Towards Scalable Evaluation of Software Understanding: A Methodology Proposal

Florian Magin
Fraunhofer SIT | ATHENE
Darmstadt, Germany
florian.magin@sit.fraunhofer.de

Magdalena Wache
Fraunhofer SIT | ATHENE
Darmstadt, Germany
magdalena.wache@sit.fraunhofer.de

Fabian Scherf
Fraunhofer SIT | ATHENE
Darmstadt, Germany
fabian.william.scherf@sit.fraunhofer.de

Cléo Fischer
Fraunhofer SIT | ATHENE
Darmstadt, Germany
cleo.fischer@sit.fraunhofer.de

Jonas Zabel
Fraunhofer SIT | ATHENE
Darmstadt, Germany
jonas.zabel@sit.fraunhofer.de

Abstract

In reverse engineering our goal is to build systems that help people to understand software. However, the field has not converged on a way to measure software understanding. In this paper, we make the case that understanding should be measured via *performance on understanding-questions*. We propose a method for constructing understanding-questions and evaluating answers at scale. We conduct a case study in which we apply our method and compare Ghidra's default auto analysis with an analysis that supports binary constructs that are specific to Objective-C.

CCS Concepts

• **Security and privacy**;

Keywords

Decompilation, Evaluation, Understanding, Large Language Models

ACM Reference Format:
Florian Magin, Magdalena Wache, Fabian Scherf, Cléo Fischer, and Jonas Zabel. 2025. Towards Scalable Evaluation of Software Understanding: A Methodology Proposal. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3733822.3764672

1 Introduction

The majority of software deployed today is proprietary, so the people deploying it only have access to the compiled binaries, and not the original source code. This is a major problem for safety and security because those who deploy software need to understand the software to be aware of vulnerabilities and safety issues and make informed decisions. The field of reverse engineering seeks to build *software analysis systems* that allow users to answer questions about a software binary's behavior without requiring access to the source code. The most prominent tools are decompilers which attempt to produce an alternative source code for a binary.

One of the biggest challenges in reverse engineering is that measuring the performance of an analysis system is difficult. A fundamental obstacle is that there is no ground truth to compare against. For example, the "correct" output of a decompiler is undefined because the source code that produces a binary is not unique. Moreover, metrics that have commonly been used to evaluate software analysis systems are either easy to game, which makes them unsuitable for measuring progress, or too costly to implement. This absence of a good evaluation metric is an important problem because being able to evaluate current developments is vital for iterating on approaches and making progress.

In this paper, we propose a framework for using *understanding-questions* to evaluate software analysis systems: We consider an analysis better than another if it increases the ability to answer questions about the software because the ability to answer questions shows *understanding* of the software. As part of that framework, we develop a way to create understanding-questions at scale.

We formalize our experimental setup and make the experiments that are conducted based on our framework adaptable, transparent and reproducible. This experimental template goes beyond evaluating the impact of an analysis on software understanding, and also allows the investigation of other questions, such as comparing the ability to understand software between different LLMs or to potentially evaluate if an LLM's ability to understand software is predictive of human understanding.

Our main contributions are:

(1) A method for evaluating software understanding based on understanding-questions which is both: hard to game and practical (Section 4)
(2) A way to create and evaluate understanding-questions at scale (Section 5) and a case study (Section 6)

2 Background

2.1 Measuring Helpfulness for Answering Real-World Questions

In reverse engineering, our goal is to build better systems that *help people answer real-world questions related to the software they work with*, such as: "Given this large software package in my supply chain, how likely is it that because of this package sensitive information is leaked?" Using the terminology from the Software Understanding for National Security Roadmap [8] – we want to build systems that help *mission owners* answer *mission questions*. The most direct way to measure how helpful a new analysis system is for answering real-world questions is running a *controlled field experiment*. That means providing a group of people who are responsible for deploying software with an analysis system, and checking whether their performance on real-world questions improves over a control group of people who only have access to a baseline analysis system, e.g. a standard analysis tool. However, there are various practical problems with field experiments.

