importantly, the argument layout space can also be used as a means to evaluate existing error messages, and to identify potential gaps in argument components for these messages.

RQ3: How are the components of argument layout instantiated for presenting compiler error messages? Once the macrostructure argument layouts are identified, learning how the components within these layouts are instantiated provide microstructure details for what information developers find useful within each component. For example, one way to instantiate backing for a warrant might be to provide a link to external documentation—and if we find that accepted answers do so, toolsmiths may also consider incorporating such information in the presentation of their compiler error messages.

B. Study Design

Research context. Previous research by Treude and colleagues identified questions regarding error messages as being one of the top categories. [14], and other research supports that Stack Overflow today is a primary resource for software engineering problems [15]. Additionally, Stack Overflow provides an open-access API, through Stack Exchange Data Explorer,[3]

[3] http://data.stackexchange.com/

that allows researchers to mine their database. An initial query against this dataset confirmed that questions about compiler error messages exist in Stack Overflow across a diversity of programming languages and platforms.

Data collection. Using the Stack Exchange Data Explorer[4], we extracted all posts of type question or answer, tagged as "compiler-errors" or "compiler-warnings." We added compiler warnings because some systems allow the developer to flags warnings as errors, and thus we included these in our set.

A subset of these questions link to an associated *accepted answer*, which in this paper we term *question-answer pairs*. An accepted answer is an answer marked by the original questioner as being satisfactory in resolving or addressing their original question. Although a question may have multiple answers, only one may be marked as accepted. We used accepted answers as a proxy to identify helpful answers.

For each question, we extracted the compiler error message from the compiler used in the question. If the question did not contain a compiler error message, the question-answer pair was dropped from analysis.

Sampling strategy. To obtain diversity across programming languages, we used stratified sampling across the top languages on Stack Overflow for compiler errors, until the rank covered over 95% of all of the messages. This threshold was exceeded at Python (Table I). Within each strata, we used simple random sampling for selecting question-answer pairs to analyze, in which each question-answer pair has an equal probability of being selected. As we sampled, we discarded questions that did not refer to or display a specific error message, were incorrectly tagged (for example, not relating to an error message), were related to issues in not being able to invoke the compiler in the first place (for example, "g++ not found"), or question-answer pairs that are unambiguously "trolling," [16] such as through deliberately bogus questions.[5] We continued this process until we obtained 30 question-answer pairs for each of the top seven languages, for a total of 210 question-answer pairs.

Qualitative closed coding. The first and second authors performed closed coding, that is, coding over pre-defined labels, for compiler error messages extracted from the Stack Overflow question and over the complete Stack Overflow accepted answer for that question. We tagged each using labels from the Toulmin model of argument: claim, grounds, warrant, qualifier, rebuttal, and backing. Our preliminary coding also identified a form of evidence necessary to support error messages: *resolutions*, or fix suggestions. Thus, we had a total of seven labels, and a compiler error message or Stack Overflow accepted answer may be assigned more than one label.

During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [17]. Using this technique, the first and second authors collaboratively code to achieve agreement and to clarify the definitions of the codes. We coded the first 20% of messages using the negotiated agreement technique, and then independently coded the remaining messages.

Supporting verifiability. To support verifiability, we provide supporting online materials containing verbatim transcripts of the Stack Overflow question-answer pairs, as well as the intermediate analysis which led to our findings, which other researchers may audit.[6] If using a supported PDF reader, quotations from Stack Overflow are hyperlinked and can be clicked to take the reader to the corresponding Stack Overflow page.[7].

V. Analysis

A. RQ1: Are unsatisfactory compiler error messages and satisfactory Stack Overflow answers explainable in terms of argument layout?

For this research question, we want to quantify whether the components used in argument layout between compiler error messages and Stack Overflow answers can explain difficulties developers have with compiler error messages. That is, do error messages and Stack Overflow answers employ significantly different argument layouts. To quantify these differences, we apply a statistical, permutation testing approach by Simpson and colleagues that allows comparison across two groups when each observation in the group is an ordered set [18].

To leverage this analysis framework, an error message, whether it is a compiler error message extracted from a Stack Overflow question (E_Q) or a Stack Overflow accepted answer (E_A), is represented as an ordered set in terms of argumentation components:

$$E = \langle a_1, a_2, \ldots, a_n, r \rangle \tag{1}$$

where $a_1, a_2, \ldots, a_n$ are the labels for the argument components, such as grounds, warrants, and backing, and r is an extended resolution component. For each component, a binary `true` or `false` indicates the presence or absence of the component within the argument.

