An Empirical Study on the Relationship Between Code Verifiability and Code Understandability

Anonymous Author(s)

ABSTRACT

Proponents of software verification have argued that simpler code is easier to verify: that is, that verification tools issue fewer false positives, and require less human intervention, when analyzing simpler code. This paper empirically validates this assumption by comparing the number of false positives produced by four state-of-the-art verification tools with 20 measures of code understandability collected from humans in six prior studies. We combined correlation results on all 20 of these measures using statistical meta-analysis methods imported from medicine, which to our knowledge have not previously been used to combine results from multiple human studies in software engineering.

Our experiments show that, in aggregate, there is a small-to-medium correlation between verifiability and understandability ($r = 0.27$). This supports the claim that easy-to-verify code is often easier to understand than code that requires more effort to verify. Our work has implications for the users and designers of verification tools and for future attempts to automatically measure the comprehensibility of code: using verification tools may have ancillary benefits to understandability, and measuring understandability may require reasoning about semantic, not just syntactic, properties of code.

ACM Reference Format:
Anonymous Author(s). 2023. An Empirical Study on the Relationship Between Code Verifiability and Code Understandability. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programmers must deeply understand source code in order to implement new features, fix bugs, refactor, review code, and do other essential software engineering activities [4, 53, 66, 86]. However, understanding code is challenging and time-consuming for developers: studies [56, 98] have estimated developers spend 58%–70% of their time understanding code.

Complexity is a major reason why code can be hard to understand [3, 5, 6, 66, 74]: algorithms may be written in convoluted ways or be composed of numerous interacting code structures and dependencies. There are two major sources of complexity in code: *essential complexity*, which is needed for the code to work, and *accidental complexity*, which could be removed while retaining the code's semantics [5, 13]. Whether the complexity is essential or accidental, understanding complex code demands high cognitive effort from developers [3, 74].

Researchers have proposed many metrics to approximate code complexity [3, 18, 21, 37, 39, 61, 82, 100] using vocabulary size (*e.g.*, Halstead's complexity [34]), program execution paths (*e.g.*, McCabe's cyclomatic complexity [55]), program data flow (*e.g.*, Beyer's DepDegree [9]), *etc.*. These syntactic metrics are intended to alert developers about complex code so they can refactor or simplify it to remove accidental complexity [4, 32, 66], or to predict developers' cognitive load when understanding code [59, 66, 74]. However, recent studies have found that (some of) these metrics (*e.g.*, McCabe's) either weakly or do not correlate at all with code understandability as perceived by developers or measured by their behavior and brain activity [26, 66, 74]. Other studies have demonstrated that certain code structures (*e.g.*, if vs for loops, flat vs nested constructs, or repetitive code patterns) lead to higher or lower understanding effort (*a.k.a. code understandability* or *comprehensibility*) [3, 12, 26, 39, 41, 49], which diverges from the simplistic way metrics (*e.g.*, McCabe's) measure code complexity [3, 26, 39, 43, 74].

In this paper, we investigate the relationship between complexity and *code verifiability*—how easy or hard it is for an automated verification tool to prove safety properties about the code, such as the absence of null pointer violations or out-of-bounds array accesses. Our research is motivated by the common assumption within the software verification community that *simpler code is both easier to verify by automated verification tools and easier to understand by developers.* For instance, the Checker Framework [65] user manual states this assumption explicitly in its advice to handle an unexpected warning: "rewrite your code to be simpler for the checker to analyze; this is likely to make it easier for people to understand, too" [87]. The documentation of the OpenJML verification tool states [89]: "success in checking the consistency of the specifications and the code will depend on... the complexity and style in which the code and specifications are written" [64]. This assumption is widely held by verification experts, but it has never been validated empirically.

The intuition behind this assumption is that a verifier can handle a certain amount of complexity before it issues a false positive warning. If it is possible to remove the false positive warning, then the complexity that caused it must be accidental rather than essential, and therefore removing the false positive reduces the overall complexity of the code. For example, consider accessing a possibly-null pointer in a Java-like language. A simple null check might use an if statement. A more complex variant with the same semantics might dereference the pointer within a try statement and use a catch statement to intercept the resulting exception if the pointer is null. The second, more convoluted variant (with its significant accidental complexity) might not be verified—a null pointer dereference does occur, but it is intercepted before it crashes the program.

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A verifier would need to model exceptional control flow to avoid a false positive warning.

The goal of this paper is to empirically validate the purported relationship between verifiability and code comprehensibility—and therefore either confirm or refute the common assumption that easy-to-understand code is easy to verify (and vice-versa). To do so, we need a proxy for verifiability. Verifiers analyze source code to prove the absence of particular classes of defects (*e.g.*, null dereferences) using *sound* analyses. A sound verifier can find all defects (of a well-defined class) in the code. However, most interesting properties of programs are undecidable [70], so all sound verifiers produce false positive warnings: that is, they conservatively issue a warning when they cannot produce a proof. The user of the verifier must sort the true positive warnings that correspond to real bugs from the false positive warnings that occur due to the verifier's imprecision. These false positive warnings are a good proxy for verifiability because the fewer false positives encountered in a given piece of code, the less work a developer using the verifier will need to do to verify that code.

With that in mind, *we* ***hypothesize*** *that a correlation exists between a code snippet's comprehensibility, as judged by humans, and its verifiability, as measured by false positive warnings.*

We conducted an empirical study to validate this hypothesis—the first time that this common assumption in the verification community has been tested empirically. Our study compares the number of false positive warnings produced by three state-of-the-art, sound static code verifiers [57, 65, 89] and one industrial-strength, unsound static analysis tool based on a sound core [15] with ≈19.6k measurements of proxies for code understandability collected from humans in six prior studies [12, 14, 66, 68, 74, 79] for 211 Java code snippets. Such measurements come from 20 metrics that fall into four categories [59]: (1) human-judged *ratings*, (2) program output *correctness*, (3) comprehension *time*, and (4) *physiological* (*i.e.*, brain activity) metrics. We used a statistical meta-analysis technique to examine the correlation between verifiability and these understability metrics in aggregate; our methodology is inspired by meta-analyses in the medical field that combine the results of studies of the same or similar drugs on diverse populations. Without importing this statistical analysis technique, we would have lacked a statistically-valid method to combine the correlation results on each of the 20 metrics into a single summary correlation. Given the small sample sizes of the original studies and the danger of multiple comparisons, our meta-analysis technique permits us to draw methodologically-sound conclusions about the overall trend that would otherwise be fraught or impossible.

We found a small-to-medium correlation between the proxies for understandability and verifiability, in aggregate ($r = 0.27$); individually, 13 of 20 individual metrics were correlated with verifiability. This trend suggests that more often than not, code that is easier to verify is easier for humans to understand. A key implication of this result is that, when using a verification tool, developers *should* make changes to the code to make it easier to verify automatically: doing so is *more likely than not* to make the code easier for a human to understand, because any complexity removed is likely accidental. This means that verification tools provide a secondary benefit beyond their guarantees of the absence of errors: code that can be easily verified will be easier for future developers to improve and extend. Another key implication is that our results provide evidence for a relationship between the *semantics* of a piece of code and its understandability, which helps to explain the apparent ineffectiveness of prior *syntactic* approaches. A final implication is that the verifiability of a code snippet, as measured automatically by the false positives issued by extant verification tools, might be a useful input to models of code understandability.

In summary, the main contributions of this paper are:

- a meta-analysis methodology for combining the results of multiple human studies, imported from medicine but novel in software engineering;
- empirical evidence of the correlation between understandability and verifiability derived from applying that methodology, which supports the common assumption that code that is easier to verify is easier for humans to understand (and vice-versa). Our results have key implications for the design and deployment of verification tools in practice and for automated metrics of code comprehensibility; and
- an online replication package that enables verification and replication of our results [?] and enables future research on the topic. The package includes code snippets, human-based comprehensibility measurements, verification tools, scripts to process tool output and produce the study results, the raw study results, and documentation for replication.

2 EMPIRICAL STUDY DESIGN

The goal of this empirical study is to assess the correlation between human-based code comprehensibility metrics and code verifiability—*i.e.*, how many false positive warnings static code verification tools issue. Intuitively, our goal is to check if code that is easy to verify is also easy for humans to understand. To that end, we formulate three research questions (RQs):

RQ1 How do individual human-based code comprehensibility metrics correlate with tool-based code verifiability?
RQ2 How do human-based code comprehensibility metrics correlate with tool-based code verifiability in aggregate?
RQ3 Do the answers to **RQ1** and **RQ2** differ by verifier?
RQ4 Do different kinds of comprehensibility metrics correlate better or worse with tool-based verifiability?

