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 attempts to verify this assumption empirically by correlating the number of false positives produced by four state-of-the-art verification tools with measures of code comprehensibility collected from humans in six prior studies on a corpus of 211 program snippets, totaling 3,872 lines of non-comment, non-blank source code.**

We have found support for the hypothesis that code verifiability (as measured by the number of false positives produced by the verifiers) correlates with human-judged code comprehensibility. This suggests that the assumption that easy-to-verify code is often easier to understand than code that requires more effort to verify is true. Our work has implications for the users and designers of verification tools and for future attempts to automatically measure the comprehensibility of code.

I. Introduction

Implementing new features, fixing bugs, refactoring, code reviewing, and other essential software engineering activities require a deep understanding of source code [1–4]. However, understanding code is challenging and time-consuming for developers: studies [5, 6] have estimated developers spend 58%–70% of their time understanding code. Complexity is a major reason why code can be hard to understand [2, 7–10]: algorithms may be written in convoluted ways or be composed of numerous interacting code structures and dependencies. Understanding complex code may demand high cognitive effort from developers while simple code may demand lower cognitive effort [7, 8].

Researchers have proposed many metrics to predict code complexity [7, 11–17] using syntactic proxies that measure code properties such as vocabulary size (*e.g.*, Halstead's complexity [18]), program execution paths (*e.g.*, McCabe's cyclomatic complexity [19]), or program data flow (*e.g.*, Beyer's DepDegree [20]). These metrics are intended to alert developers about complex code so they can refactor or simplify it [2, 3, 21], or to predict how developers perceive code complexity and their cognitive load when understanding code [2, 8, 22]. 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 [2, 8, 23]. 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*) [7, 15, 23–26], which diverges from the simplistic way metrics (*e.g.*, McCabe's) measure code complexity [7, 8, 15, 23, 27].

Given these findings, determining what makes source code simple or complex—and hence easier or harder to understand—remains an open problem. In this paper, we take a step forward in addressing this problem and empirically investigate how automated code verifiability relates to code complexity and understandability. We define *code verifiability* as how easy or hard it is for an automated verification tool to prove general 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 easier to verify by automated verification tools, and consequently, easier to understand by developers*. For instance, the Checker Framework [28] 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" [29]. The documentation of the OpenJML verification tool states [30]: "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" [31]. This assumption is widely held by verification experts, but has never been validated empirically.

The intution behind this assumption is that verifiers are designed and tuned by humans to handle *common, expected code patterns*—the verifiers themselves encode what their designers believe is simple in the rules they use to reason about code. When a verifier cannot prove that there is not a bug in a piece of code, it could be that the code is complex in a way that the verifier cannot understand. 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 use some other code structure, such as dereferencing the pointer within a `try` statement, and using a `catch` statement to intercept the resulting exception if the pointer is `null`. A verifier trying to prove the absence of crashes resulting from null pointer dereferences can almost certainly certify that the first, simpler variant is correct. But the second, more convoluted variant might not be verified—a null pointer dereference does occur, but it is intercepted before it crashes the program. The verifier would need to model exceptional control flow to avoid a false positive warning.

The goal of this paper is to empirically validate this 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 (due to Rice's Theorem [32]), so all sound verifiers produce false positive warnings: that is, they imprecisely model what the program might do and conservatively issue a warning when they cannot produce a proof. A developer using such a verifier then 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 less/more false positives encountered in a given piece of code, the less/more work a developer using the verifier will need to do to verify their code.

With that in mind, *we* ***hypothesize*** *that there is a correlation between the comprehensibility of a snippet of code, as judged by humans, and tool-based verifiability, as measured by false positive warnings*.

In this paper, we report the results of an empirical study that we conducted to validate this hypothesis—the first time that this common assumption in the verification community has been validated empirically. Our study correlated the number of false positive warnings produced by four state-of-the-art, sound static code verifiers [28, 30, 33, 34] and ≈19.6k human-based measurements of code understandability, collected in six prior studies [2, 8, 26, 35–37] for 211 Java code snippets. Such measurements come from 20 metrics that fall into four categories [22]: (1) human-judged *ratings*, (2) program output *correctness*, (3) comprehension *time*, and (4) *physiological* (*i.e.*, brain activity) metrics. Based on statistical correlation and meta-analyses on the collected data, we assessed the overall and per-tool correlation, considering metrics individually and grouped into the four aforementioned categories.

We found evidence that the ease of verifying a particular piece of code is correlated with 13 of the 20 considered understandability measurements of that code, which suggest that more often than not, code that is easier to verify is considered easier to understand by humans. 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, and doing so is *more likely than not* to make the code easier for a human to understand. This means that verification tools provide a secondary benefit beyond their guarantees of the absence of errors: code that can be easily verified would be easier for future developers to improve and extend. Another key implication is that there is a likely relationship between the *semantics* of a piece of code and its understandability, so *syntactic* measures of complexity are unlikely to *ever* successfully model how well humans understand a piece of code.

In summary, the main contributions of this paper are:

- empirical evidence of the correlation between understandability and verifiability, confirming the common assumption that code that is easier to verify is also easier for humans to understand. 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 [38] 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.