Given any two error messages, E_1 and E_2, represented as an argumentation set, we can now calculate the Jaccard index, a metric that represents the similarity between two sets:

$$J = \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|} \tag{2}$$

The Jaccard index ranges from 0, indicating no overlap, to 1, indicating perfect overlap between the sets. Intuitively, it is simply the intersection over the union of the sets.

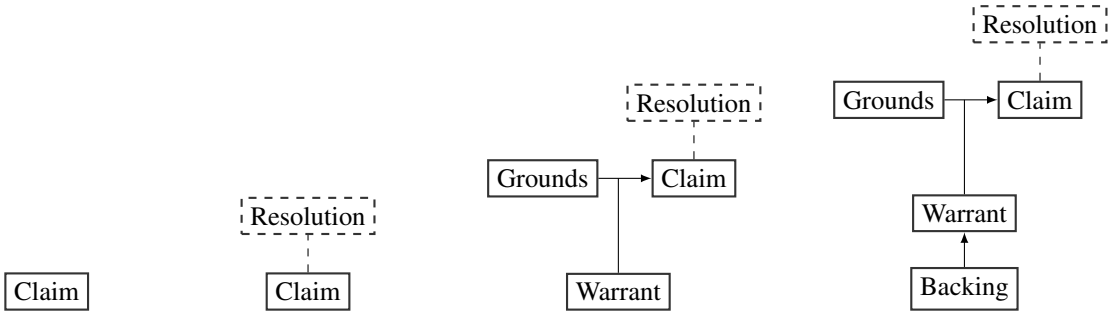
Next, we need to calculate a way to quantity difference between *groups*, rather than individual error messages. To do so, we calculate two intermediate measures, $M_j(\text{Within})$ and $M_j(\text{Between})$. $M_j(\text{Within})$ is the arithmetic mean of the

[4]https://data.stackexchange.com/

[5]For example, the post "Why is this program erroneously rejected by three C++ compilers?" attempts to compile a hand-written C++ program scanned as an image, through three different compilers. The offered answers are equally sardonic. (http://stackoverflow.com/questions/5508110/)

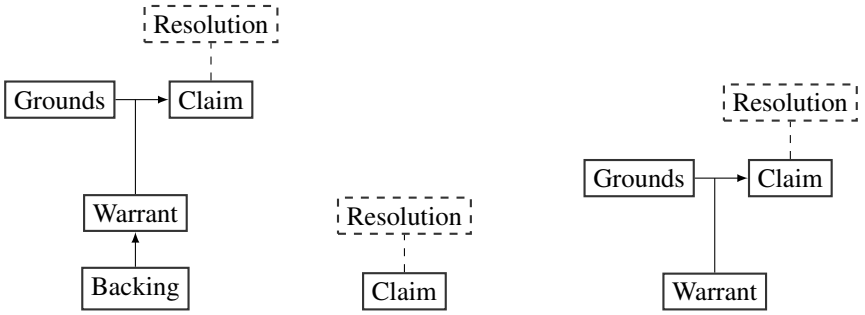
[6]URL to be provided after blind review.

[7]These references are indicated as Q*:id* or A*:id*, and can be directly accessed through https://www.stackoverflow.com/questions/:id



(a) Claim-only (191) (b) Claim-resolution (128) (c) Simple argument layout (8) (d) With backing (1)

Fig. 3. Identified argument layouts for compiler error messages (as found in Stack Overflow questions). Counts are indicated in parentheses.



(a) With backing (102) (b) Claim-resolution (59) (c) Simple argument layout (49)

Fig. 4. Identified argument layouts for Stack Overflow accepted answers. Counts are indicated in parentheses.

Jaccard indices, for all possible pairings of sets, that occur within a group. In our case, this would be all of the pairings that occur within compiler error messages, *and* all of the pairings that occur within Stack Overflow accepted answers. $M_j(\text{Between})$ is the arithmetic mean of Jaccard indices, for all possible pairings that occur *between* the two groups. Together, $M_j(\text{Within})$ and $M_j(\text{Between})$ are combined to form a Jaccard ratio:

$$R_J = \frac{M_j(\text{Within})}{M_j(\text{Between})} \tag{3}$$

Having the Jaccard ratio, R_J, we now want to determine if the ratio is significant. However, since we do not know what the true distribution is under the null hypothesis—no significant group difference—we use permutation testing to empirically generate this distribution. To do so, we randomly flip the labels within pairs at random, and then calculate ${R_J}^{\text{perm}}$, N_{perm} times. The larger the number of permutation calculations, the more precise the empirically p-value becomes. Although Simpson and colleagues provide theoretical bounds for N_{perm}, in practical terms, setting the number of permutations to an arbitrarily large numbers—such as 10,000—is sufficient to obtain a reasonable p-value.