(1) *Noise*: There are many factors playing into the ability to answer real-world questions that do not have to do with the analysis system. Therefore, to make the measurement meaningful, the sample size needs to be very large.
(2) *Delayed results*: Many particularly relevant questions are about predicting the future, for example whether the software will be exploited in a certain way. For such questions, we only get results after a long time because we need to wait for the outcome to happen in order to have a ground truth. This issue may be circumvented by replacing prediction with *retrodiction* [14], which means asking experts to answer real-world questions about past versions of the software. Then the ground truth, e.g. whether a specific attack was successful, is immediately available. The problem with retrodiction that is that experts may already know the answers to some of the questions, which contaminates the data.
(3) *Randomization*: There may be confounders that influence the measurement result. For example, people that are willing to adopt a new analysis system may also be generally better at predicting outcomes, so they may outperform others even if the new analysis system is not useful at all. To make sure that confounders are eliminated, it needs to be randomized who uses the new system, and not based on their personal preference. However, it can be difficult to implement randomization in practice because some people may not be easily convinced to use a new analysis system.
(4) *Cost*: It is very expensive to run a study in practice with a reasonable sample size.

So in practice, a controlled field experiment is infeasible for most researchers. In particular, when developing a new analysis system, we need a metric that is quickly and cheaply available in order to enable a tight feedback loop for improving the system.

However, it is important to ensure that such a metric actually tracks understanding, and does not fall prey to *Goodhart's law*, which we explain in the following section.

2.2 Goodhart's Law and Gameable Metrics

"When a measure becomes a target, it ceases to be a good measure." This observation has been attributed to Charles Goodhart [9] and consequently, it is commonly called *Goodhart's Law* [4, 6]. When a metric is *gameable*, sometimes called "goodhartable", that means the metric can be optimized without optimizing the target that we actually want to optimize.

For example, the productivity of a programmer is correlated with how many lines of code they write, and therefore lines of code is a metric for productivity. However, a CEO who wants to incentivize productivity should not reward programmers for every line of code they write – which would turn lines of code into a *target* – because the programmers would just start to write many meaningless lines of code. That means lines of code is a *gameable* metric.

A metric is hard to game if an improvement of the target is *necessary* for an improvement of the metric. That means improved performance on the metric is *sufficient* to infer improved performance on the target.

In the example where programmers are rewarded for writing many lines of code, an improvement in the target "productivity" is *not necessary* for an improvement in the metric "longer code".

3 Related Work and Limitations of Existing Evaluation Approaches

In this section, we review existing evaluation methodologies. In summary: quality metrics, LLM-as-judge, and recompilation success are easy to game and therefore unsuitable for evaluating software analysis systems. Other approaches, like real-world impact (e.g. CVE-based evaluations) and semantic similarity are hard to game, but both face practical difficulties.

Quality Metrics and Recompilation Success are Easy to Game. There are various *quality metrics* for software analysis systems: Superficial metrics such as lines of code or number of goto commands [15], low cyclomatic complexity [13], asking users how readable they find the code [5], or asking LLMs (LLM-as-judge) [7].

To see why all these approaches are easy to game, consider an "analysis system" which always outputs `print "hello world"`. That output contains no goto commands, has a low cyclomatic complexity, and would be judged as very readable by both humans and LLMs, so it would score highly in those metrics. However, if the function that is analyzed does anything different from printing "hello world", then the output `print "hello world"` is not helpful at all. While in this particular example, the problem would be obvious to anyone using the analysis system, this kind of gaming could also be done in a more subtle way, for example by replacing meaningless parameter names with names that carry a false meaning, e.g. by replacing `param1` with `server_address` even though `param1` is not a server address.

Another metric that is commonly used is *recompilation success*, that means whether the decompiled code can be recompiled without errors [7]. However, this metric can easily be gamed as well, since `print "hello world"` also compiles without errors.