II. Empirical Study Design

The goal of our 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 does tool-based code verifiability correlate with human-based code comprehensibility?

RQ2: How does code verifiability for specific verifiers correlate with human-based code comprehensibility?

RQ3: How does tool-based code verifiability correlate with different kinds of code comprehensibility metrics?

RQ1 encodes the *hypothesis* that we seek to validate: that a correlation exists between tool-based code verifiability and human-based code comprehensibility. For this RQ, we define code verifiability via a group of verifiers *in aggregate*. **RQ2** and **RQ3** are refinements of **RQ1** that probe what contributes to our answer to **RQ1**.

RQ2 asks whether code verifiability as measured using a single verification tool correlates with comprehensibility. Intuitively, we aim to determine if there are correlation differences across tools and with the aggregate correlation measured for **RQ1**. **RQ3** asks whether there is any difference in correlation between code verifiability and different proxies for code comprehensibility. Based on prior work [22], we focus on four types of human-based comprehensibility metrics, namely *correctness*, *rating*, *time*, and *physiological* metrics. Together, the answers to **RQ2** and **RQ3** help us explain our results for **RQ1**: they show which tool(s) and which metric(s) are responsible for the correlations we observe.

To answer the **RQ**s, we compiled a set of human-based code comprehensibility measurements for a set of code snippets from prior studies (section II-A). Then, we executed four verification tools on those snippets to measure how often each snippet cannot be verified (sections II-B and II-C). Finally, we correlated the code comprehensibility metrics with the number of warnings produced by the tools and analyzed the correlation results (section II-D).

A. Code Datasets and Understandability Measurements

We used existing datasets (DSs) from six prior understandability studies [2, 8, 26, 35–37]—see table I. Each study used a different set of code snippets and proxy metrics to measure understandability using different groups of human

TABLE I: Datasets (DSs) of code snippets and understandability measurements/metrics used in our study.

DS	Snippets	NCLOC	Participants	Understandability Task	Understandability Metrics	Meas.
1 [35]	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 [36]	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) P: *time_to_understand* (seconds to understand program within 60 secs.)	228
3 [37]	100 OSS methods	5 - 13	121 students	Rate prog. readability	R: *readability_level* (5-level score for readability/ease to understand)	12,100
6 [8]	50 OSS methods	18 - 75	63 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 [26]	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 [2]	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

subjects (computer science students or professional developers), who performed specific understandability tasks. In total, we used ≈19.6k understandability measurements (see the "Meas." column in table I) 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 II-B). A prior meta-study [22] contained the first five datasets, with compilable snippets—required for tool execution (section II-C). To identify the datasets and facilitate the replication of our work, we use the same nomenclature as that study (*i.e.*, Dataset 1 or DS1, *etc.*). The sixth dataset (Dataset F or DSF) comes from a more recent study [2]. 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 [2, 8, 26, 35–37]. Datasets 3, 6, and 9 derive from open source software projects (OSS)—*e.g.*, Hibernate, JFreeChart, Antlr, *etc.* [8, 26, 37]—and the remaining datasets provide implementations of algorithms taught in 1st-year programming courses (*e.g.*, reversing an array or finding a substring) [2, 35, 36]. The original studies selected short code snippets on purpose, to control for potential cofounding factors that may affect understandability [2, 35, 36].

We focused on the understandability metrics used in the meta-study conducted by Muñoz *et al.* [22]—see table I for the metrics, their type, and a brief description of them (our replication package has full descriptions [38]). We also used Muñoz *et al.*'s categorization of the metrics. *Correctness* metrics (**C** metrics in table I) measure the correctness of the program output given by the participants. *Time* metrics (**T** in table I) measure the time that participants took to read, understand, or complete a snippet. *Rating* metrics (**R** in table I) indicate the rate given by the participants about their understanding of the code snippet or code readability, using Likert scales. *Physiological* metrics (**P** in table I) measure the concentration level of the participants during program understanding, via deactivation measurements of certain brain areas (*e.g.*, Brodmann Area 31 or BA31 [35])

B. 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 the tools fail (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 [39] which use manually-curated 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* [40] 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.* [22] 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 V.

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 [41] (as a broadly-defined category of verifier, *e.g.*, as described in [42], versus model-checking or bounded model-checking). Future work will investigate other verification paradigms.

1) Selected Verification Tools: By applying the criteria defined above, we selected four verification tools: Infer [33], the Checker Framework [28], JaTyC [34], and OpenJML [30].

Infer [33] is an unsound, industrial static analysis tool based on a sound core of separation logic [43] and bi-abduction [44]. 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** [28] is a collection of pluggable typecheckers [45], which enhance a host language's type system to track an additional code property. For example, a pluggable typechecker might extend the core Java typechecker to track whether each object might be null. The Checker Framework includes many pluggable typecheckers. We used those that satisfy criterion 4, which prevent programming mistakes related to: nullness [28, 46], interning [28, 46], object construction [47], resource leaks [48], array bounds [49], signature strings [46], format strings [50], regular expressions [51], and optionals [52]. We used Checker Framework version 3.21.3.