RQ1 and **RQ2** encode our *hypothesis*: that a correlation exists between tool-based code verifiability and human-based code comprehensibility. **RQ1** asks whether specific metrics correlate with verifiability. However, due to the limitations of prior studies, sample sizes for the metrics considered individually are quite small. So, **RQ2** asks whether there is a pattern to the answers to **RQ1** that summarizes the overall direction. We answer **RQ2** statistically by combining the results of the individual metrics targeted by **RQ1**.

RQ3 asks whether code verifiability as measured using a single verification tool correlates with comprehensibility. Intuitively, we aim to determine 1) if there are correlation differences across tools in the correlations measured for **RQ1** and **RQ2** and 2) whether any particular tool dominates the results. **RQ4** asks whether there is any difference in correlation between code verifiability and different proxies for code comprehensibility. Based on prior work [59], we focus on four types of human-based comprehensibility metrics, namely *correctness*, *rating*, *time*, and *physiological* metrics. Together, the answers to **RQ3** and **RQ4** help us explain our results for **RQ1** and **RQ2**: they show which tool(s) and which metric(s) are responsible for the correlations we observe.

Our overall methodology was: first, we compiled a set of human-based code comprehensibility measurements from prior studies (section 2.1). Then, we executed four verification-based tools on the same code snippets to measure how often each snippet cannot be verified (sections 2.2 and 2.3). Finally, we correlated the comprehensibility metrics with the number of warnings produced by the tools and analyzed the correlation results using a meta-analysis methodology inspired by medicine (section 2.4). We did a correlation study rather than try to establish causation because that would require expensive controlled experiments with human subjects. Since correlation cannot exist without causation, it is practical to re-use existing studies and establish correlation first before attempting an expensive causation study, which we leave as future work.

2.1 Code and Understandability Dataset

We used existing datasets (DSs) from six prior understandability studies [12, 14, 66, 68, 74, 79], which are summarized in table 1. Each study used a different set of code snippets and proxy metrics to measure understandability using different groups of human subjects who performed specific understandability tasks. In total, we used ≈19.6k understandability measurements (see the "Meas." column in table 1) for 211 Java code snippets, collected from 364 human subjects using 20 metrics. We selected these studies because their snippets are written in Java (required by our verifiers; see section 2.2). A prior meta-study [59] contained the first five datasets, with compilable snippets—required for tool execution (section 2.3). To identify the datasets and facilitate the replication of our work, we use the same nomenclature as that study (DS1, DS2, DS3, DS6, and DS9). The last dataset (Dataset F or DSF) comes from a more recent study [66]. Since the purpose of the studies was to measure understandability, we assume that the code snippets are correct—*i.e.*, have no bugs—which allows us to assume that all warnings issued by the verification tools are false positives.

The snippets are 211 programs (5 to 75 non-blank/comments LOC or NCLOC—17 NCLOC on avg.) with different complexity levels [12, 14, 66, 68, 74, 79]. Datasets 3, 6, and 9 derive from open source software projects (OSS)—*e.g.*, Hibernate, JFreeChart, Antlr, *etc.* [12, 14, 74]—and the remaining datasets provide implementations of algorithms taught in 1st-year programming courses (*e.g.*, reversing an array or finding a substring) [66, 68, 79]. The original studies selected short code snippets to control for potential cofounding factors that may affect understandability [66, 68, 79].

We selected the understandability metrics used in the meta-study conducted by Muñoz *et al.* [59]—see table 1 for the metrics, their type, and a brief description of them (our replication package has full descriptions [?]). We also used Muñoz *et al.*'s categorization of the metrics. *Correctness* (**C** metrics in table 1) metrics measure the correctness of the program output given by the participants. *Time* (**T**) metrics measure the time that participants took to read, understand, or complete a snippet. *Rating* (**R**) metrics indicate the subjective rating given by the participants about their understanding of the code snippet or code readability, using Likert scales. *Physiological* (**P**) metrics measure the concentration level of the participants during program understanding, via deactivation measurements of brain areas (*e.g.*, Brodmann Area 31 or BA31 [79]).

Study participants were mostly CS undergraduate/graduate students with intermediate-to-high programming experience, as reported in the original papers [12, 14, 66, 68, 74, 79]. Only DS6's study included professional developers [74]—see table 1.

2.2 Verification Tools

We used the following criteria to select verification tools:

(1) Each tool must be based on a sound core—*i.e.*, the underlying technique must generate a proof.
(2) Each tool must be actively maintained.
(3) Each tool must fail to verify at least one snippet.
(4) Each tool must run mostly automatically.
(5) Each tool must target Java.

Criterion 1 requires that each tool be verification-based. Our hypothesis implies that the *process of verification* can expose code complexity: that is, our purpose in running verifiers is not to expose bugs in the code but to observe how and when the tools produce false positive warnings (due to code complexity). Therefore, each tool must perform verification under the hood (*i.e.*, must attempt to construct a proof) for our results to be meaningful. This criterion excludes non-verification static analysis tools such as FindBugs [7] which use unsound heuristics. Exploring whether those tools correlate with comprehensibility is future work. However, criterion 1 does *not* require the tool to be sound: merely that it be based on a sound core. We permit *soundiness* [52] because practical verification tools commonly only make guarantees about the absence of defects under certain conditions, and also intentionally-unsound tools based on a sound core.

Criteria 2 through 5 are practical concerns. Criterion 2 requires the verifier to be state-of-the-art so that our results are useful to the community. Criterion 3 requires each verifier to issue at least one false positive warning—for tools that verify a property that is irrelevant to the snippets (and so cannot issue warnings), we cannot do a correlation analysis. Criterion 4 excludes proof assistants and other tools that require extensive manual effort. Criterion 5 restricts the scope of the study: we focused on Java code and verifiers. We made this choice because (1) a significant portion of prior code comprehensibility studies using human subjects used Java—*e.g.*, 5/10 studies considered by Muñoz *et al.* [59] are on Java programs; no other language has more than 2 studies—and (2) Java has received significant attention from the program verification community due to its prevalence in practice. We discuss the threats to validity that this and other choices cause in section 5.

We are also interested in *variety* among the verifiers, although none of our criteria capture it explicitly. Ideally, we would select tools based on many different verification paradigms, because we want to answer our **RQs** about verification in general. Another motivation for variety among the verifiers is that tools built on different infrastructures might issue false positives due to different *sources* of complexity: their models of programs could be too conservative in different ways. Despite our desire for variety, for practical reasons, we restricted ourselves to abstract-interpretation-like static analyses [20] (as a broadly-defined category of verifier, *e.g.*, as described

Table 1: Datasets (DSs) of code snippets and understandability measurements/metrics used in our study. The metrics types are "C" for correctness, "R" for ratings, "T" for time, and "P" for physiological.

DS	Snippets	NCLOC	Participants	Understandability Task	Understandability Metrics	Meas.
1 [79]	23 CS algorithms	6 - 20	41 students	Determine prog. output	C: *correct_output_rating* (3-level correctness score for program output) R: *output_difficulty* (5-level difficulty score for determining program output) T: *time_to_give_output* (seconds to read program and answer a question)	2,829
2 [68]	12 CS algorithms	7 - 15	16 students	Determine prog. output	P: *brain_deact_31ant* (deactivation of brain area BA31ant) P: *brain_deact_31post* (deactivation of brain area BA31post) P: *brain_deact_32* (deactivation of brain area BA32) T: *time_to_understand* (seconds to understand program within 60 secs.)	228
3 [14]	100 OSS methods	5 - 13	121 students	Rate prog. readability	R: *readability_level* (5-level score for readability/ease to understand)	12,100
6 [74]	50 OSS methods	18 - 75	50 students and 13 developers	Rate underst./answer Qs	R: *binary_understandability* (0/1 program understandability score) C: *correct_verif_questions* (% of correct answers to verification questions) T: *time_to_understand* (seconds to understand program)	1,197
9 [12]	10 OSS methods	10 - 34	104 students	Rate read./complete prog.	C: *gap_accuracy* (0/1 accuracy score for filling in program blanks) R: *readability_level_ba* (5-level avg. score for readability b/a code completion) R: *readability_level_before* (5-level score for readability before code completion) T: *time_to_read_complete* (avg. seconds to rate readability and complete code)	2,600
F [66]	16 CS algorithms	7 - 19	19 students	Determine prog. output	P: *brain_deact_31*(deactivation of brain area BA31) P: *brain_deact_32* (deactivation of brain area BA32) R: *complexity_level* (score for program complexity) C: *perc_correct_output* (% of subjects who correctly gave program output) T: *time_to_understand* (seconds to understand program within 60 seconds)	631

in [24], versus model-checking or bounded model-checking). Future work will investigate other verification paradigms.

2.2.1 *Selected Verification Tools.*

By applying the criteria defined above, we selected four verification tools: Infer [15], the Checker Framework [65], JaTyC [57], and OpenJML [89].