Real-World Impact and Semantic Similarity. The real-world impact of an analysis, such as finding new vulnerabilities that are assigned CVEs, is hard to game, since software understanding is a necessary condition to find vulnerabilities. This holds also for a semantic similarity metric: The idea of similarity-based approaches is that we compare how similar the decompiled code is to the original source code. If an analysis system produces an output that is semantically similar to the original source code, that is sufficient for understanding (assuming that the original source code provides significantly more understanding than the binary, which is usually the case), so similarity-based approaches are not easy to game.

These two metrics are reasonable but come with practical problems. Real-world impact is a very good metric, but it can only be used in rare cases. For example, an analysis that improves software understanding in malware analysis, e.g. understanding the behavior of malware, will never lead to a CVE.

For using semantic similarity to the original source code as a metric, we first need a similarity metric for code. One approach that naturally comes to mind is using a code embedding such as GraphCodeBert [11], and computing the distance in the embedding space. The distance in embedding space has been shown to represent semantic meaning [2]. However, state-of-the-art embedding models are uninterpretable and may consider irrelevant properties such as variable name formatting, e.g. `CamelCase` vs `snake_case`.

Because of these difficulties, we propose a an evaluation framework based on *understanding-questions* , which is practical, hard to game, and intuitively captures the understanding that is necessary to answer real-world questions. We introduce our framework in the following section.

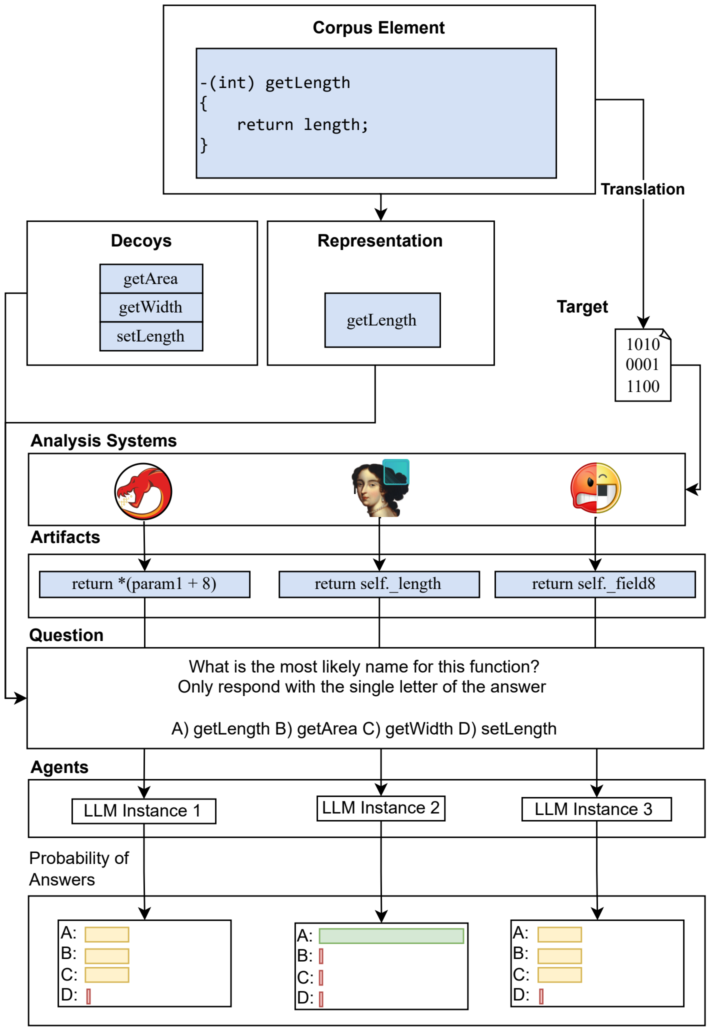
4 The Evaluation Framework

In this section, we present our evaluation framework for measuring software understanding. Unlike the existing approaches we reviewed in Section 3, our approach is not gameable and it is practical to implement at scale. Our framework is based on the idea that we can quantify the understanding which an analysis system enables by providing the output of the analysis system to an *agent* (e.g. an LLM, or a person), and then testing how well the agent performs on answering understanding-questions. If the agent's performance when answering the questions is significantly improved after changing the analysis while keeping all other experiment parameters constant, then the analysis was the cause for the improvement. Importantly, since understanding-questions can have substantial variation in their difficulty, understanding-question performance is meaningless in isolation. Therefore, it is crucial to always measure the performance of an analysis in comparison to at least one other baseline analysis. In the following, we introduce the components of our framework, as depicted in Figure 1.