The **Java Typestate Checker (JaTyC)** [34] is a typestate analysis [53]. 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 [34] (a typestate verifier) and RAPID [54] (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 [30] 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) [55]; it is the latest in a series of tools verifying JML specifications by reduction to SMT going back to ESC/Java [56]. We used OpenJML version 0.17.0-alpha-15 with the default solver z3 [57] version 4.3.1.

2) Verification Tools Considered but Not Used: We considered and discarded three other verifiers: JayHorn [58], which fails criterion 2 [59]; CogniCrypt [60], which fails criterion 3; and Java PathFinder [61], which fails criterion 4.

C. Snippet Preparation and Tool Execution

We acquired compilable code snippets from prior work [2, 22]. 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 [38]. 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.

D. Correlation and Analysis Methods

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 [35]. Following Muñoz *et al.* [22], 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 [62], we observed that the Likert items in the original datasets represent discrete values on continuous scales [22], 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 (section III-D1).

We used Kendall's τ [63] to correlate the comprehensibility measurements and the number of tool warnings because [64] (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 [2, 8, 22]. As in previous studies [2, 8], we 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 $0.5 \leq |\tau|$ [64].

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 (*i.e.*, comprehensibility is correlated with verifiability). 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.

For answering **RQ2**, we applied the same methodology as for **RQ1** but only considering the number of warnings produced by each individual tool (*i.e.*, no aggregation was used).

For answering **RQ3**, we performed a statistical meta-analysis [65] of the correlation results obtained for the metrics in each metric category: *time*, *correctness*, *rating*, and *physiological*—*i.e.*, we performed four meta-analyses, one for each metric group. A meta-analysis is appropriate for answering **RQ3** because it combines individual correlation results that come from different metrics (in the same metric category) as a single aggregated correlation result [22, 65].

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 [65]. 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) [65]. Intuitively, a random-effects-based meta-analysis estimates the true effect size as the weighted average of the effect sizes of the individual studies [65], where the weights are estimated via statistical methods (*e.g.*, Sidik and Jonkman's [66]).

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. We performed a meta-analysis for each metric category, rather than considering a single group with all the metrics, because metrics across groups are fundamentally different in their definition and numeric interpretation (*e.g.*, time to understand is fundamentally different than ratings).

To perform the random-effects meta-analyses, we followed a standard procedure for data preparation and analysis [65]. First, we transformed Kendall's τ values into Person's r values [67]. Then, we transformed the r values to be approximately normally distributed, using Fisher's scaling. We executed the meta-analysis on the resulting data points using Sidik and Jonkman's estimator [66] since the heterogeneity of the individual studies may be large [68]. We used R's *meta* package [69] to run the meta-analyses and used forest plots [68] 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 the distinct statistical analyses, we emphasize that they should be interpreted with caution given the relatively small set of data points that we used for correlation. For example, DS2 only contains 12 snippets, which means only 12 data points were used for correlation.

III. Study Results and Discussion

The scripts and data we used to generate the results in this section are available in our replication package [38].

A. RQ1: Correlating Comprehensibility and Verifiability

Table II summarizes the results of our correlation analysis when aggregating the warnings from the verifiers. We computed each metric's correlation (Kendall's τ) with the number of warnings. Table II reveals that for 13 of the 20 (65%) metrics, the direction of the correlation supports our hypothesis of a positive answer to **RQ1**. For 2 metrics, there is *no* correlation, and for the last 5 metrics, the correlation is in the opposite direction than expected. Table II 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 4 that do not, their correlation is *small*.

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 all the metrics, both the majority of metrics with medium-strength correlations and the majority of metrics overall correlate with code verifiability.

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 five metrics that anti-correlate with verifiability are concentrated in two of

TABLE II: Correlation results based on Kendall's τ (**K.'s τ**) for each dataset (**DS**) and **Metric**. A metric falls into one **Type**: **C**orrectness, **T**ime, **R**ating, & **P**hysiological. The expected direction of correlation (**+/-**), 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**) indicate a correlation of at least *medium*, 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	-	-0.02	-
	correct_verif	C	-	-0.03	-
	time_to_understand	T	+	-0.24*	n
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

the categories of metrics: rating and time. 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 III-C.

The correlations we observe are not supported by every dataset. DS6 and DS9 are relative outliers: neither provides much support for a correlation between understanding and verifiability. Both DS6 and DS9 have unusual features: DS6 is the only prior study that collected its data from professional developers rather than students, which is a threat to the validity of our conclusions; we discuss this threat and others in section V. DS9 is the smallest dataset, which makes it more prone to outliers. Further, there are three variants of this dataset with different comments; we explore whether these different versions would impact our results in section III-D2.

B. RQ2: Correlation for Different Verification Tools

To answer **RQ2**, we repeated the analysis used for **RQ1** (section III-A) independently for each tool (*i.e.*, no warning aggregation). We first summarize the conclusions for **RQ2** and then discuss in detail the results for each tool. Table III shows statistics about the number of snippets that each tool warned on and the total number of warnings, by dataset.

General conclusions for RQ2. The results for each tool are similar to the overall results for **RQ1** presented in section III-A (see the replication package for the raw results [38]), so we analyze the (1) the differences we observed between tools, and (2) the relative contribution to the results of each tool's output.

