On the Relationship between Code Verifiability and Understandability

Kobi Feldman  
jfeldman@wm.edu  
College of William & Mary  
Williamsburg, Virginia, USA

Martin Kellogg  
martin.kellogg@njit.edu  
New Jersey Institute of Technology  
Newark, New Jersey, USA

Oscar Chaparro  
oscarcb@wm.edu  
College of William & Mary  
Williamsburg, Virginia, USA

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. We empirically validate this assumption by comparing the number of warnings produced by four state-of-the-art verification tools on 211 snippets of Java code with 20 metrics of code comprehensibility from human subjects in six prior studies.

Our experiments, based on a statistical (meta-)analysis, show that, in aggregate, there is a small correlation ($r = 0.23$) between understandability and verifiability. The results support 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 code comprehensibility: verification tools may have ancillary benefits to understandability, and measuring understandability may require reasoning about semantic, not just syntactic, code properties.

CCS CONCEPTS
- • Software and its engineering → Formal software verification; • General and reference → Empirical studies.

KEYWORDS
Verification, static analysis, code comprehension, meta-analysis

ACM Reference Format:

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, 56, 69, 90]. However, understanding code is challenging and time-consuming for developers: studies [59, 104] 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, 69, 78]: 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, 14]. Whether the complexity is essential or accidental, understanding complex code demands high cognitive effort from developers [3, 78].

Researchers have proposed many metrics to approximate code complexity [3, 19, 21, 39, 41, 64, 85, 106] using vocabulary size (e.g., Halstead’s complexity [36]), program execution paths (e.g., McCabe’s cyclomatic complexity [58]), 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, 34, 69], or to predict developers’ cognitive load when understanding code [62, 69, 78]. 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, 69, 78]. 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, 13, 26, 41, 43, 52], which diverges from the simplistic way metrics (e.g., McCabe’s) measure code complexity [3, 26, 41, 45, 78].

In this paper, we investigate the relationship between understandability and code verifiability—how easy or hard it is for a developer to use a 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 in the software verification community that simpler code is both easier to verify by verification tools and easier to understand by developers. For example, the Checker Framework [68] user manual states this assumption explicitly in its advice about unexpected warnings: “rewrite your code to be simpler for the checker to analyze; this is likely to make it easier for people to understand, too” [91]. The documentation of the OpenJML verification tool says [93], “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” [67]. This assumption is widely held by verification experts but has never been validated empirically.

The intuition behind this assumption is that a verifier can handle a certain amount of code complexity before it issues a warning. If
it is possible to remove the warning by changing the code, then the complexity that caused it must be accidental rather than essential, and therefore removing the warning 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. A verifier would need to model exceptional control flow to avoid a false positive warning. Alternatively, a verifier might warn about code that makes unstated assumptions. For example, by dereferencing a possibly-null pointer without checking it first, the code assumes that the pointer has already been checked. A verifier might warn that this is unsafe unless a human provides a specification that the pointer is non-null. In that case, the verifier can verify the dereference, but then must check that the value assigned to the variable really is non-null at each assignment. A warning because of a missing specification can also indicate complexity that a human might need to reason about to understand the code: the human might need to determine why it is safe to dereference the pointer.

Our goal is to empirically validate the purported relationship between verifiability and understandability—and therefore either confirm or refute the assumption that easy-to-understand code is easy to verify (and vice-versa). To do so, we need to measure 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 [74], 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 false positive warnings due to the verifier’s imprecision or due to the need to state code assumptions as specifications. The combination of false positive warnings and warnings about unstated assumptions (which we refer to as “false positives” or “warnings” for brevity) is a good proxy for verifiability because the fewer such warnings 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 is tested empirically. Our study compares the number of warnings produced by three state-of-the-art, sound static code verifiers [60, 68, 93] and one industrial-strength, unsound static analysis tool based on a sound core [16] with ≈18k measurements code understandability proxies collected from humans in six prior studies [13, 15, 69, 71, 78, 82] for 211 Java code snippets. Such measurements come from 20 metrics in four categories [62]: (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 [11, 37] to examine the correlation between verifiability and these understandability metrics in aggregate. Given the small sample sizes of the original studies and the danger of multiple comparisons, this meta-analysis technique permits us to draw methodologically-sound conclusions about the overall trends.

We found a small correlation between verifiability and the proxies for understandability, in aggregate ($r = 0.23$); individually, 13 of 20 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.

One implication of this result is that our results provide evidence for a relationship between the semantics of a piece of code and its understandability, which may explain (in part) the apparent ineffectiveness of prior syntactic approaches. This implies that the verifiability of a code snippet, as measured automatically by the warnings issued by extant verification tools, might be a useful input to models of code understandability [15, 78, 94]. Another implication is that when using a verification tool, developers should consider making 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. If a developer makes a change to remove a false positive without changing the code’s semantics, any complexity they remove must have been accidental. This means that verification tools provide a secondary benefit beyond their guarantees of the absence of errors: code that can be easily verified should be easier for future developers to improve and extend.

In summary, the main contributions of this paper are:

- empirical evidence of a correlation between code understandability and verifiability derived via meta-analysis, supporting the common assumption that easier-to-verify code is easier for humans to understand (and vice-versa). The results have implications for the design and deployment of verification tools and for developing more accurate automated metrics of code comprehensibility; and
- an online replication package [27] that enables verification and replication of our results and future research.

## 2 EMPirical STUDY DESIGN

Our goal is to assess the correlation between human-based code comprehensibility metrics and code verifiability—i.e., how many 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 these 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** What is the impact of each verification tool on the aggregated correlation results?
- **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 individual metrics correlate with verifiability. However, due to the limitations of prior studies, sample sizes for the metrics considered individually are too small to draw...
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.