Corpus. We call the data on which the evaluation is performed the *corpus*. For example, a corpus could be a collection of open-source projects, gathered from GitHub or the language's package index. The corpus consists of *corpus elements*, which determine the unit on which we want to measure understanding. For our case study in Section 6 we use the CocoaPods[1] repository, which is the package index for Objective-C and Swift, as the corpus. The corpus elements are functions. Other options for corpus elements would be compilation units or even whole projects.

Representation. After choosing a corpus, we extract a *representation* from each element. The representation is information which captures a property of the corpus element, in a legible way. The representation can be a short string, but it can also be a more complex attribute like a human-readable description of the relevant semantics of the corpus element. In our case-study, we choose the function name, e.g. `getLength`. The representation can be derived deterministically via a conventional algorithm, but it could also be derived non-deterministically, e.g. via an LLM.

Translation from a Corpus Element to a Target. We call the transformation that is applied to the corpus element, which hinders



```
-(int) getLength
{
    return length;
}
```

What is the most likely name for this function?
Only respond with the single letter of the answer

A) getLength B) getArea C) getWidth D) setLength

Figure 1: Overview of our methodology: We evaluate two analysis systems against a baseline system. The target is analyzed by each of the analysis systems each producing pseudocode as their artifact. We provide the artifacts to different instances of an LLM, and measure the LLM's performance on multiple-choice understanding-questions

understanding by deleting or obfuscating information, the *translation*. We evaluate systems that aim to revert the translation and recover the lost information. The output of the translation is called the *target* which is analyzed by the analysis system. The canonical example for a translation is a compiler which turns source code into a compiled target, but the idea is much more general: The translation could be a process that strips symbols, or a string obfuscation or an obfuscation that adds superfluous information like opaque predicates. In our case study, we use the official IDE for Objective-C, Xcode, for the translation.

Analysis System and Artifact. The target is analyzed using an *analysis system*. We call the output of the analysis system *artifact*. The typical example is a decompiler which turns a binary target into a pseudo-code artifact or some other high-level intermediate language. The analysis system is not necessarily a single analysis tool. Rather, it could also be a composition of analyses, which produce intermediary artifacts. The analysis system is also not required to produce a deterministic output, so for example, it can involve an LLM-based decompiler or an LLM summary.

Agent, Understanding-Questions and Decoys. To evaluate an analysis system, we construct *understanding-questions* based on the corpus which are then posed to an *agent*. The agents could be humans or multiple instances of an LLM. The agent gets access to either the baseline, or the new analysis system, and we measure how much the agent's ability to answer the questions increases if it has access to the new analysis. To enable scalable question creation and evaluation, we use multiple-choice questions. That means, we ask the agent to discern between the representation of the corpus element of the provided artifact and wrong answers, called *decoys*. The questions are created via a simple string template, for an example see appendix A.

Example. In Figure 1, we show an example for one way to use our framework. We chose the function name `getLength` as the representation. We compare two new analyses with a baseline analysis. The first analysis outputs the pseudo-code `return self._field8` as the artifact, the second outputs `return self._length`, and the baseline analysis outputs the artifact `return *(param1 + 8)`. To evaluate if the new analyses improve understanding, we give an agent access to the artifact and ask it which of the names A) `getLength`, B) `getArea`, C) `getWidth`, or D) `setLength` is the most likely name for the analyzed function.

Gaming Resilience. To ensure that the evaluation is not gameable, we require that in any evaluation:

(1) The performance is measured as the improvement *relative to a baseline analysis*.
(2) The process of choosing the experiment parameters, especially the corpus and process of deriving representations, is reported in detail.

These two requirements prevent two different types of gaming:

Firstly, one might choose the decoys in such a way that the understanding-questions become very easy. However, in that case the baseline also has a very good performance, the performance *improvement relative to the baseline* is small.