Our overall finding with respect to **RQ2** is that, of the four considered tools, three tools (the Checker Framework, JaTyC, and OpenJML) overall support a positive answer to **RQ1**; only Infer offers a reason for doubt. Given that Infer is intentionally unsound (*i.e.*, though it is based on a verification technique, it intentionally avoids issuing errors that its designers believed might be false positives, based on heuristic filters) and the other three tools are sound, the results suggests that *the false positives of tools attempting to verify the code correlate with human judgments of code understandability*.

1) Infer: Infer's results differ the most from our overall conclusions. It issues warnings on the fewest snippets (just 43/211 = 20.4%), mostly in datasets 3, 6, and 9 (with just one in DSF). Notably datasets 3, 6, and 9 are derived from open-source projects, not student code. Within those datasets, the snippets on which Infer warns are *anti-correlated* with 4/13 (ground-truth) metrics (*i.e.*, they do not match the expected correlation): *time_to_understand* (DS6), *readability_level_ba* (DS9), *readability_level_before* (DS9), and *time_to_read_complete* (DS9) (with *small* strength). 5 and 1 of the 13 metrics have *small* and *medium* correlation, respectively (supporting our hypothesis), and 3 metrics do not have a correlation (strength of *none*). These mixed results do not suggest any real correlation between Infer's warnings and understandability.

Infer's design may be responsible: it uses heuristics to filter low confidence warnings out of its output, because they might be false positives. The heuristics used by Infer might actually model the same complexity that the ground-truth metrics capture, and that if Infer could run with all filters disabled, this behavior would be reversed. We did run Infer with its –no-filtering option, but observed little difference with or without it (all results in the paper are with the option enabled). From Infer's public documentation [70], it is not clear if that option disables *all* of the filters the tool includes or whether it only enables a few checks that are disabled by default.

Another possible explanation is that Infer is the only inter-procedural analysis considered: that is, of the four tools, only it tracks dataflows across procedure boundaries rather than making worst-case assumptions about other parts of the code. To enable compilation of some snippets, we had to add "stub" methods—method signatures with empty bodies—for methods that the snippets use. Infer's analysis might be reasoning about these stubs and concluding that, if the program were run with the stub methods, no error could occur; the other tools more realistically assume that the stub methods will have content. In future work, we could validate this hypothesis for the snippets sourced from open-source projects by running Infer on the original open-source code in full, which would allow it to reason about the stubs' real

TABLE III: 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**

implementations; that is beyond the scope of this study.

2) The Checker Framework: The Checker Framework correlation results are similar to the overall results, mostly differing in strength, not direction. For 9/16 (56%) metrics where any correlation was possible (the Checker Framework issued no errors on DS2, so its 4 metrics are excluded), the correlation with that metric was as expected in direction, yet smaller in magnitude (0.1 on average). One of these nine metrics has a *medium* correlation, and the remaining 8 metrics have a *small* correlation. Three of the 16 metrics do not show a correlation (the *brain_deact_32* metric from DSF is the only one which differs from the **RQ1** results), while the remaining five metrics anti-correlate—these are the same metrics that anti-correlate for **RQ1**. The differences in magnitude may be due to the Checker Framework producing warnings on fewer snippets compared to JaTyC or OpenJML (table III), so there are more snippets for which no conclusions can be drawn.

3) JaTyC: The JaTyC correlation results are slightly weaker than the overall results from **RQ1**, though overall they point in the same direction. 9/20 (45%) metrics support the hypothesis, with mostly a slightly smaller magnitude (0.11 less on average) than in the overall results. Of those 11 metrics that did not agree with the overall results, 5 do not show a correlation (magnitude of *none*) and the remaining 6 anti-correlate in *small* magnitude, namely *brain_deact_32* and *time_to_understand* from DS2, *time_to_understand* from DS6, and *readability_level_ba*, *readability_level_before*, and *time_to_read_complete* from DS9. JaTyC does issue only three warnings on DS2—compared to over 500 on DS6—so the two anti-correlated metrics in that dataset might be outliers. Other than those two metrics, JaTyC mostly follows the overall results.

4) OpenJML: OpenJML closely follows the general pattern in table II, with 12/20 (60%) metrics correlating with verifiability. The only differences from the overall results are 3 metrics from DSF. *complexity_level* and *perc_correct_output*, which correlate when considering all tools, do not correlate when only considering OpenJML. *time_to_understand* anti-correlates when considering all tools, but *does* correlate when only considering OpenJML. While there are a few differences in magnitude, the general pattern is the same as the overall results: 6/7 metrics that correlate with a strength of *medium* or higher support a positive answer to **RQ1**.

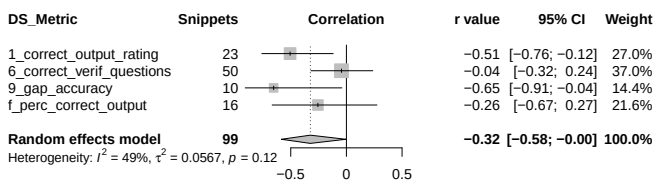
C. RQ3: Correlation for Different Types of Metrics

Figure 1 shows forest plots of the random-effects meta-analysis of our correlation results for the four metric types: *correctness*, *physiological time*, and *rating*. These plots display the observed correlation (Person's r **value**, obtained from the Kendall's τ value as described in section II-D), the 95% confidence interval (**95% CI**), and the estimated weight for each metric (**DS_Metric**). This information is shown numerically and graphically (the **Correlation** charts in fig. 1). Each gray box's size represents the 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. The horizontal lines indicate the confidence intervals calculated 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.