<table>
<thead>
<tr>
<th>DS</th>
<th>Snippets</th>
<th>NCOLC</th>
<th>Participants</th>
<th>Understandability Task</th>
<th>Understandability Metrics</th>
<th>Meas.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23 CS algorithms</td>
<td>6 - 20</td>
<td>41 students</td>
<td>Determine prog. output</td>
<td>C: correct_output_rating (3-level correctness score for program output)</td>
<td>2,829</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R: output_difficulty (5-level difficulty score for program output)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_give_output (seconds to read program and answer a question)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_31ant (deactivation of brain area BA31ant)</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_31post (deactivation of brain area BA31post)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_32 (deactivation of brain area BA32)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_understand (seconds to understand program within 60 seconds)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12 CS algorithms</td>
<td>7 - 15</td>
<td>16 students</td>
<td>Determine prog. output</td>
<td>C: correct_output_rating (3-level correctness score for program output)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R: output_difficulty (5-level difficulty score for program output)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_give_output (seconds to read program and answer a question)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_31ant (deactivation of brain area BA31ant)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_31post (deactivation of brain area BA31post)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_32 (deactivation of brain area BA32)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_understand (seconds to understand program within 60 seconds)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100 OSS methods</td>
<td>5 - 13</td>
<td>121 students</td>
<td>Rate prog. readability</td>
<td>R: readability_level (5-level score for readability/ease to understand)</td>
<td>12,100</td>
</tr>
<tr>
<td>6</td>
<td>50 OSS methods</td>
<td>18 - 75</td>
<td>50 students and 13 developers</td>
<td>Rate underst./answer Qs</td>
<td>C: correct_verif_questions (% of correct answers to verification questions)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_understand (seconds to understand program)</td>
<td>1,197</td>
</tr>
<tr>
<td>9</td>
<td>10 OSS methods</td>
<td>10 - 34</td>
<td>104 students</td>
<td>Rate read./complete prog.</td>
<td>R: readability_level (5-level avg. score for readability b/a code completion)</td>
<td>716</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R: readability_level_before (5-level score for readability before code completion)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_read_complete (avg. seconds to rate readability and complete code)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>16 CS algorithms</td>
<td>7 - 19</td>
<td>19 students</td>
<td>Determine prog. output</td>
<td>P: brain_deact_31ant (deactivation of brain area BA31)</td>
<td>935</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_31post (deactivation of brain area BA31post)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P: brain_deact_32 (deactivation of brain area BA32)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R: complexity_level (score for program complexity)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C: perc_correct_output (% of subjects who correctly gave program output)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T: time_to_understand (seconds to understand program within 60 seconds)</td>
<td></td>
</tr>
</tbody>
</table>

reliable conclusions. Therefore, RQ2 asks whether there is a pattern to the answers to RQ1 that summarizes the overall correlation trend. We answer RQ2 statistically by combining the results of the individual metrics targeted by RQ1 with a meta-analysis.

RQ3 asks whether verifiability, measured using only one tool’s warnings, correlates with comprehensibility. Intuitively, we aim to check (1) if the patterns of correlations are similar across tools, (indicating generalizability), and (2) whether any particular tool dominates the results. To answer the second part, we use a “leave-one-out” ablation analysis, dropping each tool individually. RQ4 asks whether there is any difference in correlation between verifiability and different proxies for code comprehensibility. Based on prior work [62], we focus on four metric categories: correctness, rating, time, and physiological. Together, the answers to RQ3 and RQ4 help us explain our RQ1 and RQ2 results: they explore the tool(s) and metric(s) responsible for the observed correlations.

To answer our RQs, we first compiled a set of human-based code comprehensibility measurements from prior studies (section 2.1). Then, we defined our metric for code verifiability (section 2.2) and executed four verification-based tools on the same code snippets to measure how often each snippet cannot be verified (sections 2.3 and 2.4). Next, we conducted an analysis of the warnings produced by the verifiers to ensure that they met our definition of “false positive” (section 2.5). Finally, we correlated the comprehensibility metrics with the number of warnings produced by the tools and analyzed the correlation results using meta-analysis (section 2.6).

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 a causation study, which we leave as future work.

2.1 Code and Understandability Datasets

We used existing datasets (DSs) from six prior understandability studies [13, 15, 69, 71, 78, 82], 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—see table 1.

To select these datasets, we leveraged the systematic literature review conducted by Muñoz et al. [62], who found ten studies that measured code understandability with publicly available data. From these ten studies, we selected the five studies whose snippets were written in Java, since the verifiers we consider only work on Java code—see section 2.3. To identify the datasets and facilitate replication, we use the same nomenclature as Muñoz et al.’s: DS1, DS2, DS3, DS6, and DS9. Since those five studies were conducted before 2020, we performed a literature search of additional comprehensibility studies from 2020 to early 2023 and found the one by Peitek et al. [69] (Dataset F or DSF), who also used Java snippets.

In total, we used ≈18k understandability measurements (see the “Meas.” column in table 1) for 211 Java code snippets, collected from 364 human subjects using 20 metrics. The 211 snippet programs are 5 to 75 non-comment/blank LOC or NCOLC—17 NCOLC on avg.—with different complexity levels, as reported by the methodology of their respective studies [13, 15, 69, 71, 78, 82]. Datasets 3, 6, and 9 derive from open source software projects (OSS)—e.g., Hibernate, JFreeChart, Antlr, Spring, & Weka [13, 15, 78]—and the other datasets are implementations of algorithms from 1st-year programming courses (e.g., reversing an array) [69, 71, 82]. The original studies selected short code snippets to control for potential confounding factors that may affect understandability [69, 71, 82].