Infer [15] is an unsound, industrial static analysis tool based on a sound core of separation logic [62] and bi-abduction [16]. Separation logic enables reasoning about mutations to program state independently, making it scalable; bi-abduction is an inference procedure that automates separation logic reasoning. Infer is unsound by design: despite internally using a sound, separation-logic-based analysis, it uses heuristics to prune all but the most likely bugs from its output, because it is tailored for deployment in industrial settings. We used Infer version 1.1.0.

The **Checker Framework** [65] is a collection of pluggable typecheckers [28], which enhance a host language's type system to track an additional code property, such as whether each object might be null. The Checker Framework includes many pluggable typecheckers. We used the nine that satisfy criterion 4, which prevent programming mistakes related to: nullness [23, 65], interning [23, 65], object construction [45], resource leaks [46], array bounds [44], signature strings [23], format strings [95], regular expressions [83], and optionals [88]. We used Checker Framework version 3.21.3.

The **Java Typestate Checker (JaTyC)** [57] is a typestate analysis [85]. A typestate analysis extends a type system to also track *states*—for example, a typestate system might track that a File is first closed, then open, then eventually closed. Currently-maintained typestate-based Java static analysis tools include JaTyC [57] (a typestate verifier) and RAPID [25] (an unsound static analysis tool based on a sound core that permits false negatives when verification is expensive). We chose to use JaTyC rather than RAPID for two reasons. First, JaTyC ships with specifications for general programming mistakes, but RAPID focuses on mistakes arising from mis-uses of cloud APIs; the snippets in our study do not interact with cloud APIs. Second, JaTyC is open-source, but RAPID is closed-source. We used JaTyC commit b438683.

OpenJML [89] converts verification conditions to SMT formulae and dispatches those formulae to an external satisfiability solver. OpenJML verifies specifications expressed in the Java Modeling Language (JML) [51]; it is the latest in a series of tools verifying JML specifications by reduction to SMT going back to ESC/Java [27]. We used OpenJML version 0.17.0-alpha-15 with the default solver z3 [22] version 4.3.1.

2.2.2 *Verification Tools Considered but Not Used.*

We considered and discarded three other verifiers: JayHorn [42], which fails criterion 2 [75]; CogniCrypt [48], which fails criterion 3; and Java PathFinder [36], which fails criterion 4.

2.3 Snippet Preparation and Tool Execution

We acquired compilable code snippets from prior work [59, 66]. We made some modifications to prepare the snippets for tool execution. These changes did *not* alter the program semantics (and so, the underlying code complexity and comprehensibility were not altered either). DS3 included 4 commented-out snippets. We uncommented these snippets and created stubs for the classes, method calls, *etc.* they use without modifying any of the snippets' internal semantics. We added code comments at the beginning and end of snippets to differentiate snippet and non-snippet code, and thus, link individual tool warnings to the right snippets (based on code line numbers). We wrote scripts to execute the verifiers on the snippets. The scripts were prepared so that all verification failures for each tool were displayed in each script run. For each tool, we redirect the warning output to a text file for parsing via heuristics—see our replication package for the list of heuristics defined for each tool [?]. Our analysis of warnings for each snippet indicates a fairly uniform distribution of warning types on the datasets. Our replication package provides these distributions per dataset.

2.4 Correlation and Analysis Methods

2.4.1 Aggregation. We aggregated the comprehensibility measurements and the number of tool warnings for each code snippet in the datasets. The resulting pairs of comprehensibility and verifiability values per snippet can be correlated for sets of snippets.

Specifically, we averaged the individual code comprehensibility measurements per snippet for each metric. For example, for each snippet in DS1 we averaged the 41 *time_to_give_output* measurements collected from the 41 participants in the corresponding study [79]. Following Muñoz *et al.* [59], we averaged discrete measurements, which mostly come from Likert scale responses in the original studies. For example, the metric *output_difficulty* (from DS1) is the perceived difficulty in determining program output using a 0-4 discrete scale. While there is no clear indication of whether Likert scales represent ordinal or continuous intervals [58], we observed that the Likert items in the original datasets represent discrete values on continuous scales [59], so it is reasonable to average these values to obtain one measurement per snippet.

Regarding code verifiability, we summed up the number of warnings from the verification tools for each snippet. We considered averaging rather than summing up. However, since the correlation coefficient that we used (see below) is robust to data scaling (*i.e.*, the average is essentially a scaled sum), imbalances in the number of warnings from each tool do not change the correlation results. Further, we performed an ablation experiment to investigate possible effects of warning imbalances on correlation (**??**).

2.4.2 Statistical methods. We used Kendall's $\boldsymbol{\tau}$ [47] to correlate the individual comprehensibility measurements and the tool warnings because [19] (1) it does not assume the data to be normally distributed and have a linear relationship, (2) it is robust to outliers, and (3) it has been used in prior comprehensibility studies [59, 66, 74]. As in previous studies [66, 74], we follow Cohen's guidelines [19] and interpret the correlation strength as *none* when $0 \leq |\tau| < 0.1$, *small* when $0.1 \leq |\tau| < 0.3$, *medium* when $0.3 \leq |\tau| < 0.5$, and *large* when $0.5 \leq |\tau|$ [19].

To answer **RQ1**, we first stated the expected correlation (as either *positive* or *negative*) between each comprehensibility metric and code verifiability that would support our hypothesis. For some metrics, such as *correct_output_rating* in DS1, a *negative* correlation indicates support for the hypothesis—if humans can deduce the correct output *more* often, the hypothesis predicts a *lower* number of warnings from the verifiers. A *positive* expected correlation, such as for *time_to_understand* in DS6, indicates that higher values in that metric support the hypothesis—*e.g.*, if humans take *longer* to understand a snippet, our hypothesis predicts that *more* warnings will be issued on that snippet. We computed the correlation (and its strength) between the comprehensibility metrics and code verifiability and compared the observed correlations with the expected ones to check if the results validate or refute our hypothesis.

To answer **RQ2**, we performed a statistical meta-analysis [11] of the correlation results. A meta-analysis is appropriate for answering **RQ2** because it combines individual correlation results that come from different metrics as a single aggregated correlation result [11, 59]. In disciplines like medicine, a meta-analysis is used to combine the results of independent scientific studies on closely-related research questions (*e.g.*, establishing the effect of a treatment for a disease), where each study reports quantitative results (*e.g.*, a measured effect size of the treatment) with some degree of error [11]. The meta-analysis statistically derives an estimate of the unknown common truth (*e.g.*, the true effect size of the treatment), accounting for the errors of the individual studies. Typically, a meta-analysis follows the random-effects model to account for variations in study designs (*e.g.*, different human populations) [11]. Intuitively, a random-effects-based meta-analysis estimates the true effect size as the weighted average of the effect sizes of the individual studies [11], where the weights are estimated via statistical methods (*e.g.*, Sidik and Jonkman's [77]).

In our case, the correlation analysis *for each metric* represents an individual study that seeks to establish the correlation between code comprehensibility (measured by that metric) and verifiability. Since the comprehensibility measurements come from different studies with different designs (*i.e.*, with different goals, comprehensibility interpretations and metrics, code snippets, human subjects, *etc.*), a random-effects meta-analysis is appropriate.

To perform the random-effects meta-analyses, we followed a standard procedure for data preparation and analysis [11]. First, we transformed Kendall's τ values into Pearson's $\boldsymbol{r}$ values [94]. Then, we transformed the r values to be approximately normally distributed, using Fisher's scaling. Next, we normalized the signs of the individual metric r correlations so that a negative correlation (the choice of negative is arbitrary: choosing positive leads to the same results with the opposite sign) supports our hypothesis: we multiplied by -1 the correlation value for metrics where a positive correlation would support the hypothesis. This strategy is also used in other disciplines when combining different metrics whose signs have opposite interpretations, *e.g.*, in [92]. We executed the meta-analysis on the resulting normalized data points using Sidik and Jonkman's estimator [77] since the heterogeneity of the individual studies may be large [35]. We used R's *meta* package [76] to run the meta-analyses and used forest plots [35] to visualize the Pearson's r values, their estimated confidence intervals, the estimated weights for the aggregated correlation, and additional meta-analysis results (*e.g.*, p-values and variance).

While we provide the p-values of all of these statistical analyses, we emphasize that especially the **RQ1** results for individual metrics should be interpreted with caution given the relatively small sample sizes and that fact that 20 metrics are considered. For example, DS2 only contains 12 snippets, which means only 12 data points were used for correlation for its metrics. We used a meta-analysis because interpreting the individual metric results is dangerous and can be misleading. Our meta-analysis also obviates the need for statistical correction to avoid multiple comparisons, such as Holm-Bonferroni's [38]: the meta-analysis aggregates all of the results and informs us of the overall trend. We use the same interpretation guidelines for Pearson's r values that we used for Kendall's τ (*small* when $0.1 \leq |r| < 0.3$, *etc.*) [19, 66, 74].