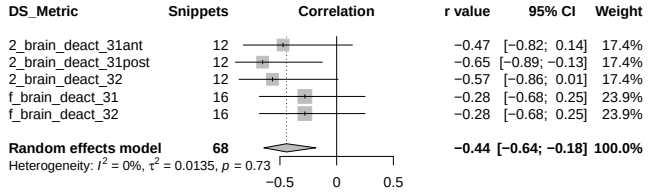
The plots in fig. 1 show the confidence intervals are rather wide for most metrics, which indicates relatively high variability in their correlations. The metrics from Datasets 1, 2, 9, and F have higher variability than those of DS3 and DS6, likely because of the smaller number of datapoints used for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

Figures 1a and 1b, 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.

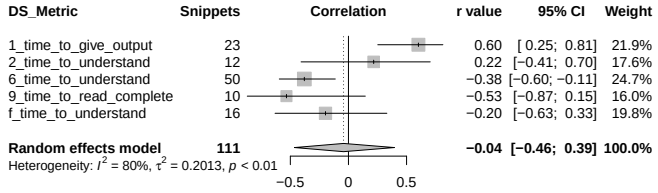
The results for both *time* and *rating* metrics (see figs. 1c and 1d, respectively) are more mixed. *Time* (fig. 1c) is the most varied metric type: the random-effects model shows that there is no discernible aggregate correlation. One possible explanation for this result is that, for all of these metrics, the



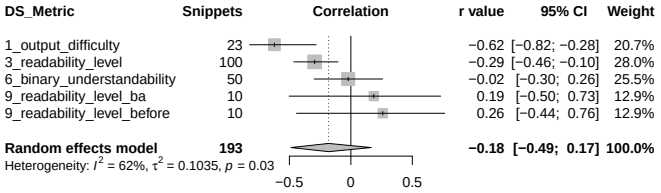
(a) Correctness metrics.



(b) Physiological metrics.



(c) Time metrics.



(d) Rating metrics.

Fig. 1: 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.

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. 1d), 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.

D. Robustness Experiments

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

1) Ablation Experiment: We re-computed the correlations for **RQ1** (section III-A) four times, each time leaving out one of the tools and aggregating the warnings produced by the other three. The goal was to test if one tool dominated the results. When individually excluding every tool except OpenJML, we found results consistent with sections III-A and III-B: we observed the same number of metrics with the same correlation direction that supports our hypothesis, with only changes in magnitude. The OpenJML results are less consistent: four metrics changed from having a correlation supporting our hypothesis to no correlation (*correct_output_rating* and *time_to_give_output* from DS1) or anti-correlation (*brain_deact_32* and *time_to_understand* from DS2). This means that OpenJML may be dominating the other tools in DS1 and DS2, which could be because it produces more warnings than the other tools *combined* for those datasets. While we observed in section III-B that Infer's correlation results are relatively dissimilar from the other three tools, this ablation experiment did not show any significant differences when Infer was not included. This might be because there are relatively few warnings from Infer, so it is dominated in the correlation analysis by the other tools. Our replication package [38] includes the full results of this ablation experiment.

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

Metric	+/-	NC	BC	GC
gap_accuracy	-	-0.24	-0.30	0.0
readability_level_ba	-	0.12	0.06	0.61*
readability_level_before	-	0.24	0.09	0.68*
time_to_read_complete	+	-0.24	-0.06	-0.12

2) Handling Code Comments in Dataset 9: DS9's original study created 3 versions for its 10 snippets, according to three types of code comments: good, bad, and no comments [26]. The results presented elsewhere in this section used the "No comments" (NC) version of DS9, because none of the four verifiers use comments as part of their logic. However, this choice might be source of possible bias, hence, 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 IV 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 good, these metrics are extremely anti-correlated with verifiability: that is, humans rated the snippets on which the tools issued more warnings as *more readable*. Though this result is surprising, that it only occurs when the comments are good—combined with the tiny

TABLE V: Correlation results (**Kendall's τ values**) for OpenJML, considering the approaches used to handle timeouts: (1) **ignoring** timeouts; (2) **under**-estimating the warnings hidden by timeouts; (3) is **over**-estimating the warnings hidden by timeouts. A * means statistical significance at $p < 0.05$.

		Approach		
DS	**Metric**	**1:Ignore**	**2:Under**	**3:Over**
3	*readability_level*	-0.10	-0.18*	-0.17*
6	*binary_understand*	0.03	-0.03	-0.02
	correct_verif	0.01	-0.02	-0.03
	time_to_understand	-0.24*	-0.25*	-0.24*

sample size of just 10 snippets—leads us to believe that the comments on the most difficult-to-verify methods are good enough to fully explain the method.

3) 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 lead to exponential run time in the size of the query, making it necessary to set a timeout when analyzing a collection of snippets. We used a 60 minute timeout, which resulted in 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 number of warnings in the dataset were issued on it. The data in table II 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 number of warnings that a no-timeout run of OpenJML would encounter, while approach 3 *overestimates* the number of warnings in a no-timeout run. We re-ran the correlation analysis under all three conditions; the results are in table V. We did not observe any significant differences between the treatments of timeouts—the overall direction and strength of the correlations are similar in all three treatments.

IV. 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 [29, 31] 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 are concerned with reducing false positives, because they are a key barrier to adopting static analyzers [71, 72]. 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 that provides such refactoring suggestions is more likely to be used in practice [73].

Researchers interested in code comprehension can benefit from our results. That there is a correlation between verifiability and understandability suggests 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 *any* 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.

V. 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 (section III-B) and with an ablation experiment (section III-D1). 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. The measurements for 5/6 datasets were collected from student subjects (only DS6 used professional software engineers), so our results may not apply to more experienced programmers.

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.

VI. 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 [7, 8, 11–15], though the concept is not easy to define due to different interpretations [9, 10]. Most metrics rely on simple, syntactic properties such as code size or branching paths [2, 11]. Applications of these metrics include detecting complex code so that developers can simplify it during software evolution [2, 3, 21]. The motivation is that complex code is harder to understand, while simple code is easier to understand [7, 8], 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.* [8] 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.* [74].

Researchers have explored the limitations of classical complexity metrics [7, 8, 15, 23, 27]. For example, Ajami *et al.* [7] found that different code constructs (*e.g.*, `if`s 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 [2]. Recent work has proposed new metrics such as Cognitive Complexity (COG) [75, 76], which assigns different weights to different code constructs. Muñoz *et al.* [22] 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 III-C 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 [2, 7, 15, 24, 26, 35, 36, 77]. Since it is difficult to precisely define understandability, some studies have used it interchangeably with readability [22, 26, 37, 78, 79] (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 [2, 35, 36], biometrics sensors [80–82], or eye-tracking devices [80, 83–85], 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 [7, 24], code (micro-)patterns [15, 25, 26], identifier quality and style [86, 87], code comments [26], information gathering tasks [35, 84, 87, 88], comprehension tools [89], code reading behavior [90–92], code authorship [93], high-level comprehension strategies [92], programmer experience [5, 86], and the use of traditional complexity metrics [94]. 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. Though verification and static analysis tools are becoming more common in industry [95, 96], studies of their use and the challenges developers face in deploying them [72, 95, 97–99] suggest false positives remain a problem in practice [71, 72]. While researchers have proposed prioritization strategies to surface warnings that are more likely to be real defects to developers [73, 97], recent evidence suggests that they may not have much of an impact when deployed [99]. 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].

VII. 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] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Symp. on the Found. of Soft. Eng. (FSE)*, 2012, pp. 1–11.

[2] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: An fMRI study," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2021, pp. 524–536.

[3] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old habits die hard: Why refactoring for understandability does not give immediate benefits," in *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*, 2015, pp. 504–507.

[4] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *Trans. on Soft. Eng. and Methodology (TSEM)*, vol. 23, no. 4, pp. 1–37, 2014.

[5] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *Trans. on Soft. Eng. (TSE)*, vol. 44, no. 10, pp. 951–976, 2018.

[6] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer - an investigation of how developers spend their time," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2015, pp. 25–35.

[7] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms — what really affects code complexity?" *Emp. Soft. Eng.*, vol. 24, no. 1, pp. 287–328, 2019.

[8] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *Trans. on Soft. Eng. (TSE)*, vol. 47, no. 3, pp. 595–613, 2019.

[9] V. Antinyan, M. Staron, and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," *Emp. Soft. Eng.*, vol. 22, no. 6, pp. 3057–3087, 2017.

[10] V. Antinyan, "Evaluating essential and accidental code complexity triggers by practitioners' perception," *IEEE Soft.*, vol. 37, no. 6, pp. 86–93, 2020.

[11] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Jour. of Sys. and Soft.*, vol. 128, pp. 164–197, 2017.

[12] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *Trans. on Soft. Eng. (TSE)*, vol. SE-5, no. 2, pp. 96–104, 1979.

[13] H. Zuse, "Criteria for program comprehension derived from software complexity metrics," in *Workshop on Prog. Compr*, 1993, pp. 8–16.

[14] H. M. Sneed, "Understanding software through numbers: A metric based approach to program comprehension," *Jour. of Soft. Maint.: Research and Practice*, vol. 7, no. 6, pp. 405–419, 1995.

[15] A. Jbara and D. G. Feitelson, "How programmers read regular code: a controlled experiment using eye tracking," *Emp. Soft. Eng.*, vol. 22, no. 3, pp. 1440–1477, 2017.

[16] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Trans. on Soft. Eng. (TSE)*, vol. 20, no. 6, pp. 476–493, 1994.

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

[18] M. H. Halstead, *Elements of Soft. Science*. Elsevier, 1977.

[19] T. McCabe, "A complexity measure," *Trans. on Soft. Eng. (TSE)*, vol. SE-2, no. 4, pp. 308–320, 1976.

[20] D. Beyer and A. Fararooy, "A simple and effective measure for complex low-level dependencies," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2010, pp. 80–83.

[21] J. García-Munoz, M. García-Valls, and J. Escribano-Barreno, "Improved metrics handling in sonarqube for software quality monitoring," in *Intl. Conf. on Distributed Comp. and Art. Intel.*, 2016, pp. 463–470.

[22] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM)*, 2020, pp. 1–12.

[23] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, "Exploring software measures to assess program comprehension," in *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM)*, 2011, pp. 127–136.

[24] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, "An empirical study assessing source code readability in comprehension," in *Intl. Conf. on Soft. Maint. and Evol. (ICSME)*, 2019, pp. 513–523.

[25] C. Langhout and M. Aniche, "Atoms of confusion in java," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2021, pp. 25–35.

[26] J. Börstler and B. Paech, "The role of method chains and comments in software readability and comprehension—an experiment," *Trans. on Soft. Eng. (TSE)*, vol. 42, no. 9, pp. 886–898, 2016.

[27] C. Kaner, S. Member, and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *Intl. Soft. Metrics Symp. (METRICS)*, 2004.

[28] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*, Seattle, WA, USA, July 2008, pp. 201–212.

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

[30] The OpenJML Developers, "OpenJML," https://www.openjml.org/, 2022.

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

[32] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[33] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symp.* Springer, 2015, pp. 3–11.

[34] J. Mota, M. Giunti, and A. Ravara, "Java typestate checker," in *Intl. Conf. on Coord. Lang. and Models*. Springer, 2021, pp. 121–133.

[35] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2014, pp. 378–389.

[36] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "A look into programmers' heads," *Trans. on Soft. Eng. (TSE)*, vol. 46, no. 4, pp. 442–462, 2018.

[37] R. Buse and W. Weimer, "Learning a metric for code readability," *Trans. on Soft. Eng. (TSE)*, vol. 36, no. 4, pp. 546–558, 2009.

[38] Anonymous Author(s), "Online replication pacakge," https://tinyurl.com/5azcv3jm, 2022.

[39] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[40] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[41] P. Cousot and R. Cousot, "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, Jan. 1977, pp. 238–252.

[42] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Trans. on Computer-Aided Design of Integ. Circ. and Sys.*, vol. 27, no. 7, pp. 1165–1178, 2008.

[43] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Intl. Workshop on Computer Science Logic*. Springer, 2001, pp. 1–19.

[44] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Principles of Programming Languages (POPL)*, 2009, pp. 289–300.

[45] J. S. Foster, M. Fähndrich, and A. Aiken, "A theory of type qualifiers," in *Conf. on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, USA, May 1999, pp. 192–203.

[46] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller, "Building and using pluggable type-checkers," in *Intl. Conf. on Soft. Eng. (ICSE)*, Waikiki, Hawaii, USA, May 2011, pp. 681–690.

[47] M. Kellogg, M. Ran, M. Sridharan, M. Schäf, and M. D. Ernst, "Verifying object construction," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2020, pp. 1447–1458.

[48] M. Kellogg, N. Shadab, M. Sridharan, and M. D. Ernst, "Lightweight and modular resource leak verification," in *European Soft. Eng. Conf. and Symp. on the Found. of Soft. Eng. (ESEC/FSE)*, 2021.

[49] M. Kellogg, V. Dort, S. Millstein, and M. D. Ernst, "Lightweight verification of array indexing," in *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*, 2018, pp. 3–14.

[50] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst, "A type system for format strings," in *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*, 2014, pp. 127–137.

[51] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*, Beijing, China, June 2012, pp. 20–26.

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

[53] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Sngineering*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986.

[54] M. Emmi, L. Hadarean, R. Jhala, L. Pike, N. Rosner, M. Schäf, A. Sengupta, and W. Visser, "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)*, 2021, pp. 1416–1426.

[55] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a Java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA 1998)*. Citeseer, 1998, pp. 404–420.

[56] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Programming Language Design and Implementation (PLDI)*, 2002, pp. 234–245.

[57] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, Mar. 2008, pp. 337–340.

[58] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "JayHorn: A framework for verifying Java programs," in *Intl. Conf. on Computer Aided Verification (CAV)*. Springer, 2016, pp. 352–358.

[59] M. Schäf and P. Rümmer, personal communication, 2022.

[60] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An extensible approach to validating the correct usage of cryptographic APIs," in *European Conf. on Object-Oriented Programming (ECOOP)*, Amsterdam, Netherlands, July 2018, pp. 10:1–10:27.

[61] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *Intl. Jour. on Soft. Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[62] J. Murray, "Likert data: what to use, parametric or non-parametric?" *Intl. Jour. of Business and Social Science*, vol. 4, no. 11, 2013.

[63] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[64] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, 3rd ed. Routledge, 2002.

[65] M. Borenstein, L. V. Hedges, J. P. T. Higgins, and H. R. Rothstein, *Introduction to Meta-Analysis*. John Wiley & Sons, 2009.

[66] K. Sidik and J. N. Jonkman, "Simple heterogeneity variance estimation for meta-analysis," *Jour. of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 54, no. 2, pp. 367–384, 2005.

[67] D. Walker, "Jmasm9: Converting kendall's tau for correlational or meta-analytic analyses," *Jour. of M. A. Stat. Meth.*, vol. 2, no. 2, 2003.

[68] M. Harrer, P. Cuijpers, T. A. Furukawa, and D. D. Ebert, *Doing Meta-Analysis with R: A Hands-On Guide*. Chapman and Hall/CRC, 2021.

[69] G. Schwarzer, "meta: General package for meta-analysis," 2022. [Online]. Available: https://CRAN.R-project.org/package=meta

[70] "Infer's manual," 2022. [Online]. Available: https://fbinfer.com/docs/man-infer/

[71] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Intl. Conf. on Soft. Eng. (ICSE)*, 2013, pp. 672–681.

[72] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Intl. Symp. on Soft. Testing and Analysis (ISSTA)*, 2022, pp. 532–543.

[73] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Intl. Conf. on Soft. Eng. (ICSE)*, vol. 1, 2015, pp. 598–608.

[74] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, ""automatically assessing code understandability" reanalyzed: combined metrics matter," in *Intl. Conf. on Mining Soft. Repositories (MSR)*, 2018, pp. 314–318.

[75] G. A. Campbell, "Cognitive complexity: an overview and evaluation," in *Intl. Conf. on Technical Debt*, 2018, pp. 57–58.

[76] R. Saborido, J. Ferrer, F. Chicano, and E. Alba, "Automatizing software cognitive complexity reduction," *IEEE Access*, vol. 10, pp. 11 642–11 656, 2022.

[77] D. Gopstein, A.-L. Fayard, S. Apel, and J. Cappos, "Thinking aloud about confusing code: a qualitative investigation of program comprehension and atoms of confusion," in *European Soft. Eng. Conf. and Symp. on the Found. of Soft. Eng. (ESEC/FSE)*, 2020, pp. 605–616.

[78] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Emp. Soft. Eng.*, vol. 25, no. 6, pp. 5374–5412, 2020.

[79] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in *Intl. Conf. on Soft. Maint. and Evol. (ICSME)*, 2020, pp. 348–359.

[80] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2014, pp. 402–413.

[81] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile, "A replication study on code comprehension and expertise using lightweight biometric sensors," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2019, pp. 311–322.

[82] M. K.-C. Yeh, D. Gopstein, Y. Yan, and Y. Zhuang, "Detecting and comparing brain activity in short program comprehension using eeg," in *Frontiers in Education Conf. (FIE)*, 2017, pp. 1–5.

[83] R. Turner, M. Falcone, B. Sharif, and A. Lazar, "An eye-tracking study assessing the comprehension of c++ and python source code," in *Symp. on Eye Tracking Research and Applications*, 2014, pp. 231–234.

[84] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Emp. Soft. Eng.*, vol. 18, no. 2, pp. 219–276, 2013.

[85] A. Abbad-Andaloussi, T. Sorg, and B. Weber, "Estimating developers' cognitive load at a fine-grained level using eye-tracking measures," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2022, pp. 111–121.

[86] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking code readability, structure, and comprehension among novices: It's complicated," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2019, pp. 84–94.

[87] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *European Soft. Eng. Conf. and Symp. on Found. of Soft. Eng. (ESEC/FSE'17)*, 2017, pp. 140–150.

[88] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *European Soft. Eng. Conf. and the Symp. on on the Found. of Soft. Eng. (ESEC/FSE)*, 2007, pp. 361–370.

[89] M. A. D. Storey, K. Wong, and H. A. Müller, "How do program understanding tools affect how programmers understand programs?" *Science of Computer Programming*, vol. 36, no. 2, pp. 183–207, 2000.

[90] N. Peitek, J. Siegmund, and S. Apel, "What drives the reading order of programmers? an eye tracking study," in *Intl. Conf. on Prog. Compr. (ICPC)*, 2020, pp. 342–353.

[91] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic, "Developer reading behavior while summarizing java methods: Size and context matters," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2019, pp. 384–395.

[92] J. Siegmund, "Program comprehension: Past, present, and future," in *Intl. Conf. on Soft. Analysis, Evolution, and ReEng. (SANER)*, vol. 5, 2016, pp. 13–20.

[93] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2010, pp. 385–394.

[94] M. Wyrich, A. Preikschat, D. Graziotin, and S. Wagner, "The mind is a powerful place: How showing code comprehensibility metrics influences code understanding," in *Intl. Conf. on Soft. Eng. (ICSE)*, 2021, pp. 512–523.

[95] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "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, 2016, pp. 470–481.

[96] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Intl. Symp. on Soft. Reliab. Eng.*, 2004, pp. 245–256.

[97] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Emp. Soft. Eng.*, vol. 25, no. 2, pp. 1419–1457, 2020.

[98] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in *Symp. on Usable Privacy and Security (SOUPS)*, 2020, pp. 221–238.

[99] N. Mansoor, T. Muske, A. Serebrenik, and B. Sharif, "An empirical assessment of repositioning of static analysis alarms," in *Intl. Working Conf. on Source Code Analysis & Manipulation*, 2022.