Having a list of ${R_J}^{\text{perm}}$ (of length N_{perm}), we can now calculate the p-value directly by counting the number of instances where ${R_j}^{\text{perm}} > R_J$ over the number of permuted instances:

$$p = \frac{\#({R_j}^{\text{perm}} > R_J)}{N_{\text{perm}}} \tag{4}$$

If the p-value is less than α (we use $\alpha < 0.05$), the null hypothesis is rejected and we can conclude that compiler error messages and Stack Overflow accepted answers are significantly different in terms of argumentation macrostructure.

B. RQ2: What argument layouts are used in compiler error messages and Stack Overflow accepted answers?

To understand what types of arguments layouts are effective in obtaining the status of accepted answer, we used quasi-statistics—essentially, transforming qualitative data to simple counts—to aid the interpretations of the Stack Overflow data [19]. In this analysis, we merged identical sets, that is, sets with the same ordered values for presence or absence, to form argumentation templates. Then, we treated negligible components—those components with few counts—as wildcards to further reduce the argument layout space. In theory, given an ordered set of size n, 2^n permutations of sets are possible to obtain. In practice, if error messages and Stack Overflow answers can be framed in terms of argumentation theory, the set of observed set arrangements should be significantly fewer—in fact, if they are reasonable argument layouts then they should more or less resemble variations of the simple argument layout or the extended argument layout (Section III).

C. RQ3: How are the components of argument layout instantiated for presenting compiler error messages?

To identify the *microstructure* of argument, that is, the techniques developers use within the argumentation components, we performed a second qualitative coding exercise over the first closed coding. For this analysis, we performed descriptive coding to label the types of evidence provided within the accepted answers [20]. As a concrete example, the argumentation component of *backing* can be provided by through pointing to a local or program element in the code (blame), through a code example that provides evidence for the problem, or through external resources, such as language-specification documentation.

In addition to random sampling, we performed *purposive sampling*, or non-probabilistic sampling, on question-answer pairs to compose *memos* [21]. These memos captured interesting exchanges or properties of the question-answer pairs to promote depth and credibility, and to frame the information needs and responses posters' through their reported experiences. That is, they provide a *thick description* to contextualize the findings [22].

VI. Results

A. RQ1: Are unsatisfactory compiler error messages and satisfactory Stack Overflow answers explainable in terms of argument layout?

The Jaccard ratio of the two groups are $R_j = 1.6441$, with permutation testing yielding a significant difference between the two groups (for repeated iterations, $p = 0.008 \pm 0.001$). Because we have computed a pair-wise statistic, the implication is that the compiler error message and Stack Overflow accepted answers are significantly different in terms of argumentation layout than the compiler error message provided to the question.

Because the questioner asked a question about the compiler error message, this indicates some confusion with the error messages they were presented with. Because the same questioner also marked the Stack Overflow answer as accepted, we can assume that the answer has resolved whatever confusion they had in the original question. Since the argument structure between the compiler error message and the accepted answer are significantly different in terms of argument layout, we can conclude that differences in macrostructure argument layout can be attributed to the acceptance of the Stack Overflow answer.

B. RQ2: What argument layouts are used in compiler error messages and Stack Overflow accepted answers?

After performing the argumentation set merge operations for identical argumentation sets, we further treated the components of qualifiers and rebuttal as wildcards because we found negligible instances of these components compared to the other argument structure categories.

The results of this merge as found in Figure 3 and Figure 4, for compiler error messages and for Stack Overflow accepted answers, respectively. For each the group, the argument layouts are ordered from most frequently observed to least frequently observed. In this quasi-statistical reporting, it is clear to see why the argument layout for compiler errors and Stack Overflow accepted answers were found to be significantly different in RQ1: compiler error messages predominantly present a claim with no additional information, and occasionally present a resolution (that is, a fix) to resolve the claim. In contrast, Stack Overflow accepted answers are inverted in argumentation layout frequency; the most frequent argument layout extends simple argument layout with backing, and least frequently provides solely a resolution for the claim. In our investigation, we did not find any instances in which Stack Overflow accepted answers solely rephrased the compiler error message (that is, the claim-only layout).

Thus, not only do Stack Overflow accepted answers more closely align with Toulmin model's of argumentation, these answers satisfactory resolved the confusion of the developer when the original compiler error message did not.

TABLE II
Argument Layout Components for Error Messages

Attribute	Description
Simple Argument Components	
Claim (Section VI-C1)	The claim is the concluding assertion or judgment about a problem in the code.
Ground (Section VI-C2)	Facts, rules, and evidence to support the claim.
Warrant (Section VI-C3)	Bridging statements that connect the grounds to the claim. Provides justification for using the grounds to support the claim.
Extended Argument Components	
Backing (Section VI-C4)	Additional evidence to support the warrant, if the warrant is not accepted.
Qualifier (Section VI-C5)	This is the degree of belief for a claim, used to weaken the claim.
Rebuttal (Section VI-C6)	Exceptions to the claim or other components of the argument.
Resolution	
Resolution (Section VI-C7)	Resolutions suggest concrete actions to the source code to remediate the problem. Strictly speaking, not an argumentation component, but can be used to short-circuit an argument.