To answer **RQ3**, we applied the same methodology as for **RQ2** but only considering the number of warnings produced by each individual tool (*i.e.*, no aggregation was used). We also performed a "leave one tool out" ablation experiment to check if any single tool was dominating the overall meta-analysis results..

To answer **RQ4**, we repeated the same methodology used for **RQ2** for only the metrics in each metric category: *time*, *correctness*,

Table 2: Correlation results based on Kendall's τ (K.'s τ) for each dataset (DS) and Metric. A metric falls into one Type: Correctness, Time, Rating, & Physiological. The expected correlation direction (+/-), if our hypothesis is correct, is either positive (+) or negative (-). We assess τ's direction/strength, compared to the expected correlation (RQ1?): '-' means *no* correlation, 'y' means expected and measured correlations match (thus supporting our hypothesis), and 'n' means they do not match. Capital letters (Y or N) mean a *medium* or higher correlation, otherwise, a *small* correlation. τ's significance is tested at the $p < 0.05$ (*) & $p < 0.01$ levels (**).

DS	Metric	Type	+/-	K.'s τ	RQ1?
1	*correct_output_rating*	C	-	-0.34*	Y
	output_difficulty	R	-	-0.43**	Y
	time_to_give_output	T	+	0.41**	Y
2	*brain_deact_31ant*	P	-	-0.31	Y
	brain_deact_31post	P	-	-0.45	Y
	brain_deact_32	P	-	-0.38	Y
	time_to_understand	T	+	0.14	y
3	*readability_level*	R	-	-0.17*	y
6	*binary_understand*	R	-	0.00	-
	correct_verif_questions	C	-	0.00	-
	time_to_understand	T	+	0.05	-
9	*gap_accuracy*	C	-	-0.45	Y
	readability_level_ba	R	-	0.12	n
	readability_level_before	R	-	0.17	n
	time_to_read_complete	T	+	-0.36	N
F	*brain_deact_31*	P	-	-0.18	y
	brain_deact_32	P	-	-0.18	y
	complexity_level	R	+	0.35	Y
	perc_correct_out	C	-	-0.16	y
	time_to_understand	T	+	-0.13	n

rating, and *physiological*—*i.e.*, we performed four meta-analyses, one for each metric group.

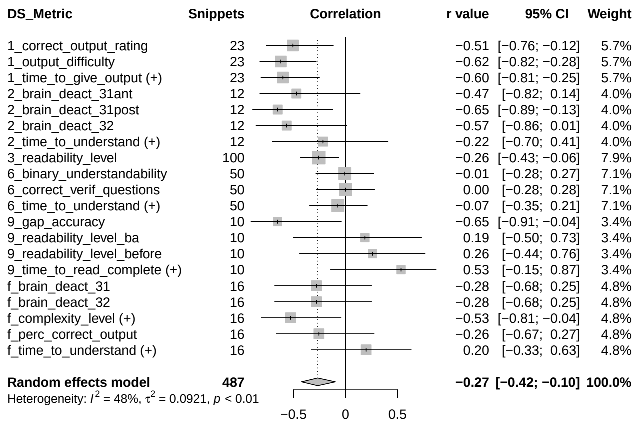
3 STUDY RESULTS AND DISCUSSION

The scripts and data we used to generate the results in this section are available in our replication package [?].

3.1 RQ1: Individual Correlation Results

Table 2 summarizes the results of each metric's correlation (Kendall's τ) with the total number of warnings across all tools. Table 2 reveals that for 13 of the 20 (65%) metrics, the direction of the correlation supports our hypothesis of a positive answer to **RQ1**. For 3 metrics, there is *no* correlation, and for the last 4 metrics, the correlation is in the opposite direction than expected. Table 2 indicates the strength of the correlation in the rightmost column. Of the metrics where we found a *medium* or higher correlation, 8/9 are in the direction that supports our hypothesis. For the other 5 metrics that support our hypothesis and the other 3 that do not, their correlation is *small.*

With regard to metric categories, 3/4 correctness metrics, 3/6 rating metrics, 2/5 time metrics, and 5/5 physiological metrics correlate with verifiability. Interestingly, all 4 metrics that anti-correlate



DS_Metric	Snippets	r value	95% CI	Weight
1_correct_output_rating	23	−0.51	[−0.76; −0.12]	5.7%
1_output_difficulty	23	−0.62	[−0.82; −0.28]	5.7%
1_time_to_give_output (+)	23	−0.60	[−0.81; −0.25]	5.7%
2_brain_deact_31ant	12	−0.47	[−0.82; 0.14]	4.0%
2_brain_deact_31post	12	−0.65	[−0.89; −0.13]	4.0%
2_brain_deact_32	12	−0.57	[−0.86; 0.01]	4.0%
2_time_to_understand (+)	12	−0.22	[−0.70; 0.41]	4.0%
3_readability_level	100	−0.26	[−0.43; −0.06]	7.9%
6_binary_understandability	50	−0.01	[−0.28; 0.27]	7.1%
6_correct_verif_questions	50	0.00	[−0.28; 0.28]	7.1%
6_time_to_understand (+)	50	−0.07	[−0.35; 0.21]	7.1%
9_gap_accuracy	10	−0.65	[−0.91; −0.04]	3.4%
9_readability_level_ba	10	0.19	[−0.50; 0.73]	3.4%
9_readability_level_before	10	0.26	[−0.44; 0.76]	3.4%
9_time_to_read_complete (+)	10	0.53	[−0.15; 0.87]	3.4%
f_brain_deact_31	16	−0.28	[−0.68; 0.25]	4.8%
f_brain_deact_32	16	−0.28	[−0.68; 0.25]	4.8%
f_complexity_level (+)	16	−0.53	[−0.81; −0.04]	4.8%
f_perc_correct_output	16	−0.26	[−0.67; 0.27]	4.8%
f_time_to_understand (+)	16	0.20	[−0.33; 0.63]	4.8%
Random effects model	**487**	**−0.27**	**[−0.42; −0.10]**	**100.0%**

Figure 1: Results of the random-effects meta-analysis of the metrics in table 2. Note that the correlation r values of the metrics with positive expected direction (+) are negated.

with verifiability are concentrated in the rating and time categories. These two metric categories are the two most subjective: ratings are opinions, and some time metrics require the subject to signal the experimenter when they are complete. We further investigate the differences between metric categories in section 3.4.

3.2 RQ2: Aggregate Correlation Results

Interpreting table 2 directly, however, is difficult: what does 13/20 metrics agreeing with our hypothesis mean, exactly? To answer this question, we performed a meta-analysis (fig. 1). This plot displays the observed correlation (Person's ***r* value**, obtained from the Kendall's τ value as described in section 2.4.2), the 95% confidence interval (**95% CI**), and the estimated weight for each metric (**DS_Metric**). This information is shown numerically and graphically (the **Correlation** chart in fig. 1). Each gray box's size is its estimated weight (larger box size means larger weight), and the box's middle point represents the correlation with respect to the solid vertical line at zero. There is a negative correlation if the box is to the left of the vertical line; positive if it is to the right; all metrics have been normalized so that the expected correlation is negative (that is, a negative correlation supports a positive answer to **RQ2**). The horizontal lines indicate the confidence intervals for each metric. At the bottom, the forest plots show the aggregated correlation (see the **Random Effects Model** row), and related information, calculated by the meta-analysis. The diamond and the dotted vertical line are the aggregated correlation; the width of the diamond represents the confidence interval.

The results of the meta-analysis show that there is a small-to-medium correlation supporting an affirmative answer to **RQ2** ($r = -0.27$, with a 95% CI that contains both some small and some medium correlations: $r = -0.42$ to $r = -0.10$, $p < 0.01$). We interpret these results overall as support for the hypothesis that tool-based verifiability and humans' ability to understand code are correlated. Though this finding is not universally supported by every metric, the meta-analysis gives us confidence that overall, understandability metrics correlate with code verifiability.

The plots in fig. 1 show wide confidence intervals for most metrics, which indicates relatively high variability in their correlations. The metrics from DSes 1, 2, 9, and F have higher variability than those of DS3 and DS6, likely because of the smaller number of datapoints used for correlation for Datasets 1, 2, 9, and F. One key observation from fig. 1 is that all the DS9 metrics have the lowest weights calculated by the meta-analysis. This is likely because of the low number of correlation datapoints in DS9.

3.3 RQ3: Correlation Results By Tool

To answer **RQ3**, we repeated the analyses used to answer **RQ1** and **RQ2** independently for each tool (*i.e.*, no warning aggregation). We also repeated the analysis in a "leave-one-out" ablation experiment. Table 3 shows data about the number of snippets that each tool warned on and the total number of warnings, by dataset. Repeating the meta-analysis (**??**) on only the warnings produced by each tool individually gave similar results to the "all tools" meta-analysis in fig. 1 (plots in the style of fig. 1 are elided in this section for space, but can be found in our replication package):

► **Infer**: $r = -0.18$, with a 95% CI of $[-0.34, -0.00]$, with $p = 0.51$. The results for Infer have low confidence because of the relatively small number of warnings it produces.