We selected the understandability metrics used in the meta-study conducted by Muñoz et al. [62]—see table 1 for the metrics, their type, and a brief description of them (our replication package has full descriptions [27]). We also used Muñoz et al.’s categorization of the metrics. Correctness metrics (marked with a C in table 1) 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 [82]).

Study participants were mostly CS undergraduate/graduate students with intermediate-to-high programming experience, as reported in the original papers [13, 15, 69, 71, 78, 82]. Only DS6’s study included professional developers [78]—see table 1.

We used all the available data from the six original studies. Their measurements come in aggregate or individual form: e.g., the physiological measurements in DS2 [71] and DSF [69] are provided per snippet averaged across participants, while the DS3 measurements come for each participant and snippet [15]. Some studies also included an uneven numbers of participants per snippet, due to different methodological decisions. For example, DS9’s study included a random assignment of participants to one of six sequences of five snippets [13]. In DS6’s study [78], six snippets were understood by eight participants and the remaining 44 snippets were understood by nine. For these reasons, each dataset’s number of measurements (see the “Meas.” column in table 1) is not always divisible by the number of snippets and participants.

### 2.2 Proxy for Code Verifiability

We define code verifiability as the effort that a developer incurs when using a verification tool to prove safety properties about a snippet of code. Since measuring this effort is infeasible without running a study with verification tool users, we use false positive counts as an automatable proxy for verifiability.

We define a “false positive” as a verifier warning that indicates the verifier is unable to prove that a code snippet is correct due to a weakness in the verifier (i.e., undecidability [74]) or due to a missing code specification, which a human should provide. In effect, a false positive warning represents a “fact” about the code that the verifier needs, but cannot prove with the snippet’s code only.

Our definition of “false positive” differs from the typical one when evaluating the precision and recall of a verifier. In that context, it is assumed that correct specifications are explicit and available, and “false positive” means a fact that the verifier cannot prove, even with a specification. Conversely, in our context, we want a proxy for the difficulty of verifying a snippet. That difficulty includes both writing specifications and suppressing false positive warnings, so it is sensible to include both in our proxy for verifiability. In other words, the fewer warnings a verifier issues, the less work a developer using the verifier must do to verify a code snippet.

We considered two other proxies for verifiability: number of false positives after writing specifications and number of “facts” verified about the code. We discarded the former proxy because the full context of how the snippets are intended to be used is not available and because writing specifications is error- and bias-prone. The latter was discarded because none of the verifiers provide the proxy directly, and because approximating whether or not a verifier needs to even check a fact is undecidable for some properties considered by a verifier (e.g., determining what is considered a resource in the code by a resource leak verifier [48]), so a precise count is impossible.

### 2.3 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 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 soundness [35] (and intentionally-unsound tools) because practical verification tools commonly only make guarantees about the absence of defects under certain conditions.

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 warning—for tools that verify a property that is irrelevant to the snippets (and so do not issue any 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) verifiers are usually language-dependent, (2) many prior code comprehensibility studies on human subjects used Java—e.g., 5/10 studies in Muñoz et al. [62] and no other language has more than 2/10 studies—and (3) 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 4.

#### 2.3.1 Selected Verification Tools

By applying the criteria defined above, we selected four verification tools:

**Infer** [16] is an unsound, industrial static analysis tool based on a sound core of separation logic [65] and bi-abduction [17]. 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. Infer warns about possible null dereferences, data races, and resource leaks. We used Infer version 1.1.0.

The **Checker Framework** [68] is a collection of pluggable type-checkers [30], 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 type-checkers. We used the nine that satisfy criterion 4, which prevent programming mistakes related to: nullness [24, 68], interning [24, 68], object construction [47], resource leaks [48], array bounds [46], signature strings [24], format strings [101], regular expressions [86], and optionals [92]. We used Checker Framework version 3.21.3.
The Java Typestate Checker (JaTyC) [60] is a typestate analysis [88]. 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 [60] (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. JaTyC warns about possible null dereferences, incomplete protocols on objects that have a defined lifecycle (such as sockets or files), and about violations of its ownership discipline, which is similar in spirit to Rust’s [50]. We used JaTyC commit b438683.

OpenJML [93] 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) [54]; it is the latest in a series of tools verifying JML specifications by reduction to SMT going back to ESC/Java [29]. OpenJML verifies the absence of a collection of a common programming errors, including out-of-bounds array accesses, null pointer dereferences, integer over- and underflows, and others. We used OpenJML 0.17.0-alpha-15 with the default solver z3 [23] v. 4.3.1.

2.3.2 Verification Tools Considered but Not Used. We considered and discarded three other verifiers: JayHorn [44], which fails criterion 2 [79]; CogniCrypt [51], which fails criterion 3; and Java Pathfinder [38], which fails criterion 4.

2.4 Snippet Preparation and Tool Execution
We acquired the snippets from prior work [62, 69] but had to make some modifications to prepare them for tool execution. DS3 included 4 commented-out snippets, which we uncommented. To make the snippets compilable, we created “stubs” for the classes, method calls, etc. they use without modifying the snippets themselves. Since the the snippets themselves did not change, their underlying, measured code comprehensibility did not change either: in the original studies, the snippets were provided to the humans in isolation. At the same time, our modifications would change the program’s state if the snippets were to be executed. We performed a manual analysis of tool warnings to ensure our modifications did not cause spurious warnings (see section 2.5). We created scripts to execute the verifiers on the snippets and display all verification failures for each tool. Table 2 shows descriptive statistics of the warnings issued by each tool on each dataset.