Secondly, one might retroactively restrict the corpus to preferentially include elements for which the agent using the new analysis outperformed the agent using the baseline analysis. However, that result is still an interesting contribution: It demonstrates for which specific corpus subset the analysis improves the performance. There is only a problem if evaluators claim that the subset of the corpus is representative for the whole corpus. However, this type of gaming is made more difficult by the requirement that evaluators report in detail how they chose the corpus.

5 Question Creation at Scale

The creation of understanding-questions is a central part of our framework because it determines *what type of understanding* is evaluated. In this section, we discuss which kind of understanding-questions are possible, and provide various examples of question creation methods (Section 5.1), methods for sampling decoys (Section 5.2), as well as a method to extend the corpus (Section 5.3).

5.1 Question Creation

We can create different kinds of understanding-questions using the general idea of creating a multiple-choice question in which the agent needs to recognize the representation belonging to target which was translated from the corpus element. What kind of understanding we evaluate depends both on the choice of the *representation* which represents understanding, and on the choice of *translation* which hinders understanding. The abstract understanding-question is in every case a simple template, where only the artifact of a fixed corpus element (e.g. a decompiled function body) is presented and the agent is asked to decide what its representation (e.g. the function name) is out of a set of answers.

We illustrate this idea with some examples for representations and translations that give rise to meaningful tasks.

5.1.1 Original Source Recognition. The most basic version of a recognition question can be constructed by choosing the identity function as the representation, and using a regular compiler as the translation, so the agent's task is to recognize the original source code. The limitation of this approach is that the original source code can contain too much specific information that is not related to interesting kinds of understanding. If an artifact involves the number 6a09e667 and the source code also contains this number, then the agent can reasonably assume that the artifact and the source code probably match because they seem to involve the same magic number. However it would be better if the agent understood that this number is related to SHA256, so matching against the description "computes the SHA256 hash of some data" or matching the artifact to the function name `computeHash` would show a more meaningful understanding.

5.1.2 Function Name Recognition. One way to address the limitation of original source recognition is to use the function name as the representation. In this case, the translation is still a compiler. However, the function name might be too short and not meaningful enough, but that is not necessarily a problem as the overall experiment process measures the performance *improvement relative to a baseline*. Even if the understanding that the new analysis provided only helps by ruling out a few function names that are definitely wrong, that still increases the chance of correctly guessing from the left-over set, and there is a measurable performance improvement. In our case study in Section 6, we use this way of generating questions, and measure a significant performance improvement.

5.1.3 LLM-Generated Description Recognition. We may choose LLM-generated descriptions of the original source code as the representation, and ask the agent to distinguish the description of the original code from descriptions of other corpus elements. Descriptions are a meaningful representation which carries more semantic content than function names, while avoiding the pattern-matching failure mode of original source recognition.

5.1.4 App Store Description Recognition. We may choose the translation to be a simple stripping of metadata instead of a compiler. For example, if a corpus consists of closed-source apps in an app store, the translation could be to strip away the app store descriptions and other metadata. We can then choose the description as the representation, and the agent has to recognize which description belongs to the executable code of a particular app. This example shows that our approach does not necessarily require access to the original source code.

5.1.5 Runtime-Behavior Recognition. The idea of runtime-behavior recognition is to evaluate the quality of a static analysis by measuring an agent's performance in predicting the results of a dynamic analysis. In this case, each corpus element consists of executable code plus the environment on which we run the dynamic analysis. The translation consists of removing the environment, so the target consists of only the executable code. The representation is the output of the dynamic analysis. This representation is easy to generate using existing tooling, and an increase in the ability to predict runtime-behavior is a meaningful sign for software understanding.

5.2 Decoy Sampling

To construct a multiple choice understanding-question, in addition to the correct answer, we also need to present answers that are wrong. We call these wrong answers "decoys". In this section we elaborate on how we choose the decoys.

Decoy Sampling. We talk about "decoy sampling" rather than "decoy generation", because decoys should always be sampled from a set of representations such that every wrong answer in a multiple-choice question is the correct answer to a different question. This is important because otherwise the evaluation becomes gameable. For example, one might develop a new analysis which produces pseudo-code in `snake_case`, and choose the decoys to always use `CamelCase`. Then, the agent with access to the new analysis is more likely to choose the correct answer than the baseline even if the analysis does not provide any additional understanding.

In particular, using LLMs to create decoys is not allowed because it is possible to let the LLM encode some marker into the decoy that only the new analysis can decode.

Choosing a Representation Set to Sample from. To sample the decoys, we need to specify a set we sample them from. Which set to sample from is an important choice because it is central to the kind of understanding we evaluate. For example, for function name recognition, if the decoys are sampled from all possible strings within some length constraints, then most agents will easily decide that `createWebView` is the most likely function name when the alternative is a random-looking string such as `mFQeVfxtP`, so the understanding that is measured is relatively basic.

If we want to measure a more sophisticated understanding than the understanding of what a function name looks like in general, we can instead choose the set of all representations in the corpus, i.e. all representations that are the correct answer for at least one question. This tests whether the agent understands enough to distinguish between different function names within the corpus.

However, the understanding that is needed to do this distinction well may still be relatively basic. For example, in Objective-C, the number of parameters is encoded in the function name, so the number of parameters is sufficient for the agent to exclude many incorrect function names. If we want to measure understanding beyond the ability to mach the arity, we can sample the decoys from the set of all function names with the same number of parameters. In our case study in Section 6, that is how we sample the decoys.

In general, the evaluator is responsible to find a choice of decoys that demonstrates the kind of understanding that a novel analysis provides. Even if an evaluator deliberately searches for a set of decoys that confuses the baseline analysis, but not the novel analysis, that result would still be an interesting contribution.

5.3 Extending the Corpus

As mentioned in Section 5.2, we do not allow LLM-generated decoys because of the potential for gaming. However, we do allow generating new *corpus elements* and add them to the corpus. Mutated corpus elements can be very useful because they allow us to test for more specific types of understanding. For example, we can use LLMs (or more traditional methods such as LAVA [3]) for automatically introducing vulnerabilities to the corpus elements. We can then add these modified elements to the corpus and use the introduced vulnerability as the representation. If an agent using an analysis system can recognize vulnerabilities, that demonstrates understanding about these specific vulnerabilties.

6 Case Study: Function Name Discernment for Objective-C

This case study serves as a concrete illustration of our presented methodology and shows that it is indeed an intuitive method to quantify the effect of certain choices on software understanding. We measure the performance improvement of the following task: An LLM agent is provided with decompiled code of a function and, based on this code, should decide the function's name from a provided pool of possible names, approximating that the agent possesses an understanding of the decompiled code. We compare the standard Ghidra analysis with a recent approach from the literature which enables IFDS dataflow analysis on Objective-C binaries [12]. We run our experiments with two different open-source LLMs – Deepseek V3 0324 and Devstral 24b. We do not use closed-weight models like models of the Claude or GPT family, as we could not set them up on our research cluster. Alternatively, this experiment could be instantiated with humans as the agent, if we wanted to measure the impact of the analysis tool on their ability to understand software, or to compare their improvements to the benefits an LLM gets from the analysis tool. As our corpus is limited and not rigorously statistically chosen, this study can only be considered a pilot study which illustrates the overall methodology.

6.1 Experiment Design Rationale

For setting up an experiment we need to make choices for the parameters in Section 4: corpus, translation, representation, analysis system, understanding-questions, agent, and decoys.

Corpus. We choose the CocoaPods repositories collection as our corpus, as this is one of the few large collections of Objective-C source code that are available. The assumption that comes with this choice of corpus is that the kind of understanding that we are looking to demonstrate generalizes from library code to application code. This assumption could later be reinforced or refuted by other experiments on a different corpus with a different task. We manually selected ten popular CocoaPods repositories, based on both the number of GitHub stars and the percentage of Objective-C source code. The projects include a total of 59021 lines of Objective-C code, counted without comments. *We chose the simple corpus for demonstrating our methodology, but do not claim that the improvement of the analysis is representative* – this would require to use a more representative corpus such as the whole set of CocoaPods libraries.