► The **Checker Framework**: $r = -0.18$, with a 95% CI of $[-0.37, 0.03]$, with $p = 0.01$. Just the Checker Framework's warnings correlate a little less strongly than the overall results.

► **JaTyC**: $r = -0.11$, with a 95% CI of $[-0.26, 0.04]$, with $p = 0.12$. JaTyC warnings also correlate more weakly when considered alone.

► **OpenJML**: $r = -0.25$, with a 95% CI of $[-0.41, -0.08]$, with $p = 0.06$. The OpenJML results are very similar to the overall results, but with lower confidence.

Our conclusion from these results is that, while each tool supports the hypothesis less strongly than the overall meta-analysis, all the tools show the same pattern. This gives the meta-analysis more confidence in the overall results: that the correlation exists for every studied tool is suggestive that it does, in fact, exist. However, the similarity between the overall results and the results for just OpenJML concerned us, because we worried that OpenJML might be dominating the overall results. We therefore performed an ablation study by repeating the meta-analysis using data from each combination of three tools. The results were the following:

► **No Infer**: $r = -0.27$, with a 95% CI of $[-0.41, -0.11]$, with $p = 0.02$. Infer does not appear to contribute much to the overall results: removing it does not change them much. We believe that this is due to the relatively small number of warnings it produces.

► **No Checker Framework**: $r = -0.24$, with a 95% CI of $[-0.40, -0.08]$, with $p < 0.01$. The Checker Framework is an important contributor to the overall results, but without it we find a similar result.

► **No JaTyC**: $r = -0.28$, with a 95% CI of $[-0.51, -0.01]$, with $p < 0.01$. Removing the warnings from JaTyC significantly expands the 95% CI but otherwise doesn't change the results. We interpret this to mean that JaTyC is an important contributor to the results but that it is mostly in line with the other tools.

► **No OpenJML**: $r = -0.16$, with a 95% CI of $[-0.32, 0.02]$, with $p = 0.03$. OpenJML is definitely an important contributor to our results: without its warnings, the size of the correlation is notably smaller. Even without it, though, there *is* a correlation (even if a somewhat smaller one). So, while OpenJML's warnings are important to our results, the other tools also show a similar pattern.

Taken together, the results in this section show that the correlation we found in **??** is not entirely driven by any tool: the overall results remain similar (if a bit weaker) for each tool individually and for each combination of three tools (*i.e.*,, without each tool). OpenJML, the tool which contributes the largest number of warnings, is (unsurprisingly) responsible for much (but not all) of the observed correlation, and considering just the other three tools still produces a correlated result (though of a smaller magnitude). We interpret these results to mean that this correlation exists across regardless of the specific verifier in use—meaning that our results apply to verification in general.

3.4 RQ4: Correlation Results by Metric Kind

Figure 2 shows forest plots of the random-effects meta-analysis of our correlation results for the four metric types: *correctness*, *physiological time*, and *rating*.

Figures 2a and 2b, reveal a reasonably *negative* correlation for the *correctness* and *physiological* metrics (-0.32 and -0.44 aggregated *r* values, respectively). Compared to rating metrics, correctness and physiological measurements are more objective proxies of understandability, since they do not require a human to give an opinion—their ground truth is derived from a fact about the world rather than what a human believes, making them stronger in the same way that measuring what a developer does is more effective than asking them what they do. We interpret that the strength, consistency, and objectiveness of these results support for our hypothesis that there is a correlation between verifiability and human understandability.

The results for both *time* and *rating* metrics (see figs. 2c and 2d, respectively) are more mixed. *Time* (fig. 2c) is the most varied metric type: the random-effects model shows that there is no discernible aggregate correlation. One possible explanation for this result is that, for all of these metrics, the human must decide when they understand the snippet well enough to signal to the experimenter. That process introduces more variation from individuals as different people might have different levels of "understanding" that they reach before signaling the experimenter. Validating this conjecture is in our plans for future work.

With respect to the *rating* metrics (fig. 2d), we observe that despite the DS9 rating metrics being anti-correlated individually (with respect to our hypothesis), the aggregated correlation from the meta-analysis supports our hypothesis and the weights of the individual metrics from DS9 are significantly lower than for the other rating metrics. This phenomenon likely occurs because the confidence intervals for those metrics are large, which means that their correlations are less reliable.

3.5 Robustness Experiments

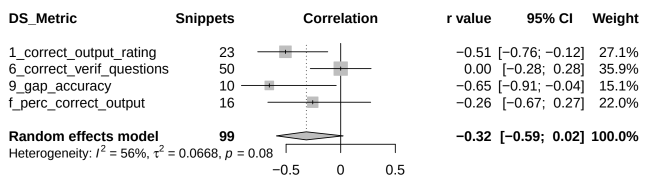
We ran experiments to probe the robustness of the findings for the RQs and mitigate some threats to validity.

3.5.1 *Handling Code Comments in Dataset 9.*

DS9's original study had 3 versions of each of its 10 snippets, with three types of code comments: "good", "bad", and no comments [12]. The results presented elsewhere in this section used the "No comments" (NC) version of DS9, because none of the four verifiers use comments as

Table 3: Number of snippets each tool warns on and the total number of warnings per dataset.

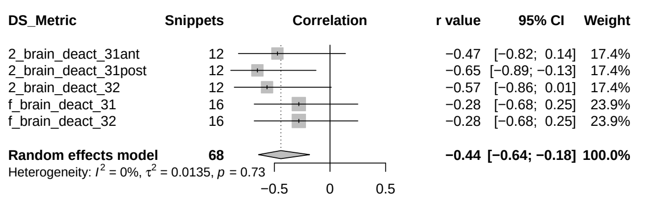
	Snippets Warned On							Total Warnings						
Dataset:	1	2	3	6	9	F	**All**	1	2	3	6	9	F	**All**
Infer	0/23	0/12	13/100	23/50	6/10	1/16	**43/211**	0	0	13	24	6	1	**44**
Checker Fr.	3/23	0/12	19/100	28/50	4/10	3/16	**57/211**	7	0	52	83	4	3	**149**
JaTyC	3/23	1/12	88/100	40/50	10/10	2/16	**144/211**	14	3	327	537	37	6	**924**
OpenJML	14/23	6/12	69/100	41/50	10/10	13/16	**153/211**	29	11	808	219	24	29	**1120**
All Tools	17/23	7/12	94/100	48/50	10/10	15/16	**191/211**	50	14	1200	863	71	39	**2237**



DS_Metric	Snippets	r value	95% CI	Weight
1_correct_output_rating	23	−0.51	[−0.76; −0.12]	27.1%
6_correct_verif_questions	50	0.00	[−0.28; 0.28]	35.9%
9_gap_accuracy	10	−0.65	[−0.91; −0.04]	15.1%
f_perc_correct_output	16	−0.26	[−0.67; 0.27]	22.0%
Random effects model	**99**	**−0.32**	**[−0.59; 0.02]**	**100.0%**

Heterogeneity: $I^2 = 56\%$, $\tau^2 = 0.0668$, $p = 0.08$

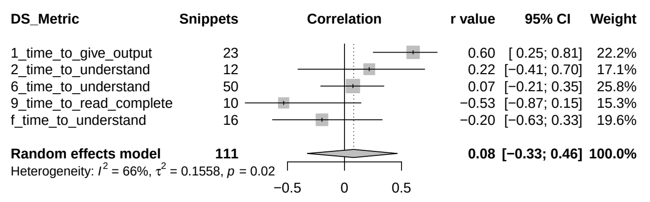
(a) Correctness metrics.



DS_Metric	Snippets	r value	95% CI	Weight
2_brain_deact_31ant	12	−0.47	[−0.82; 0.14]	17.4%
2_brain_deact_31post	12	−0.65	[−0.89; −0.13]	17.4%
2_brain_deact_32	12	−0.57	[−0.86; 0.01]	17.4%
f_brain_deact_31	16	−0.28	[−0.68; 0.25]	23.9%
f_brain_deact_32	16	−0.28	[−0.68; 0.25]	23.9%
Random effects model	**68**	**−0.44**	**[−0.64; −0.18]**	**100.0%**

Heterogeneity: $I^2 = 0\%$, $\tau^2 = 0.0135$, $p = 0.73$

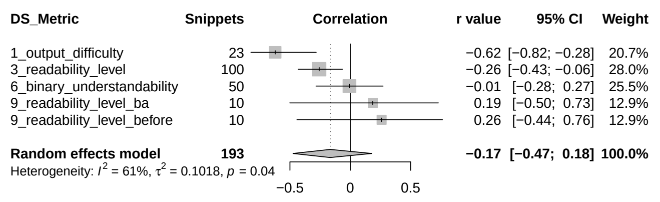
(b) Physiological metrics.



DS_Metric	Snippets	r value	95% CI	Weight
1_time_to_give_output	23	0.60	[0.25; 0.81]	22.2%
2_time_to_understand	12	0.22	[−0.41; 0.70]	17.1%
6_time_to_understand	50	0.07	[−0.21; 0.35]	25.8%
9_time_to_read_complete	10	−0.53	[−0.87; 0.15]	15.3%
f_time_to_understand	16	−0.20	[−0.63; 0.33]	19.6%
Random effects model	**111**	**0.08**	**[−0.33; 0.46]**	**100.0%**

Heterogeneity: $I^2 = 66\%$, $\tau^2 = 0.1558$, $p = 0.02$

(c) Time metrics. Note that the expected correlation is positive.



DS_Metric	Snippets	r value	95% CI	Weight
1_output_difficulty	23	−0.62	[−0.82; −0.28]	20.7%
3_readability_level	100	−0.26	[−0.43; −0.06]	28.0%
6_binary_understandability	50	−0.01	[−0.28; 0.27]	25.5%
9_readability_level_ba	10	0.19	[−0.50; 0.73]	12.9%
9_readability_level_before	10	0.26	[−0.44; 0.76]	12.9%
Random effects model	**193**	**−0.17**	**[−0.47; 0.18]**	**100.0%**

Heterogeneity: $I^2 = 61\%$, $\tau^2 = 0.1018$, $p = 0.04$

(d) Rating metrics.

Figure 2: Overall results of the random-effects meta-analysis for each metric type. The correlation direction that supports our hypothesis for *time* metrics is *positive*, while for the other three metric types, a *negative* correlation supports our hypothesis.

Table 4: Correlation results (Kendall's τ) on different versions of DS9: "No" (NC), "Bad" (BC), and "Good Comments" (GC). A * indicates statistical significance at the $p < 0.05$ level.

Metric	+/-	NC	BC	GC
gap_accuracy	-	-0.45	-0.31	-0.45
readability_level_ba	-	0.12	0.45	-0.02
readability_level_before	-	0.17	0.48	0.10
time_to_read_complete	+	-0.36	-0.36	-0.74*

part of their logic. However, this choice might be source of possible bias, so we analyzed how the correlation results would change if we had used the "Good comments" (GC) or "Bad comments" (BC) versions of the dataset. Note that because none of the verifiers take comments into account, their warnings are exactly the same—the only differences are in the comprehensibility measurements.

Table 4 shows how the correlation results differ for the three versions of DS9. A significant difference is observed in the two readability metrics: when the comments are bad, these metrics are anti-correlated with verifiability: that is, humans rated the snippets on which the tools issued more warnings as *more readable*. Oddly, we see a similar phenomenon for the time metrics, but it occurs only for the good (rather than bad) comments. DS9 is one of the smallest datasets (with only 10 snippets), so one possibility is that these are artifacts. In particular, we do not have an explanation for why readability anti-correlates with verifiability if and only if the comments are bad. That the time metric anti-correlates when comments are good is easier to understand: good comments should make it easier to reason about the code, and thus reduce the need for humans to do semantic reasoning themselves.

Table 5: Correlation results (Kendall's τ) for OpenJML, for each timeout-handling approach: (1) ignore timeouts; (2) under-estimate the warnings hidden by timeouts; (3) over-estimate the warnings hidden by timeouts. τ's significance is tested at the $p < 0.05$ (*) & $p < 0.01$ levels (**).

		Approach		
DS	Metric	1: Ignore	2: Under	3: Over
3	*readability_level*	-0.20**	-0.23**	-0.17*
6	*binary_understand*	-0.07	-0.07	0.00
	correct_verif	-0.06	-0.06	0.00
	time_to_understand	0.11	0.11	0.05

3.5.2 *Handling OpenJML Timeouts.*

OpenJML uses an SMT solver under the hood. Though modern SMT solvers return results quickly for most queries using sophisticated heuristics, some queries do lead to exponential run time, making it necessary to set a timeout when analyzing a collection of snippets. We used a 60 minute timeout, which leads to 2/50 snippets in DS6 and 39/100 snippets in DS3 timing out (and zero in the other datasets). We considered three approaches in our correlation analysis for timeouts: 1) ignore snippets containing timeouts entirely, 2) count each timeout as zero warnings (but do count any other warnings issued in the snippet before timing out), or 3) count each snippet that timed out as if the maximum warning count in the dataset. The data in table 2 uses approach 3. The reason we chose approach 3 over approach 2 is that timeouts typically occur on the most complicated SMT queries—which might hide many warnings. Therefore, approach 2 *underestimates* the warning count that a no-timeout run of OpenJML would encounter, while approach 3 *overestimates* the warning count in a no-timeout run. We re-ran the correlation analysis under all three conditions; the results are in table 5. We did not observe any significant differences between the treatments of timeouts—the overall direction and strength of the correlations are similar.

4 IMPLICATIONS

Our results have implications for both users and builders of verifiers, as well as code comprehensibility researchers.

The primary implication for **users of verification tools** is that there is truth to the common assumption [64, 87] that easier to verify code is also easier for humans to understand. When a verifier issues a false positive on code under analysis, the developer *should* consider refactoring the code to allow the tool to verify it: doing so may result in code that is easier for humans to understand. Further, the correlation between understandability and verifiability suggests that programmers who want to make their code easier for others to understand could adopt verifiers both to detect bugs and to help improve understandability via refactoring to avoid false positives.

Designers of verification tools want to reduce false positives, because they are a key barrier to adopting static analyzers [40, 60]. However, the correlation between false positives and understandability could be useful for tool builders. Suppose that a verifier, when it cannot verify some code, suggested a semantically-equivalent refactoring that *would* be verifiable. If the user of such a tool knows of the correlation between verifiability and understandability, they may be more willing to accept such a suggestion: the verifiable refactoring might also be easier for other humans to understand. As far as we are aware, no verification tools exploit this method to improve the code under analysis. A tool with such refactoring suggestions is more likely to be used in practice [73].

Researchers interested in code comprehension can benefit from our results. The correlation between verifiability and understandability increases our confidence that there is a semantic component to human code understanding. Most prior attempts to design automated metrics for code understandability have used *syntactic* measurements that do not account for the program semantics—that is, what the program *means* rather than what it looks like. Our results suggest that an automated metric computed only from syntactic information may be insufficient to explain whether a human can understand a piece of code, so researchers should focus on metrics that account for the code's semantic properties.

Future work could develop better code comprehensibility metrics by incorporating verifiability into models that predict human-based comprehensibility [14, 74, 90]. If the link between verifiability and comprehensibility exists (as our results suggest), verifiability information could complement the syntactic features of these models (e.g., program construct counts). This information, which encodes program semantics, can be captured by adapting existing verification tools. For example, we could provide how many false positives a tool produces on a snippet as an input feature to these models. Researchers have developed many verification tools, which may help capture different complexity types and could be repurposed to model comprehensibility. Future studies can analyze the complexity types captured by different tools.

5 LIMITATIONS AND THREATS TO VALIDITY

Our study shows a *correlation* between verifiability and comprehensibility, without establishing that one *causes* the other. Our results should therefore be interpreted carefully: though we hypothesize that less complex—hence more comprehensible—code may make verification less likely to fail, future research is required to validate this hypothesized causal link.

There are threats to the external validity of our study: the correlation we found may not generalize beyond the specific conditions of our study. The snippets are all Java code, so the results may not generalize to other languages. We only used a few verifiers: we were limited by parcity of practical tools that can analyze the snippets. While limitations or bugs in individual tools could skew our results, we mitigated this threat by re-running the experiments individually for each tool and with an ablation experiment (**??**), which demonstrated that no single tool dominates the results. The snippets are all relatively small compared to full programs; the comprehensibility of larger programs may differ. Further, 3/6 datasets are composed of snippets from introductory CS courses rather than real-life programs, but this is mitigated by the other 3 datasets of open-source snippets from real programs.

Another threat to external validity is that the data in prior studies was mostly collected from students: only DS6 used professional software engineers (and only 13 professionals, whose results are mixed with 50 students), so our results may not apply to more experienced programmers. Future work should conduct human understandability studies with professional engineers.

Beyond the datasets and tools, there are threats to internal and construct validity. We assumed the snippets do not contain any real bugs and therefore the warnings from the verifiers are all false positives. The presence of a real bug would make a snippet seem "harder to verify" in our correlation analysis (because every verifier would warn about it), even if the snippet is easy for humans to understand, skewing the results. However, because the snippets are sourced from studies of human subjects (who each examined the snippets), it is reasonable to assume they do not contain real bugs.

6 RELATED WORK

Our research is related to work on code complexity metrics, empirical validation of complexity metrics, code understandability, and studies of verification tools.