### Table 2: Number of snippets each tool warns on and the total number of warnings per dataset.

<table>
<thead>
<tr>
<th>Tool \ Dataset</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>F</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infer</td>
<td>0/23</td>
<td>0/12</td>
<td>1/100</td>
<td>5/50</td>
<td>0/10</td>
<td>1/16</td>
<td>7/211</td>
</tr>
<tr>
<td>Checker Fr.</td>
<td>3/23</td>
<td>0/12</td>
<td>18/100</td>
<td>28/50</td>
<td>4/10</td>
<td>3/16</td>
<td>56/211</td>
</tr>
<tr>
<td>JaTyC</td>
<td>3/23</td>
<td>1/12</td>
<td>88/100</td>
<td>40/50</td>
<td>10/10</td>
<td>2/16</td>
<td>144/211</td>
</tr>
<tr>
<td>OpenJML</td>
<td>14/23</td>
<td>6/12</td>
<td>69/100</td>
<td>41/50</td>
<td>10/10</td>
<td>13/16</td>
<td>153/211</td>
</tr>
<tr>
<td>All Tools</td>
<td>17/23</td>
<td>7/12</td>
<td>93/100</td>
<td>48/50</td>
<td>10/10</td>
<td>15/16</td>
<td>190/211</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool \ Dataset</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>9</th>
<th>F</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infer</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Checker Fr.</td>
<td>7</td>
<td>0</td>
<td>51</td>
<td>83</td>
<td>4</td>
<td>3</td>
<td>148</td>
</tr>
<tr>
<td>JaTyC</td>
<td>14</td>
<td>3</td>
<td>327</td>
<td>537</td>
<td>37</td>
<td>6</td>
<td>924</td>
</tr>
<tr>
<td>OpenJML</td>
<td>29</td>
<td>11</td>
<td>808</td>
<td>219</td>
<td>24</td>
<td>29</td>
<td>1120</td>
</tr>
<tr>
<td>All Tools</td>
<td>50</td>
<td>14</td>
<td>1,187</td>
<td>846</td>
<td>65</td>
<td>39</td>
<td>2,201</td>
</tr>
</tbody>
</table>

2.5 Code Correctness and Warning Validation
We assumed that every warning issued by a verifier about a snippet is a “false positive”, according to the definition we presented in section 2.2. In effect, this means that we are treating the snippets as if they are correct. For example, if a snippet dereferences a pointer without a null check, we assume that pointer is non-null; if a snippet accesses an array without checking a bound, we assume that the bound was checked elsewhere in the program, etc. Each verifier warning therefore represents some fact that the verifier needs, but cannot prove with the snippet’s code only. We consider these reasonable assumptions because: (1) no context about the snippets is available (or was presented to the human subjects in the prior studies), and (2) the snippets are likely to be correct as researchers showed them to humans in prior comprehensibility studies.

However, to check that these assumptions do not skew our correlation analysis, we manually validated whether a representative sample of tool warnings were indeed false positives (according to our definition from section 2.2). After executing the verifiers, one author examined a representative subset of the warnings (a sample of 344 of 2,201, at 95% confidence level and 5% of error margin) and recorded the cause of each. A second author examined the first author’s assessment and both authors discussed the cases where the assessment was incorrect or needed more details, reaching consensus in case of disagreements (of which there were < 5). Of these 344 warnings, none were “real bugs” in the sense that they are guaranteed to make the code fail when executed. Many do represent potential bugs: that is, code that does not check boundary conditions such as nullability; however, these warnings could be removed by writing a specification for the relevant verifier indicating the assumptions made by the snippet. This means these warnings are all false positives, according to our definition from section 2.2.

In the sample, the most common reasons for a verifier to warn were: (1) violation’s of JaTyC’s Rust-like rules for mutability, and 2) violations of the verifiers’ assumptions about nullability. Other common causes were possible integer over- and underflows, too large or too small array indexes, and unsafe casts. Our analysis of warnings for each snippet indicates a fairly uniform distribution of warning types over the datasets. Our replication package provides our detailed analysis of warnings [27].

2.6 Correlation and Analysis Methods

2.6.1 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 were 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 [82]. Following Muñoz et al. [62], 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 [61], we observed that the Likert items in the original datasets represent discrete values on continuous scales [62], so it is reasonable to average these values to obtain one measurement per snippet. All physiological measurements given by the original studies are averaged across all participants who understood a given 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, for RQ3, we performed an ablation experiment to investigate possible effects of warning imbalances on correlation.

### 2.6.2 Statistical Methods.

We used Kendall’s τ [49] to correlate the individual comprehensibility metrics and the tool warnings because (1) it does not assume the data to be normally distributed and have a linear relationship [20], (2) it is robust to outliers [20], and (3) it has been used in prior comprehensibility studies [62, 69, 78]. As in previous studies [69, 78], we follow Cohen’s guidelines [20] 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 $0.5 \leq |\tau|$.

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 RQ1 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, 62]. 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 [80]).

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 estimate an aggregated correlation. In our case, however, we first combine the results of the individual correlation analyses (i.e., for each metric) for each dataset into a single aggregated correlation per dataset, to avoid the “unit-of-analysis” problem (see [37], §3.5.2). This problem arises in meta-analysis when there are inputs that are not independent (i.e., are themselves correlated), typically because they represent multiple measurements obtained on the same population. Because most of our datasets include multiple metrics that were derived from the same subjects and snippets, and therefore, are related (e.g., readability_level_before and readability_level_after from DS9 depend on one another), a naive application of meta-analysis that treated each metric as independent would over-weight studies with multiple metrics, because it would “double-count” their statistical power (i.e., multiply the statistical power of the study by the number of metrics it contains). We confirmed that most of the combinations of metrics within a single study showed medium or large correlations (19/28 combinations are medium or large correlations; of those, 13 are large), so the “unit-of-analysis” problem could seriously skew our results.