Translation. We use the regular XCode compiler to translate the library source code into the binary or multiple binaries, and remove the method names from the binaries. The translation resulted in a total of 4503 different Objective-C methods.

Analysis Systems. We compare two analysis systems. The first is Ghidra's default auto analysis, which outputs a decompiled function body (pseudo-C-code) and a function signature for every function in each binary. The second analysis is an improved auto analysis supporting Objective-C specific binary constructs such as resolving simple dynamic dispatches, creating object layouts from metadata and rewriting automated reference counting method calls [12].

Agent. As agents, we use Deepseek V3 0324 and Devstral 24b, two open-weight LLMs. We measure the performance for each combination of agent and analysis. We prompt the LLMs to respond with a single token corresponding to one multiple-choice option. These two LLMs differ in their parameter count by over one order of magnitude (685 billion vs 24 billion), which allows us to sample the effect of the analysis on software understanding across model sizes.

Representation. The representation is simply the method name of a function from the original source-code.

Understanding-Question and Decoy Creation. The agent is asked to identify the correct function name from a given set of seven possible answers (six decoys). The agent has access to the decompiled code of the function, as well as the function signature, that is the number and types of parameters recovered by the analysis. The decoy names are sampled from the set of function names with the same arity across all analyzed binaries of all projects, removing duplicates.

6.2 Experimental Results

Based on a sample size of 10000 understanding-questions[1], we measured the performance of the two agents as shown in Table 1. The performance improvement is approximately 17.5 percentage points for DeepSeek V3, and approximately 19.1 percentage points for Devstral 24b.

We handled two special cases during the evaluation. Even though the LLM agent was prompted to only respond with a single token corresponding to its chosen option, sometimes the agent started reasoning, which resulted in an invalid token response, e.g. *"Based" [on the provided . . .]* instead of the number corresponding to the agent's chosen option.

In this case, we gave the agent the exact same task with the same options in the same randomized order as before until the agent responded with a valid token, which was mostly the case after a single retry. This procedure was counted as a single answer. We chose to let the agent retry instead of ignoring the question, since the reasoning could happen for example in the more difficult cases, so ignoring the questions could bias the sample.

We sampled the decoys randomly from the equivalence class of function names with the same number of input parameters, so for a few functions with many input parameters, the equivalence class was smaller than seven, so we could not create a task with seven possible answers. We excluded these cases from our analysis for this demonstration of the methodology. However, functions with a high number of arguments could tend to have a larger function body and be more complex, making it harder for the agent to understand the function. Hence, ignoring these functions could bias the sample. Even though we expect the impact to be minor, a treatment of this case would be desirable for a proper evaluation.

[1]A question includes the specific decoys, which permits more questions than methods in the corpus

Table 1: Results for the DeepSeek V3 and Devstral 24b LLMs. Both agent's performance increases if they have access to the improved analysis.

	Default Analysis	Improved Analysis
DeepSeek V3		
Correct	77.52%	94.96%
Incorrect	22.32%	4.96%
Ignored	0.16%	0.08%
Devstral 24b		
Correct	56.58%	75.67%
Incorrect	43.31%	24.19%
Ignored	0.11%	0.14%

6.3 Discussion of Experimental Results

The experimental results display two clear patterns of note. The straight-forward one, which was the initial question of the experiment, is that the improved analysis allows correct recognition of around 20 percentage points more functions compared to the default analysis case for both LLMs. In our terminology, this could be summarized as: The analysis that was evaluated improves the ability of LLMs to understand software, as approximated by their ability to predict function names.

The secondary result is that we now have preliminary data that DeepSeek V3 has a measurably better ability to understand software compared to Devstral 24b, as approximated by function name prediction.

7 Conclusion and Future Work

For measuring software understanding in a way that is both hard to game and scalable, we propose a methodology based on multiple choice understanding-questions. A central part of the methodology is the choice of questions and decoys. In our case study, we use function name recognition as the question, and function names with the same number of parameters as the decoys, and we find that in the context of Objective-C functions, function name recognition already provides a large signal (17.5 and 19.1 percentage points performance improvement between the two analysis systems) – even though function name recognition is a relatively simple task. In future work, we are excited about similar case studies in different domains. In particular, we believe that it is valuable to not just report positive results such as ours, but also report cases when there is very little performance improvement, or even a decrease in performance on understanding-question relative to a baseline analysis. Making the field aware of negative results helps researchers avoid wasting effort on approaches that are not promising.

Furthermore, we believe that future work which connects our ideas to the cognitive science literature is promising, since there is a rich body of work investigating certain types of decoys, also called "foils", "distractors", or "traps" and their effects on human test subjects. We believe that this literature should be reviewed under the lens of reverse engineering in order to gain a deeper understanding of which questions are best suited for measuring which type of understanding. In particular, we believe that a useful way to think about understanding in general is *Bayesian inference* [10].

Framing reverse engineering as *software understanding* clarifies our goal and enables us to apply insights from cognitive science to evaluate progress in reverse engineering.

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] [n. d.]. CocoaPods Project. https://cocoapods.org/
[2] Zimin Chen and Martin Monperrus. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).
[3] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. doi:10.1109/SP.2016.15 ISSN: 2375-1207.
[4] Marc A Edwards and Siddhartha Roy. 2017. Academic research in the 21st century: Maintaining scientific integrity in a climate of perverse incentives and hypercompetition. *Environmental engineering science* 34, 1 (2017), 51–61.
[5] Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. 2023. dewolf: Improving Decompilation by leveraging User Surveys. In *Proceedings 2023 Workshop on Binary Analysis Research*. Internet Society, San Diego, CA, USA. doi:10.14722/bar.2023.23001
[6] Michael Fire and Carlos Guestrin. 2019. Over-optimization of academic publishing metrics: observing Goodhart's Law in action. *GigaScience* 8, 6 (2019), giz053.
[7] Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Bolun Zhang, and Chao Zhang. 2025. DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios. doi:10.48550/arXiv.2505.11340 arXiv:2505.11340 [cs].
[8] Douglas Ghormley, Tod Amon, Christopher Harrison, and Tim Loffredo. 2024. Software Understanding for National Security (SUNS). (2024), 65.
[9] Charles AE Goodhart and CAE Goodhart. 1984. *Problems of monetary management: the UK experience.* Springer.
[10] Thomas L. Griffiths, Nick Chater, and Joshua B. Tenenbaum. 2024. *Bayesian models of cognition: reverse engineering the mind.* The MIT Press, Cambridge, Massachusetts.
[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[12] Florian Magin, Gwendal Patat, and Fabian Scherf. 2025. Heros in Action: Analyzing Objective-C Binaries through Decompilation and IFDS . In *2025 IEEE/ACM 1st International Workshop on Advancing Static Analysis for Researchers and Industry Practitioners in Software Engineering (STATIC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–6. doi:10.1109/STATIC66697.2025.00005
[13] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.
[14] Kevin Smith and Edward Vul. 2014. Looking forwards and backwards: Similarities and differences in prediction and retrodiction. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 36.
[15] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. doi:10.14722/ndss.2015.23185

A An Example Understanding-Question

To illustrate the question generation and provide an example of an understanding-question as used in the case study, we provide the used prompt template of our case-study.

```
I have the following task for you.

Analyze the following decompiled code of a
    certain function.

Before the function body, I will also
    provide the typed arguments of the
    function in the usual convention in
    round brackets.

Based on the code, decide the function's
    name from one of the listed options,
    enumerated and indexed by numbers below.

Please respond with no reasoning or text,
    only with a single number that refers to
    the option you chose.

Decompiled Code:
{function["decompiledCode"]}

Enumerated Options:
{enumerated_options}

You have to provide an answer, so a simple
    space is not valid.

Remember to make sure that your answer is a
    single number, no text. So if you choose
    say option 5, your answer should be
    only: "5", without any apostrophes.
```