C. RQ3: How are the components of argument layout instantiated for presenting compiler error messages?

In this section, we describe the layout argument components, grounding its attributes in the contextual details we obtained from our qualitative analysis. An overview of the argument structure is presented in Table II.

1) Claim: Because of the layout of Stack Overflow, accepted answers assume that the developer has read the error message in the question, and will refer to the claim without explicit antecedent. For instance, the answer may say, "This problem" (A1225726) or "This issue" (A32831677) or

immediately chain from the question to the connect their ground and warrant (A28880386). We did however, encounter instances where developers explained error messages through first *rephrasing*, such as "it means that" (A16686282) and "is saying" (A20858493)—usually for the purpose of simplifying the jargon in the message or making an obtuse message more conversational. For example, the compiler error message `foreach statement cannot operate on variables of type because does not contain a public definition for 'GetEnumerator'` is rephrased by the accepted answer as "It means that you cannot do *foreach* on your desired object, since it does not expose a `GetEnumerator` method." Compiler authors like Czaplicki (of the Elm programming language) have also noted that error messages should be more conversational and human-like [23].

2) Ground: Grounds are an essential building block for convincing arguments; they are the substrate of declarative facts—which bridged by the warrant—support the claim, that is, the compiler error message. For example, "the variable is non-static private field" (A4114006), "`clone()` returns an `Object`" (A3941850), "`foo<T>` is a base class of `bar<T>`" (A27412912), "[t]he only supertype of `Int` and `Point` is `Any`" (A2871344), "local variables cannot have external linkage" (A5185833) all refer to grounds about the state of the program or rules about what the compiler will accept.

Consider the use of `gets()` in a C program, which in the `gcc` compiler generates the message:

```
test.c:27:2: warning: gets is deprecated
  (declared at /usr/include/stdio.h:638)
  [-Wdeprecated-declarations]

gets(temp);
^
```

The poster of the compiler error wants to suppress this warning (Q26192934), but the accepted answer explains the grounds for this warning (A26192934): "gets is deprecated *because* it's dangerous, it may cause buffer overflow."

3) Warrant: In argumentation theory, warrants are bridge terms, such as "since" or "because" that connect the ground to the claim. Often, the warrant is not explicitly expressed, and the connection between the ground and the claim must be implicity identified [9]. During our analysis, we would insert implicit "since" or "because" phrases during reading of the error message or Stack Overflow answer to identify implicit warrants.

In some compilers, messages can bridge grounds with warrants through explicit concenations, such as with the "reason:" error template in Java:

```
Test.java:6: error: method b in class Test cannot
be applied to given types;
        b(newList(type));
        ^
  required: List<T>
  found: CAP#1
  reason:
    inference variable L has incompatible bounds
    equality constraints: CAP#2
    upper bounds: List<CAP#3>,List<?>
  where T,L are type-variables:
    T extends Object declared in method <T>b(List<T>)
     ...
```

Unfortunately, the grounds for this warrant are particularly dense in itself. However, warrants needs not always be this obtuse, as the following C++ message from OpenCV indicates:

```
OpenCV Error: Image step is wrong
    (The matrix is not continuous,
    thus its number of rows can not be changed).
```

Here, the warrant is bridged through the use of the parenthetical statement.

4) Backing: A backing may be required in an argument if the warrant is not accepted; in this case, the backing is additional evidence needed to support the warrant. In practice, one should selectively support warrants; otherwise, the argument structure grows recursively and quickly becomes intractable [9]. For presenting error messages, we found that while warrants were typically additional statements, backing was provided through the use of resources. These resources include code examples or code snippets (A2640738, A1811168), references to the programming language specifications (A5005384), and occasionally, bug reports (A37830382) as well as tutorials (A2640738).

5) Qualifiers: Despite the usefulness of static analysis techniques for reporting compiler error messages to developer, many classes of analysis feedback are undecidable or computationally hard, which necessitate the use of unsound simplifications [24]. Qualifiers include statements like "should" (A29189727), "likely" (A17980236), "try" (A7316513), and "probably" (A2841647, A7328052. A7942837). Although we found such usages throughout Stack Overflow, it was difficult for us to determine if these usages are simply used as casual linguistic constructs (essentially, fillers) or if the answer actually intended to convey a judgment about belief. We did, however, find several examples when developers were confused because the wording of the compiler error made the developer believe that their own judgment was in error—for example, from the story of our hypothetical developer James in Section II), whose experiences in based on questions such as Q5013194 and Q36476599.