Dealing with the unit-of-analysis problem in meta-analyses of small numbers of studies (as in our case) with multiple correlated metrics is an open problem in statistical methods research. We considered the recently-proposed correlated and hierarchical effects (CHE) model [73], but discovered that (for our data) it was highly sensitive to the choice of the rho parameter (which represents an assumption about how much variance there is between the different metrics in each study). Since we wanted to be conservative in our choice of statistical method, we chose the “brute force” aggregation approach suggested by [37], which trades statistical precision for simplicity and conservatism: it combines the correlation results of the various metrics in each study into a single estimate of correlation, which guarantees that no statistical power is derived from the presence of multiple metrics on the same population (even if such power might be warranted). Though it also has a rho parameter, the results for our data are insensitive to the choice of rho, with extremely high and low values of rho giving nearly-identical results. All meta-analyses in section 3 use rho = 0.6.

To perform the random-effects meta-analysis, we followed a standard procedure for data preparation and analysis [11]. First, we transformed Kendall’s τ values into Pearson’s r values [100]. Then, we transformed the r values to be approximately normally distributed, using Fisher’s scaling. Next, we normalized the signs of the individual metric correlations (i.e., the r values) so that a negative
correlation supports our hypothesis (the choice of negative is arbitrary; choosing positive leads to the same results with the opposite sign): we multiplied by -1 the correlation value for metrics where a positive correlation would support the hypothesis. This strategy has been used in other disciplines when combining different metrics whose signs have opposite interpretations, e.g., in [97]. We used R’s dmetar package (version 0.0.9000) to aggregate the correlations of the metrics from each study [37], and the R’s metafor package [99] (version 3.8-1) to run the meta-analysis and generate forest plots 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 heterogeneity).

To answer RQ3, we applied the same methodology as RQ2 for each individual tool’s warnings (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 for only the metrics in each metric category: time, correctness, rating, and physiological—i.e., we performed four meta-analyses, one for each metric group.

While we provide the p-values of all of these statistical analyses, we emphasize that they should be interpreted with caution given the relatively small sample sizes (and, for RQ1, 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 also used a meta-analysis (RQ2-RQ4) because interpreting the individual metric results (RQ1) to draw general conclusions for our hypothesis can be misleading [12]. Our meta-analysis also obviates the need for statistical correction to avoid multiple comparisons, such as Holm-Bonferroni’s [40]: 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 when 0.1 ≤ |r| < 0.3, etc. [20, 69, 78].

3 STUDY RESULTS AND DISCUSSION

We present and discuss the results of our study in this section. Scripts and data that generate these results are available in our replication package [27].

3.1 RQ1: Individual Correlation Results

Table 3 summarizes the results of each metric’s correlation (based on Kendall’s τ) with the total number of warnings from all tools. We provide descriptive statistics about these results, but we emphasize that these results should be interpreted with caution: “vote-counting” (e.g., checking the number of statistically-significant metrics in each direction) can lead to misleading conclusions [12]. We avoid directly drawing conclusions from these results and instead, we investigate the aggregated trend of all metrics with a meta-analysis in section 3.2.

Table 3 shows that for 13 of the 20 (65%) metrics, the direction of the correlation supports our hypothesis. For 4 metrics, there is no correlation, and for the remaining 3 metrics, the correlation is in the opposite direction than expected. Table 3 indicates the strength of the correlation in the rightmost column. Of the metrics where we found a medium or higher correlation, 8/8 are in the direction that supports our hypothesis. For the other 5 metrics that support our

Table 3: 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 (Exp. Cor.), if our hypothesis is correct, is either Positive or Negative. We assess r’s direction/strength, compared to the expected correlation (Exp?): ‘+’ means no correlation, ‘Y’/‘N’ means expected and measured correlations match (thus supporting our hypothesis), and ‘N’/‘N’ means they do not match. Capital letters in darker colors (Y/N) mean a medium or higher correlation. Lowercase letters and lighter colors (y/n) mean a small correlation. r’s significance is tested at the p < 0.05 (‘*) & p < 0.01 levels (**).

<table>
<thead>
<tr>
<th>DS</th>
<th>Metric</th>
<th>Type</th>
<th>Exp. Cor.</th>
<th>K’s τ</th>
<th>Exp?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>correct_output_rating</td>
<td>C</td>
<td>Negative</td>
<td>-0.34</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>output_difficulty</td>
<td>R</td>
<td>Negative</td>
<td>-0.43**</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>time_to_give_output</td>
<td>T</td>
<td>Positive</td>
<td>0.41**</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>brain_deact_31ant</td>
<td>P</td>
<td>Negative</td>
<td>-0.31</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>brain_deact_31post</td>
<td>P</td>
<td>Negative</td>
<td>-0.45</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>brain_deact_32</td>
<td>P</td>
<td>Negative</td>
<td>-0.38</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>time_to_understand</td>
<td>T</td>
<td>Positive</td>
<td>0.14</td>
<td>y</td>
</tr>
<tr>
<td>3</td>
<td>readability_level</td>
<td>R</td>
<td>Negative</td>
<td>-0.17*</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>binary_understand</td>
<td>R</td>
<td>Negative</td>
<td>0.01</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>correct_virt_questions</td>
<td>C</td>
<td>Negative</td>
<td>0.02</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>time_to_understand</td>
<td>T</td>
<td>Positive</td>
<td>0.05</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>gap_accuracy</td>
<td>C</td>
<td>Negative</td>
<td>-0.34</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>readability_level_before</td>
<td>R</td>
<td>Negative</td>
<td>0.08</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>time_to_read_complete</td>
<td>T</td>
<td>Positive</td>
<td>0.13*</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>brain_deact_31</td>
<td>P</td>
<td>Negative</td>
<td>-0.18</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>brain_deact_32</td>
<td>P</td>
<td>Negative</td>
<td>-0.18</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>complexity_level</td>
<td>R</td>
<td>Positive</td>
<td>0.35</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>perc_correct_out</td>
<td>C</td>
<td>Negative</td>
<td>-0.16</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>time_to_understand</td>
<td>T</td>
<td>Positive</td>
<td>-0.13</td>
<td>n</td>
</tr>
</tbody>
</table>