Code complexity metrics. Researchers have proposed many metrics for code complexity [3, 21, 39, 61, 74, 82, 100], though the concept is not easy to define due to different interpretations [5, 6]. Most metrics rely on simple, syntactic properties such as code size or branching paths [61, 66]. Applications of these metrics include detecting complex code so developers can simplify it during software evolution [4, 32, 66]. The motivation is that complex code is harder to understand [3, 74], which may have important repercussions on developer effort and on software quality (*e.g.*, bugs introduced due to misunderstood code). Our correlation results imply that code that is easier to verify might also be simple because it is easier to understand by humans; we believe the underlying mechanism might be that simple code fits into the expected code patterns of a verification technique. Our results also suggest that a complexity metric that aims to capture human understandability should consider not only syntactic information about the code, but also its semantics.

Empirical validation of complexity metrics. Scalabrino *et al.* [74] collected code understandability measurements from professional developers about open-source code. They correlated their measurements with 121 syntactic complexity metrics (*e.g.*, cyclomatic complexity, LOC, *etc.*) and developer-related properties (*e.g.*, code author's experience and background). The study found only small correlations, and only for a few metrics, though a model trained on combinations of metrics performed better. Similar results were found by Trockman *et al.* [90].

Researchers have explored the limitations of classical complexity metrics [3, 26, 39, 43, 74]. For example, Ajami *et al.* [3] found that different code constructs (*e.g.*, ifs vs. for loops) have different effects on how developers comprehend code, implying that metrics such as cyclomatic complexity, which weights code constructs equally, fail to capture understandability [66]. Recent work has proposed new metrics such as Cognitive Complexity (COG) [17, 72], which assigns different weights to different code constructs. Muñoz *et al.* [59] conducted a correlation meta-analysis between COG and human understandability. They found that time and rating metrics have a modest correlation with COG, while correctness and physiological metrics have no correlation. Since our results in section 3.3 indicate that verifiability has a modest correlation with correctness and physiological metrics, a combination of verifiability and COG is a promising avenue for future work.

Our study extends prior work by providing empirical evidence of the correlation between code verifiability and human-based code understandability. To the best of our knowledge, we are the first to empirically investigate this relationship.

Studying code understandability. Researchers have studied code understandability and the factors that affect it via controlled experiments and other user studies [3, 12, 33, 39, 41, 66, 68, 79]. Since it is difficult to precisely define understandability, some studies have used it interchangeably with readability [12, 14, 59, 63, 69] (a different, yet related concept). Measurements include the time to read, understand, or complete code, the correctness of the output given by the participants, and perceived code complexity, readability or understandability. Physiological measures, collected via fMRI scanners [66, 68, 79], biometrics sensors [29, 31, 99], or eye-tracking devices [1, 10, 29, 91], give a more objective perspective of comprehensibility. Our study utilizes these human-based measurements of understandability to assess their correlation with code verifiability.

Factors that affect understandability and readability include: code constructs [3, 41], code (micro-)patterns [12, 39, 49], identifier quality and style [80, 96], code comments [12], information gathering tasks [10, 50, 79, 80], comprehension tools [84], code reading behavior [2, 67, 78], code authorship [30], high-level comprehension strategies [78], programmer experience [96, 98], and the use of traditional complexity metrics [97]. Our work investigates a new factor that may affect understandability: code verifiability. Our results suggest there is a correlation between these variables, yet future studies are needed to assess causality.

Studies of verification and static analysis tools. The most closely related work is a small study conducted to evaluate a code readability model [14]: the readability model was found to correlate moderately with snippets on which FindBugs [7] issued warnings. Unlike the tools used in our study, FindBugs is *not* a verification tool: it uses heuristics to identify possibly-buggy code. Further, we correlated verifiability with metrics of human understanding; the earlier study correlated FindBugs warnings with an automated readability *model* trained using human judgments.

Though verification and static analysis tools are becoming more common in industry [8, 71], studies of their use and the challenges developers face in deploying them [8, 54, 60, 81, 93] suggest false positives remain a problem in practice [40, 60]. While researchers have proposed prioritization strategies to surface warnings that are more likely to be real defects to developers [73, 93], recent evidence suggests that they may not have much of an impact when deployed [54]. Our work gives a new perspective the problem of false positives. We have shown that the presence of false positives from verifiers correlates with more difficult-to-understand code. We hope that this perspective will encourage developers to view false positives from verifiers as opportunities to improve their code rather than as something to merely find defects [73].

7 CONCLUSIONS AND FUTURE WORK

Our empirical study on the correlation between tool-based verifiability and human-based metrics of code understanding suggests there *is* a connection between whether a tool can verify a code snippet and how easy it is for a human to understand. Verifiability is a promising alternative to traditional code complexity metrics, and future work could combine measures of tool-based verifiability with modern complexity metrics like cognitive complexity that seem to capture different aspects of human understanding into a unified, automatic model. Our results are also promising support for the prospect of increased adoption of verifiers: our results offer a new perspective on the classic problem of false positives, since they suggest that false positives from verifiers are opportunities to make code more understandable by humans.

REFERENCES

[1] Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating Developers' Cognitive Load at a Fine-grained Level Using Eye-tracking Measures. In *Intl. Conf. on Prog. Compr. (ICPC)*. 111–121.

[2] Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. 2019. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *Intl. Conf. on Soft. Eng. (ICSE)*. 384–395.

[3] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2019. Syntax, predicates, idioms — what really affects code complexity? *Emp. Soft. Eng.* 24, 1 (2019), 287–328.

[4] Erik Ammerlaan, Wim Veninga, and Andy Zaidman. 2015. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*. 504–507.

[5] Vard Antinyan. 2020. Evaluating Essential and Accidental Code Complexity Triggers by Practitioners' Perception. *IEEE Soft.* 37, 6 (2020), 86–93.

[6] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Emp. Soft. Eng.* 22, 6 (2017), 3057–3087.

[7] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE Software* 25, 5 (2008), 22–29.

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*, Vol. 1. 470–481.

[9] Dirk Beyer and Ashgan Fararooy. 2010. A Simple and Effective Measure for Complex Low-Level Dependencies. In *Intl. Conf. on Prog. Compr. (ICPC)*. 80–83.

[10] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Emp. Soft. Eng.* 18, 2 (2013), 219–276.

[11] Michael Borenstein, Larry V. Hedges, Julian P. T. Higgins, and Hannah R. Rothstein. 2009. *Introduction to Meta-Analysis.* John Wiley & Sons.

[12] Jürgen Börstler and Barbara Paech. 2016. The role of method chains and comments in software readability and comprehension—An experiment. *Trans. on Soft. Eng. (TSE)* 42, 9 (2016), 886–898.

[13] Frederick Brooks and H Kugler. 1987. *No silver bullet.* April.

[14] Raymond Buse and Westley Weimer. 2009. Learning a metric for code readability. *Trans. on Soft. Eng. (TSE)* 36, 4 (2009), 546–558.

[15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symp.* Springer, 3–11.

[16] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages (POPL)*. 289–300.

[17] G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation. In *Intl. Conf. on Technical Debt.* 57–58.

[18] S.R. Chidamber and C.F. Kemerer. 1994. A metrics suite for object oriented design. *Trans. on Soft. Eng. (TSE)* 20, 6 (1994), 476–493.

[19] Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. 2002. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences* (3 ed.). Routledge.

[20] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symp. on the Principles of Programming Languages (POPL)*. Los Angeles, CA, 238–252.

[21] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *Trans. on Soft. Eng. (TSE)* SE-5, 2 (1979), 96–104.

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, 337–340.

[23] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *Intl. Conf. on Soft. Eng. (ICSE)*. Waikiki, Hawaii, USA, 681–690.

[24] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *Trans. on Computer-Aided Design of Integ. Circ. and Sys.* 27, 7 (2008), 1165–1178.

[25] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. RAPID: checking API usage for the cloud in the cloud. In *European Soft. Eng. Conf. and Symp. on the Found. of Soft. Eng. (ESEC/FSE)*. 1416–1426.

[26] Janet Feigenspan, Sven Apel, Jorg Liebig, and Christian Kastner. 2011. Exploring Software Measures to Assess Program Comprehension. In *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM)*. 127–136.

[27] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*. 234–245.

[28] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *Conf. on Programming Language Design and Implementation (PLDI)*. Atlanta, GA, USA, 192–203.

[29] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *Intl. Conf. on Soft. Eng. (ICSE)*. 402–413.

[30] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. 2010. A degree-of-knowledge model to capture source code familiarity. In *Intl. Conf. on Soft. Eng. (ICSE)*. 385–394.

[31] Davide Fucci, Daniela Girardi, Nicole Novielli, Luigi Quaranta, and Filippo Lanubile. 2019. A Replication Study on Code Comprehension and Expertise using Lightweight Biometric Sensors. In *Intl. Conf. on Prog. Compr. (ICPC)*. 311–322.