6) Rebuttal: We found few instances of rebuttals within Stack Overflow accepted answers, and one of the reasons we believe rebuttals to be relatively infrequency is that an author must have an expectation of what to rebut in order to provide a rebuttal in the first place. Thus, we interpreted rebuttals liberally as statements in which an answer would retract a particular ground or resolution due to a particular constraint—for example, due to a bug in the compiler (A2858799, A1167204). Another means of rebuttal occurs when the accepted answer provides reasons for *ignoring* a

claim, as in A11180068. Here, the accepted answer suggests downgrading a ReSharper warning from a warning to a hint in order to not get "desensitized to their warnings, which are usually useful."

7) Resolution: Although not present in Toulmin's model of argument, one way in which error messages can convince developers of an argument is to offer the solution to that argument that resolves their issue. Typically, Stack Overflow accepted answers provide these resolution in a style similar to "Quick Fixes" in IDEs—they briefly describe what will be changed, show the resulting code after applying the change, and demonstrate that the compiler defect will be removed as a result of applying the change. A prototypical example of how answers provide resolutions is found in A8783019. Here, the answer notes, "You're missing an & in the definition." The answer then proceeds to show the original code:

```
float computeDotProduct3(Vector3f& vec_a,
  Vector3f vec_b) {
```

against the suggested fix:

```
float computeDotProduct3(Vector3f& vec_a,
  ector3f& vec_b) {
```

VII. Limitations

The sole use of Stack Overflow, combined with qualitative research methodologies, introduces trade-offs in the design and reporting of our study.

Survivorship bias. The question-answer pairs on Stack Overflow exhibit survivorship bias, in that they only reflect questions that were not easily answered through means; thus these questions are posted only after exhausting other possibilities. The consequence of survivorship bias is that the compiler error messages within our data set may in some ways be pathological: they represent the extreme cases in which developers must resort to external help, which may not occur routinely during normal software development.

Likewise, there is also known bias in when and why developers choose to answer questions. To illustrate, Mamykina and colleagues identified situations in which some questions never receive a response, for example, when they are about relatively obscure technologies, when there are few users for a particular topic, and when questions are tedious to answer [15].

The effect of this bias is that our results likely underestimate the effectiveness of existing compiler error messages, because only those that are pathological have questions posted about them. Thus, our results should be interpreted as being conditioned on messages already-known to be problematic, rather than generalized to all compiler error messages.

Self-selection bias. A second type of bias manifests through self-selection bias, since participants in Stack Overflow voluntarily choose to post to Stack Overflow. Consequently, the types of participants that post to Stack Overflow may have an affect on both the barriers that developers face as well as the way in which accepted answers are presented. As one example, Ford and colleagues identified a variety of factors, such as "fear of negative feedback" and "intimidating community size", as inhibiting potential posters from asking their questions on Stack Overflow [25]. Yet other users are "one-day flies," posting once and only Stack Overflow, who have been found to post lower-quality questions than other types of users [26]. Consequently, our analysis may fail to identify pre-cursor barriers that exist even before a developer chooses to post to Stack Overflow, unless a developer explicitly reports within the question.

Identifying argument microstructure. The design space of argument microstructure is constrained to available affordances in Stack Overflow. For example, answers in Stack Overflow must use mostly text notation, although past research has found that developers sometimes place diagrammatic annotations on their code to help with comprehension [27]. Similarly, Flanagan and colleagues uses a diagrammatic representation on the source code to help developers understand code flow for an error [28]. Other tools like Path Project [29] and Theseus use visual overlays on the source code, which are not expressible within Stack Overflow except through rudimentary methods like adding comments to the source. Thus, the design space of attributes is biased towards linear, text-based representations of compiler error messages.

Generalizability. As a qualitative approach, our findings do not offer external validity in the traditional sense of nomothetic, sample-to-population, or *statistical generalization*. For example, we cannot claim that the differences in design space usage between error messages and Stack Overflow accepted answers generalize to those outside the ones we observed within our study. That is, our findings are embedded within Stack Overflow and contextualized to understand a particular aspect of developer experiences as they comprehend and resolve compiler error messages within these question-answer pairs. As one example, the argument layout for compiler error messages is likely to significantly underrepresent claim-resolution layouts, as resolutions in integrated development environments appear in a different location—such as Quick Fixes in the editor margin—than the compiler error message.

In place of statistical generalization, our qualitative findings support an idiographic means of satisfying external validity: *analytic generalization* [30]. In analytic generalization, we generalize from individual statements within question-answer pairs to broader concepts or higher-order abstractions through the application of argumentation theory.

VIII. Related Work

We examine related work from for three research areas: research that has identified comprehension barriers for compiler error messages, research proposing principles or guidelines for the design of compiler error messages, and other qualitative research using Stack Overflow as a means to inform software development.

A. Barriers to Error Message Comprehension

Johnson and colleagues conducted an interview study with developers to identify barriers to using static analysis tools.

Their interviewers reported barriers such as "poorly presented" tool output and "false positives" as contributing to comprehension difficulties [13]. One interviewer suggested that that error messages be presented in some "distinct structure" to facilitate comprehension [13]. A follow-up study by Johnson and colleagues identified that mismatches between developers' programming knowledge against information provided by the error message contribute to this confusion [31]. A large-scale multi-method study at Microsoft identified several presentation "pain points," such as "bad warnings messages" and "bad visualization of warnings" [32].

Ko and Myers found that many comprehension difficulties are due to programmers' false assumptions formed while trying to resolve errors [33]. Similarly, Lieber and colleagues postulated that difficulties in resolving errors were due to faulty mental models, or misconceptions, that remained uncorrected until the developer manually requested information explicitly from their programming environment; they developed an always-on visualization in the integrated development to proactively address misconceptions [34].

Still other work has focused on barriers novice developers face. For example, Marceau and colleagues found that students struggle with the "carefully-designed vocabulary of the error message" and often misinterpret the highlighted source code [35]. And a large-scale study of novice compilations found that minor syntax issues and typos are attributable to why compiler error messages are emitted [36].

B. Design Criteria for Compiler Error Messages

The history of design criteria for improving compiler error messages is both long and sometimes sordid; many of these guidelines are today are considered to be pedestrian [37], [3]. Early work by Horner suggested guidelines for the display of error messages, such as the use of headings that identify the version of the compiler being used, a "coordinate system" for relating the error message back to the source code listing, and the "memory addresses" relating to the error message [38]. Shneiderman focused less on the structural design of the error message and more on the holistic presentation, recommending that errors should have a positive tone, be specific using the developer's language, provide actionable information, and have a consistent, comprehensible format [39].

More recently, Traver adapted criteria for Nielson's heuristic evaluation [40] to compiler error messages, suggesting principles such as clarity, specificity, context-insensitivity, as well as previously-seen guidelines such as positive tone and matching the developers' language [1]. Notably, Dean suggested that perhaps one could understand the design of compiler error messages through the psychology of how people communicate to other people, an inspiration for this research [10].

C. Using Stack Overflow for Software Engineering Research

Stack Overflow has been used effectively in both quantitative and qualitative contexts to support software engineering research. Treude and colleagues categorized the kinds of questions that are asked on Stack Overflow, finding that questions about errors are ranked in the top five [41]. Rosen and Shihab used topic modeling to identify questions mobile developers ask. Yang and colleagues also used topic modeling to identify security-related questions [42]. To understand why Java developers struggle with cryptographic APIs, Nadi and colleagues empirically investigated, as a component of their study, Stack Overflow posts to find that APIs are perceived to be too low-level [43]. Particularly influential to our own work is that of Nasehi and colleagues, who used qualitative coding to identify the attributes of good code examples [44]. The results suggested to us that a comparable methodological approach could inform the design of compiler error messages.

Studies on the Stack Overflow community itself provides support of its sufficiency for use as a research domain. For example, Mamykina and colleagues identified both the technical design of the site as well as the visibility and involvement of the design team as contributing to the high quality of answers found within the community [15]. Anderson and colleagues identify a temporal relationship between posts and accepted answers: the first correct answer is usually the one that is accepted, motivating both quick and correct responses to Stack Overflow questions [45]. Consequently, Bacchelli and colleagues, recognizing the benefits of Stack Overflow answers for developers, introduced an Eclipse plugin to integrate Stack Overflow knowledge directly within the IDE [46].

IX. Discussion

In this section, we discuss two directions from our findings. First, we discuss the design space of argumentation layout in terms and its implications for macrostructure and microstructure presentation. Second, we discuss recent trends in compiler architecture that error message diagnostics can potentially leverage to provide better argumentation to developers.

Design space of argumentation layout. As we have shown in our results, error messages can be examined in terms of argumentation theory in terms of both macrostructure and microstructure. Importantly, our results show that compile error messages in questions from Stack Overflow frequently do not even minimally meet the simple argument structure—instead, only providing the claim itself, with no supporting evidence (Figure 3). Although microstructure plays an important role in supporting argumentation, we could consider how likely microstructure improvements are in benefiting developers when the underlying macrostructure is inadequately supported.

Consequently, argumentation theory allows toolsmiths to be intentional about the presentation of error messages. Rather than relying on intuition or other ad-hoc approaches for how to best improve an error message, toolsmiths can first ask which argumentation layout is most appropriate for the type of diagnostic they wish to convey to a developer. For example, for simple syntax errors, automatic resolutions such as Quick Fixes may be entirely sufficient to aid the developer. In contrast, resolutions for which there is not a single obvious resolution, or for which the resolution is not immediately self-evident, toolsmiths may opt to progressively disclose argumentation structure. That is, they might first choose to

present only the claim: then, should the developer remain confused, they can first request from the compiler a simple argument layout. If the simple argument is insufficient, the developer may be request an extended argument layout from the compiler. Research prototypes like FIXBUGS recognize the need to support a spectrum of resolution types, from simple one-shot resolutions to those that require evaluating multiple potential resolutions [47].

Roadmap for automation. Compilers are increasingly shifting from the traditional, opaque, black box architecture—one in which source codes comes in and machine code comes out the other—to an introspective architecture that allows editors and tools to surface the sophisticated analysis performed within these compilers. For example, recent versions of Visual Studio provide developer tooling, such as support for "smart" refactoring and code navigation, by having the integrated development environment closely coordinate with the compiler [48]. This type of compiler introspection might also be retrofitted to support error message diagnostics. For example, when additional warrants are needed to support an argument, the IDE might introspectively query the compiler to obtain additional evidence (or tools) that could help the developer understand the compiler error message.

A second route for automation in compiler technology is error message interactivity. Although error messages present information to the developer, the developer has no means to communicate their own understanding of the diagnostic back to the compiler. Essentially, the developer has no way to provide or request a rebuttal from the compiler in the event of disagreement. Given that rebuttal is a component of argumentation theory, toolsmiths may want to consider ways in which developers can negotiate with the compilers. For example, if the compiler indicates that a variable may be `null`, one way to interact with the compiler could be to allow the developer to insert an `assert(x != null)` statement in the code to indicate disagreement with the claim. Then, the compiler, having information on the source of disagreement, can present further information to support its claim.

X. CONCLUSIONS

In this study, we applied argumentation theory to the domain of compiler error messages, by framing error messages as arguments intended to convince the developer of a particular problem within their source code. To understand how such arguments should be structured, we examined question-answer pairs from Stack Overflow—in which a question from a questioner contains the compiler error message—coupled with an answer accepted by that questioner as the alternative, human-authored error message.

Our findings suggest that arguments as presented by compiler error messages are significantly different than the arguments that developers accept as satisfactory. Specifically, we found that, in contrast with human-authored accepted answers, compiler error messages do not provide adequate argumentation layout in terms of macrostructure (Section VI-A). Indeed, many compiler error messages present only the claim, without any supporting argument components, to the developer. Furthermore, a detailed qualitative analysis identified orthogonal, yet effective, argument layouts. For certain compiler error messages, a claim-with-resolution layout is sufficient to help the developer. In other situations, more elaborate argument layouts are used by authors to convince developers for why the compiler error is actually an error (Section VI-B).

Finally, we investigated the microstructure of arguments to identify how accepted answers support their arguments at the statement level, when adequate macrostructure support is available. We found that arguments use code examples, language specifications, and even bug reports as backing to support their arguments (Section VI-C).

The results of our work provide empirical evidence for the utility of argument layout when presenting compiler error messages to developers. Although argument may seem adversarial, it is precisely through argument that we come to agreement—and through agreement, resolution.

REFERENCES

[1] V. J. Traver, "On compiler error messages: What they say and what they mean," *Advances in Human-Computer Interaction*, vol. 2010, pp. 1–26, 2010.

[2] M. Wand, "Finding the source of type errors," in *POPL*, Jan. 1986, pp. 38–43.

[3] P. J. Brown, "Error messages: the neglected area of the man/machine interface," *Communications of the ACM*, vol. 26, no. 4, pp. 246–249, Apr. 1983.

[4] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, May 2014, pp. 252–261.

[5] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in C++," in *First Workshop on C++ Template Programming*. Germany, 2000.

[6] "Ensuring completeness of switch statements." [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSRTLW_9.1.0/org.eclipse.jdt.doc.user/tasks/task-ensuring_switch_completeness.htm

[7] L. Augustsson, "Compiling pattern matching." Springer, Berlin, Heidelberg, 1985, pp. 368–381.

[8] N. Pinkwart, K. Ashley, C. Lynch, and V. Aleven, "Evaluating an intelligent tutoring system for making legal arguments with hypotheticals," *Int. J. Artif. Intell. Ed.*, vol. 19, no. 4, pp. 401–424, Dec. 2009.

[9] F. H. van Eemeren, B. Garssen, E. C. W. Krabbe, A. F. Snoeck Henkemans, B. Verheij, and J. H. M. Wagemans, *Handbook of Argumentation Theory*. Dordrecht: Springer Netherlands, 2014.

[10] M. Dean, "How a computer should talk to people," *IBM Systems Journal*, vol. 21, no. 4, pp. 424–453, 1982.

[11] S. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003.

[12] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" in *ACM SIGCSE Bulletin*, Feb. 2008, pp. 168–172.

[13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE*, May 2013, pp. 672–681.

[14] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?" in *ICSE*, 2011, pp. 804–807.

[15] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *CHI*, May 2011, pp. 2857–2866.

[16] C. Hardaker, "Trolling in asynchronous computer-mediated communication: From user discussions to academic definitions," *Journal of Politeness Research. Language, Behaviour, Culture*, vol. 6, no. 2, pp. 215–242, Jan. 2010.

[17] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding In-depth Semistructured Interviews," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, aug 2013.

[18] S. L. Simpson, R. G. Lyday, S. Hayasaka, A. P. Marsh, and P. J. Laurienti, "A permutation testing framework to compare groups of brain networks," *Frontiers in Computational Neuroscience*, vol. 7, p. 171, 2013.

[19] J. A. Maxwell, "Using numbers in qualitative research," *Qualitative Inquiry*, vol. 16, no. 6, pp. 475–482, Jul. 2010.

[20] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[21] M. Birks, Y. Chapman, and K. Francis, "Memoing in qualitative research: Probing data and processes," *Journal of Research in Nursing*, vol. 13, no. 1, pp. 68–75, jan 2008.

[22] J. Ponterotto, "Brief note on the origins, evolution, and meaning of the qualitative research concept thick description," *The Qualitative Report*, vol. 11, no. 3, 2006.

[23] "Compiler errors for humans." [Online]. Available: http://elm-lang.org/blog/compiler-errors-for-humans

[24] W. Landi and William, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, Dec. 1992.

[25] D. Ford, J. Smith, P. J. Guo, and C. Parnin, "Paradise unplugged: Identifying barriers for female participation on Stack Overflow," in *FSE*, 2016, pp. 846–857.

[26] R. Slag, M. de Waard, and A. Bacchelli, "One-day flies on stackoverflow-why the vast majority of stackoverflow users only posts once," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 458–461.

[27] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, "How developers visualize compiler messages: A foundational approach to notification construction," in *VISSOFT*, 2014.

[28] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen, "Catching bugs in the web of program invariants," in *PLDI*, vol. 31, no. 5, May 1996, pp. 23–32.

[29] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *PASTE*, nov 2008, p. 57.

[30] D. F. Polit and C. T. Beck, "Generalization in quantitative and qualitative research: Myths and strategies," *International Journal of Nursing Studies*, vol. 47, no. 11, pp. 1451–1458, 2010.

[31] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *FSE*, 2016, pp. 73–84.

[32] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *ASE*, 2016, pp. 332–343.

[33] A. Ko and B. Myers, "Development and evaluation of a model of programming errors," in *IEEE Symposium on Human Centric Computing Languages and Environments*. IEEE, 2003, pp. 7–14.

[34] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *CHI*, Apr. 2014, pp. 2481–2490.

[35] G. Marceau, K. Fisler, and S. Krishnamurthi, "Mind your language: On novices' interactions with error messages," in *ONWARD*, Oct. 2011, pp. 3–17.

[36] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *SIGCSE*, Feb. 2015, pp. 522–527.

[37] P. G. Moulton and M. E. Muller, "DITRAN—a compiler emphasizing diagnostics," *Communications of the ACM*, vol. 10, no. 1, pp. 45–52, Jan. 1967.

[38] J. J. Horning, "What the compiler should tell the user," in *Compiler Construction: An Advanced Course*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1974, vol. 21, pp. 525–548.

[39] B. Shneiderman, "Designing computer system messages," *Communications of the ACM*, vol. 25, no. 9, pp. 610–611, Sep. 1982.

[40] J. Nielsen, "Heuristic evaluation," *Usability inspection methods*, vol. 17, no. 1, pp. 25–62, 1994.

[41] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?" in *ICSE*, 2011, p. 804.

[42] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, "What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 910–924, Sep. 2016.

[43] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *ICSE*, 2016, pp. 935–946.

[44] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 25–34.

[45] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec, "Discovering value from community activity on focused question answering sites," in *KDD*, 2012, pp. 850–858.

[46] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, Jun. 2012, pp. 26–30.

[47] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration," in *International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[48] S. Mukherjee, *Source Code Analytics With Roslyn and JavaScript Data Visualization*. Berkeley, CA: Apress, 2016.