3.2 RQ2: Aggregate Correlation Results

Because direct interpretation of table 3 is difficult due to the different sample sizes of the various studies, we performed a meta-analysis to understand the overall trend (see fig. 1). As noted in section 2.6.2,
the results are presented by dataset rather than by metric to avoid unit-of-analysis errors [37].

The forest plot in fig. 1 displays the observed correlation (Person’s $r$ value, obtained from the Kendall’s $r$ value as described in section 2.6.2) and the 95% confidence interval (Estimate [95% CI]), as well as the estimated weight for each dataset (Weight). This information is shown numerically and graphically in fig. 1. Each box’s size is the dataset’s estimated weight (a larger box size means a larger weight), and the box’s middle point represents the correlation with respect to the dashed 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 our hypothesis). The horizontal lines visualize the confidence intervals for each dataset. At the bottom, the plot shows the aggregated correlation (on the right) and related information calculated by the meta-analysis. The diamond at the bottom of the plot visualizes the aggregated correlation; the width of the diamond represents the confidence interval.

Figure 1 shows a small aggregated correlation supporting an affirmative answer to RQ2 ($r = -0.23$, with a 95% CI that contains negligible, small, medium correlations: $r = -0.46$ to $r = 0.03$, $p = 0.07$). We interpret these results overall as support for the hypothesis that tool-based verifiability and humans’ ability to understand code are correlated to some extent.

The heterogeneity of the considered studies is non-negligible ($I^2 = (Q - df)/Q = (6.74 - 5)/6.74 = 25.8\%$ – not shown in fig. 1), indicating that 25.8% of the correlation variation (i.e., variance) we observe is due to the studies measuring different factors rather than due to chance. This result validates our choice of a random-effects model for the meta-analysis.

The plots in fig. 1 show wide confidence intervals for all the datasets except DS 3 and DS6, which indicates relatively high variability in the correlations. This indicates that most of these studies were under-powered for our purpose: the number of snippets considered was not high enough to give the meta-analysis much confidence in the correlation results. The meta-analysis correspondingly gives the largest weights to the two datasets with the most snippets: about 35% weight to DS3 (with 100 snippets) and about 25% weight to DS6 (with 50 snippets). Future work should explore running understandability experiments with larger numbers of snippets, which would enable us to gain further confidence in our results.

### 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. We report only the summary results for each tool (i.e., the results of the meta-analyses) for space reasons; forest plots similar to fig. 1 as well as the individual correlation results on each tool+metric combination are available in our replication package [27].

Repeating the meta-analysis on only the warnings produced by each tool individually gave similar results to the meta-analysis in fig. 1, except for Infer. The Checker Framework results support our hypothesis more strongly than the overall meta-analysis ($r = -0.26$, 95% CI of $[-0.44, -0.06], p = 0.02$). The results of OpenJML and JaTyC support the hypothesis more weakly than the overall results ($r = -0.12$, with a 95% CI of $[-0.29, 0.07], p = 0.16$ for OpenJML and $r = -0.17$ with a 95% CI of $[-0.39, 0.08], p = 0.14$ for JaTyC). Infer has too few warnings to draw meaningful conclusions from its results ($r = -0.09$ with a 95% CI of $[-0.94, 0.91], p = 0.60$).

From these results, we conclude that, while some tools support the hypothesis less strongly than the overall meta-analysis, all the tools but Infer show the same trend. These results support the overall meta-analysis results (RQ2): the correlation measured for 3 of the 4 studied tools suggests that the correlation between verifiability and understandability indeed exists (in small magnitude); no tool shows a markedly different trend except Infer, whose trend is not meaningful due to its small warning count.

We were also concerned that a single tool might be dominating the overall results. To mitigate this threat, we performed an ablation study by repeating the meta-analysis on warning data aggregated from each combination of three tools (i.e., excluding the warnings of one tool only). Overall, the results are extremely similar for each combination of tools to the overall results—the results without Infer are in fact nearly identical—with $r$ values ranging from $-0.23$ to $-0.20$; CI lowerbounds ranging from $-0.46$ to $-0.37$, and CI upperbounds ranging from $0.01$ to $0.06$; and $p$ values from 0.04 to 0.10. We conclude from this ablation experiment that no single tool dominates the RQ2 results.

Taken together, the results in this section show that the correlation found for RQ2 is not entirely driven by any tool: the overall results remain similar (if slightly weaker) for every tool individually except Infer and for each combination of three tools (i.e., without each tool). We interpret these results to mean that the correlation exists regardless of the specific verifier in use—meaning that our results apply to verification in general.
3.4 RQ4: Correlation Results by Metric Type

In section 3.1, we observed that the correctness and physiological metric categories appeared to support our hypothesis more strongly than the rating and time categories. To test this observation, we repeated our meta-analyses for each of the four metric categories.