[32] Javier García-Munoz, Marisol García-Valls, and Julio Escribano-Barreno. 2016. Improved Metrics Handling in SonarQube for Software Quality Monitoring. In *Intl. Conf. on Distributed Comp. and Art. Intel.* 463–470.

[33] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. Thinking aloud about confusing code: a qualitative investigation of program comprehension and atoms of confusion. In *European Soft. Eng. Conf. and Symp. on the Found. of Soft. Eng. (ESEC/FSE)*. 605–616.

[34] Maurice H. Halstead. 1977. *Elements of Soft. Science.* Elsevier.

[35] Mathias Harrer, Pim Cuijpers, Toshi A. Furukawa, and David D. Ebert. 2021. *Doing Meta-Analysis with R: A Hands-On Guide.* Chapman and Hall/CRC.

[36] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *Intl. Jour. on Soft. Tools for Technology Transfer* 2, 4 (2000), 366–381.

[37] Brian Henderson-Sellers. 1995. *Object-oriented metrics: measures of complexity.* Prentice-Hall, Inc.

[38] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[39] Ahmad Jbara and Dror G. Feitelson. 2017. How programmers read regular code: a controlled experiment using eye tracking. *Emp. Soft. Eng.* 22, 3 (2017), 1440–1477.

[40] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Intl. Conf. on Soft. Eng. (ICSE)*. 672–681.

[41] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *Intl. Conf. on Soft. Maint. and Evol. (ICSME)*. 513–523.

[42] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A framework for verifying Java programs. In *Intl. Conf. on Computer Aided Verification (CAV)*. Springer, 352–358.

[43] Cem Kaner, Senior Member, and Walter P. Bond. 2004. Software Engineering Metrics: What Do They Measure and How Do We Know?. In *Intl. Soft. Metrics Symp. (METRICS)*.

[44] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. 2018. Lightweight verification of array indexing. In *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*. 3–14.

[45] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying Object Construction. In *Intl. Conf. on Soft. Eng. (ICSE)*. 1447–1458.

[46] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and modular resource leak verification. In *European Soft. Eng. Conf. and Symp. on the Found. of Soft. Eng. (ESEC/FSE)*.

[47] Maurice G. Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[48] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In *European Conf. on Object-Oriented Programming (ECOOP)*. Amsterdam, Netherlands, 10:1–10:27.

[49] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. In *Intl. Conf. on Prog. Compr. (ICPC)*. 25–35.

[50] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program comprehension as fact finding. In *European Soft. Eng. Conf. and the Symp. on on the Found. of Soft. Eng. (ESEC/FSE)*. 361–370.

[51] Gary T Leavens, Albert L Baker, and Clyde Ruby. 1998. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA 1998)*. Citeseer, 404–420.

[52] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[53] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *Trans. on Soft. Eng. and Methodology (TSEM)* 23, 4 (2014), 1–37.

[54] Niloofar Mansoor, Tukaram Muske, Alexander Serebrenik, and Bonita Sharif. 2022. An Empirical Assessment of Repositioning of Static Analysis Alarms. In *Intl. Working Conf. on Source Code Analysis & Manipulation*.

[55] T.J. McCabe. 1976. A Complexity Measure. *Trans. on Soft. Eng. (TSE)* SE-2, 4 (1976), 308–320.

[56] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *Intl. Conf. on Prog. Compr. (ICPC)*. 25–35.

[57] João Mota, Marco Giunti, and António Ravara. 2021. Java typestate checker. In *Intl. Conf. on Coord. Lang. and Models*. Springer, 121–133.

[58] Jacqueline Murray. 2013. Likert data: what to use, parametric or non-parametric? *Intl. Jour. of Business and Social Science* 4, 11 (2013).

[59] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. 2020. An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability. In *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM)*. 1–12.

[60] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*. 532–543.

[61] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Jour. of Sys. and Soft.* 128 (2017), 164–197.

[62] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Intl. Workshop on Computer Science Logic*. Springer, 1–19.

[63] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *Intl. Conf. on Soft. Maint. and Evol. (ICSME)*. 348–359.

[64] OpenJML Developers. 2022. OpenJML - formal methods tool for Java and the Java Modeling Language (JML). https://www.openjml.org/documentation/introduction.html.

[65] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*. Seattle, WA, USA, 201–212.

[66] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fMRI study. In *Intl. Conf. on Soft. Eng. (ICSE)*. 524–536.

[67] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Intl. Conf. on Prog. Compr. (ICPC)*. 342–353.

[68] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2018. A look into programmers' heads. *Trans. on Soft. Eng. (TSE)* 46, 4 (2018), 442–462.

[69] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. 2020. How does code readability change during software evolution? *Emp. Soft. Eng.* 25, 6 (2020), 5374–5412.

[70] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. of the American Mathematical Society* 74, 2 (1953), 358–366.

[71] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A comparison of bug finding tools for Java. In *Intl. Symp. on Soft. Reliab. Eng.* 245–256.

[72] Rubén Saborido, Javier Ferrer, Francisco Chicano, and Enrique Alba. 2022. Automatizing Software Cognitive Complexity Reduction. *IEEE Access* 10 (2022), 11642–11656.

[73] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Intl. Conf. on Soft. Eng. (ICSE)*, Vol. 1. 598–608.

[74] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *Trans. on Soft. Eng. (TSE)* 47, 3 (2019), 595–613.

[75] Martin Schäf and Philipp Rümmer. 2022. personal communication.

[76] Guido Schwarzer. 2022. meta: General Package for Meta-Analysis. https://CRAN.R-project.org/package=meta

[77] Kurex Sidik and Jeffrey N. Jonkman. 2005. Simple heterogeneity variance estimation for meta-analysis. *Jour. of the Royal Statistical Society: Series C (Applied Statistics)* 54, 2 (2005), 367–384.

[78] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*, Vol. 5. 13–20.

[79] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Intl. Conf. on Soft. Eng. (ICSE)*. 378–389.

[80] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In *European Soft. Eng. Conf. and Symp. on Found. of Soft. Eng. (ESEC/FSE'17)*. 140–150.

[81] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *Symp. on Usable Privacy and Security (SOUPS)*. 221–238.

[82] Harry M. Sneed. 1995. Understanding software through numbers: A metric based approach to program comprehension. *Jour. of Soft. Maint.: Research and Practice* 7, 6 (1995), 405–419.

[83] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*. Beijing, China, 20–26.

[84] M. A. D. Storey, K. Wong, and H. A. Müller. 2000. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming* 36, 2 (2000), 183–207.

[85] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Sngineering* SE-12, 1 (Jan. 1986), 157–171.

[86] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? an exploratory study in industry. In *Symp. on the Found. of Soft. Eng. (FSE)*. 1–11.

[87] The Checker Framework Developers. 2022. 2.4.5 What to do if a checker issues a warning about your code. https://checkerframework.org/manual/#handling-warnings.

[88] The Checker Framework Developers. 2022. Optional Checker for possibly-present data. https://tinyurl.com/3surnw4a.

[89] The OpenJML Developers. 2022. OpenJML. https://www.openjml.org/.

[90] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. 2018. "Automatically assessing code understandability" reanalyzed: combined metrics matter. In *Intl. Conf. on Mining Soft. Repositories (MSR)*. 314–318.

[91] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An eye-tracking study assessing the comprehension of c++ and Python source code. In *Symp. on Eye Tracking Research and Applications*. 231–234.

[92] Anne M. van Valkengoed and Linda Steg. 2019. Meta-analyses of factors motivating climate change adaptation behaviour. *Nature Climate Change* (2019), 158–163.

[93] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Emp. Soft. Eng.* 25, 2 (2020), 1419–1457.

[94] David Walker. 2003. JMASM9: Converting Kendall's Tau For Correlational Or Meta-Analytic Analyses. *Jour. of M. A. Stat. Meth.* 2, 2 (2003).

[95] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D. Ernst. 2014. A type system for format strings. In *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*. 127–137.

[96] Eliane S. Wiese, Anna N. Rafferty, and Armando Fox. 2019. Linking Code Readability, Structure, and Comprehension Among Novices: It's Complicated. In *Intl. Conf. on Soft. Eng. (ICSE)*. 84–94.

[97] Marvin Wyrich, Andreas Preikschat, Daniel Graziotin, and Stefan Wagner. 2021. The Mind Is a Powerful Place: How Showing Code Comprehensibility Metrics Influences Code Understanding. In *Intl. Conf. on Soft. Eng. (ICSE)*. 512–523.

[98] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *Trans. on Soft. Eng. (TSE)* 44, 10 (2018), 951–976.

[99] Martin K.-C. Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. 2017. Detecting and comparing brain activity in short program comprehension using EEG. In *Frontiers in Education Conf. (FIE)*. 1–5.

[100] H. Zuse. 1993. Criteria for program comprehension derived from software complexity metrics. In *Workshop on Prog. Compr.* 8–16.