The results refute the idea that these categories are a major influence on the results. The correctness, rating, and time metrics show overall results similar to fig. 1, but with wider confidence intervals: for correctness, $r = -0.28$ with 95% CI of $[-0.70, 0.27]$, $p = 0.20$; for rating, $r = -0.15$ with 95% CI of $[-0.54, 0.09]$, $p = 0.11$; and for time, $r = -0.22$ with 95% CI of $[-0.58, 0.21]$, $p = 0.23$. The results for the physiological metrics show that they have a minimal impact: $r = -0.32$ but with a huge 95% CI of $[-1.00, 0.98]$. These results, especially for the physiological metrics, are likely due to the smaller sample sizes created by considering only one metric type; e.g., there are physiological metrics in only two datasets (DS2 and DSF) with 28 total snippets between them. The dataset with the most weight in the overall results (DS3) only has rating metrics, which reduces the meta-analysis’ confidence in the other types. Finally, the heterogeneity for the three metric categories with useful results (i.e., not physiological) is higher than in the overall results (with $I^2$ of 57%, 49%, and 50% for correctness, rating, and time metrics, respectively).

3.5 Robustness Experiments

We ran additional 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 [13]. 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, 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. We see a similar phenomenon for the time metrics, but it occurs only for the good (rather than bad) comments. To explain this phenomenon, we compared the distribution of the metrics across comment categories and analyzed the scatter plots of the data used for correlation. Our analysis revealed that such disparity in correlation stems from a combination of (1) outliers found in the human measurements (likely due to data collection imprecisions in the original study [13]) and (2) the low number of data points in DS9. For example, we found that bad comment code was rated more readable by a few participants than code with no comments, even though the snippets were semantically the same. These unusual measurements led to outliers that had a considerable impact on the correlation results across comment categories (because of the small number of data points). The effect of this phenomenon on the overall results is low, because DS9 is given very low weight (6.34%, lowest among all datasets) by the meta-analysis due to its small sample size.

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 led 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 to handle 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 the maximum warning count in the dataset. All the results for RQ1-RQ4 were produced by following 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 and do not show any significant differences between the strategies for timeouts—the overall direction and strength of the correlations are similar, and the absolute size of the differences is small, meaning that the impact on the meta-analysis is negligible.

3.6 Results Discussion and Implications

3.6.1 Program Semantics and Understandability. Verification warning counts (indirectly) encode program semantics rather than syntactic properties of the code. Verification tools are trying to prove semantic properties: checking syntactic properties is decidable, so it is not the target of verifiers (which find approximate solutions
to undecidable, semantic problems, e.g., using SMT solvers). Our results suggest that there might be complexity caused by semantics, and verifiers are well suited to reasoning about that kind of complexity. Previous work using decidable syntactic metrics for complexity [3, 21, 41, 64, 78, 85, 89, 106] certainly could not capture semantics (since any non-trivial semantic property of a program is undecidable [74])—see section 5.

On one side, the measured correlation between verifiability and understandability increases our confidence that there is a semantic component to human code understanding. On the other side, the small correlation we measured indicates that there are other factors to code understanding beyond just program semantics. Neither of these conclusions are particularly surprising, but program understandability research has so far mostly focused on the non-semantic components (such as variable names or syntactic metrics—see section 5). Our work motivates the need for future studies that investigate the semantic component of code understanding; in particular, the specific semantic factors make code simple or complex, and how they impact understandability. Fortunately, our work also offers a path forward: the verification community has already built many tools that attempt to verify semantic properties (i.e., verifiers), which gives us an opportunity to leverage those existing tools to improve our understanding of code complexity and understandability.

### 3.6.2 Incorporating Verifiability into Comprehensibility Models
Most prior attempts to design automated metrics or models that measure or predict code understandability have used syntactic features that do not account for program semantics [3, 21, 41, 64, 78, 85, 89, 106]. Rather, they used syntactic features such as code branching, vocabulary size, and executions paths, among other proxies that attempt to capture code complexity (see section 5). Many of these features have shown to be poor predictors of code understandability [3, 26, 41, 45, 78]. We believe one of the reasons for this is because they do not capture complexity arising from program semantics. Based on the link we found between verifiability and comprehensibility, we hypothesize that semantic code properties would lead to more accurate models of understandability.

Future work should validate this hypothesis by incorporating verifiability into models that predict human-based comprehensibility [15, 78, 94] and measuring its impact on prediction performance. If the link between verifiability and comprehensibility exists (as our results suggest), verifiability information should complement the syntactic features of these models. Verifiability can be captured by adapting existing verification tools or by leveraging tool warning data. For example, we could provide the number of warnings a tool produces on a snippet as an input feature to these models.

### 3.6.3 Reducing False Positives to Increase Code Comprehensibility
Developers could use the warning count of verifiers to know when code might be complex, i.e., when it might need to be refactored to reduce accidental complexity. While coding, the developer can monitor the warning count of verifiers on a code snippet (e.g., a method they are writing or updating), knowing the code is correct. If this count increases, they could assess potential complex parts of the method and come up with changes to the method that would be semantically equivalent (e.g., replacing recursion, which is traditionally hard for verifiers to reason about, with a loop). This auxiliary benefit of using verification tools has not been studied in the literature, and might represent an opportunity to make verifiers more appealing to everyday developers.

This usage scenario poses a research opportunity too: what if we could automatically determine and suggest to the developer a semantically-equivalent refactoring that is easier to verify? Such a refactoring would change the code to perform the same task, but would cause a verifier to issue fewer warnings. The measured correlation between verifiability and understandability would mean, more often than not, that applying such a refactoring might make the code easier to understand. Since it is unclear if such refactoring is possible, more research should be conducted. However, if it is possible and the developer is aware of the correlation, we anticipate they would be more willing to (1) use verifiers in their everyday coding tasks, and (2) accept the refactoring suggestion. One possible issue with this approach is that our correlation includes warnings caused by missing specifications; there is large existing literature on specification inference (e.g., [22, 28, 96]) that could be leveraged to focus only on false positives when specifications are explicit.

### 3.6.4 Code Verifiability vs. Understandability vs. Complexity
Our study found a correlation between code understandability and verifiability, yet it did not find whether one of the two causes the other (i.e., correlation does not imply causation). Further research is needed to determine whether one causes the other, or whether there are other factors that cause both. However, based on our results and discussion, we hypothesize that code complexity causes both humans and verification tools to struggle to understand code. Future studies should investigate this and other possible causes.

## 4 LIMITATIONS AND THREATS TO VALIDITY

Our study shows a correlation between verifiability and understandability, but do not show one causes the other. So, our results must be interpreted carefully: further work is needed to determine causality.

Regarding threats to external validity, 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, as 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 (section 3.3), which demonstrated that no single tool dominates the results. The snippets are small compared to full programs; the comprehensibility of larger programs may differ. Further, 3/6 datasets are snippets from introductory CS courses rather than real-life programs, but this is mitigated by the other three datasets of open-source snippets.

Another threat is that subjects in the prior studies were mostly students. Only DS6 used professional software engineers (and only 13/63 participants—the other 50 were students), so our results may not apply to more experienced programmers. Future work should conduct understandability studies with professional engineers.

Beyond the datasets and tools, there are threats to internal and construct validity. We assumed the snippets are correct as written, and that each verifier warning therefore represents either a false positive or a specification that a human would need to write to verify the code. The presence of a bug would make a snippet seem “harder to verify” in our analysis (because every verifier would
warn about it), even if the snippet is easy for humans to understand, skewing the results. We mitigated this threat by manually examining a representative subset of the warnings as described in section 2.4; we did not observe any bugs in the snippets.

5 RELATED WORK

**Code complexity metrics.** Researchers have proposed many metrics for code complexity [3, 21, 41, 64, 78, 85, 106], 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 [64, 69]. These metrics are used to detect complex code so developers can simplify it during software evolution [4, 34, 69]. The motivation is that complex code is harder to understand [3, 78], which may have important repercussions on developer effort and software quality (e.g., bugs introduced due to misunderstood code). Our correlation results imply that code that is easier to verify might also be simpler and 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.** Scalabrin et al. [78] collected code understandability measurements from developers and students on 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). They found small correlations for only a few metrics, but a model trained on combinations of metrics performed better. Another study found similar results [94].

Researchers have explored the limitations of classical complexity metrics [3, 26, 41, 45, 78]. 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 [69]. Recent work has proposed new metrics such as Cognitive Complexity (COG) [18, 76], which assigns different weights to different code constructs. Muñoz et al. [62] 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. They did not take into account the unit-of-analysis problem in their meta-analyses.

We extend prior work with empirical evidence of the correlation between verifiability and human understandability. To the best of our knowledge, we are the first to investigate this empirically.

**Studying code understandability.** Researchers have studied code understandability and factors affecting it via controlled experiments and user studies [3, 13, 35, 41, 43, 69, 71, 82]. Precisely defining understandability is difficult, so some studies [13, 15, 62, 66, 72] use it interchangeably with readability (a different, yet related concept). Measurements include the time to read, understand, or complete code; the correctness of output given by the participants; perceived code complexity, readability or understandability; and (recently) physiological measures from fMRI scanners [69, 71, 82], biometrics sensors [31, 33, 105], or eye-tracking devices [1, 10, 31, 95].

Our study utilizes these human-based measurements of understandability to assess their correlation with verifiability.

Factors that affect understandability include: code constructs [3, 43] and patterns [13, 41, 52], identifier quality and style [83, 102], comments [13], information gathering tasks [10, 53, 82, 83], comprehension tools [87], code reading behavior [2, 70, 81], authorship [32], high-level comprehension strategies [81], programmer experience [102, 104], and the use of complexity metrics [103]. Our work investigates a new factor impacting 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.** A study conducted to evaluate a code readability model [15] is closely related to ours. The model was found to correlate moderately with snippets on which FindBugs [7] issued warnings. Unlike the tools in our study, FindBugs is not a verification tool (it uses heuristics to flag possibly-buggy code). We correlated verifiability with human understandability; the earlier study correlated FindBugs warnings with an automated readability model trained on human judgments. Though verification and static analysis tools are becoming more common in industry [8, 75], studies of their use and the challenges developers face in deploying them [8, 57, 63, 84, 98] suggest that false positives remain a problem in practice [42, 63]. 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 encourages developers to view false positives as opportunities to improve their code rather than as barriers to finding defects [77].

6 CONCLUSIONS AND FUTURE WORK

Our empirical study on the correlation between tool-based verifiability and human-based metrics of code understandability suggests there is a connection between whether a tool can verify a code snippet and how easy it is for a human to understand. Though our results are suggestive, our meta-analysis shows that extant studies on human code understandability lack sufficient power to enable us to draw a stronger conclusion, so more studies of understandability (preferably including many more snippets of code) are needed. Further, our work has shown only a correlation: establishing a causal link between verifiability and understandability—perhaps through a mutual cause, such as complexity—remains future work.

Verifiability is a promising alternative that complements traditional code complexity metrics, and future work could combine measures of tool-based verifiability with modern complexity metrics such as cognitive complexity that seem to capture different aspects of human understandability 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 identify potentially more complex code and make it more understandable by humans.

ACKNOWLEDGEMENTS
We thank the anonymous reviewers and Michael D. Ernst for comments that helped to improve this paper. Ji Meng Loh provided invaluable advice about our statistical approach.
REFERENCES


[38] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Necsan Amarat, Bor-Yuh Evan Chang, Samuel Z. Geyer, Udaya P. Khedker, Andreas
On the Relationship between Code Verifiability and Understandability

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA


