StackSight: Unveiling WebAssembly through Large Language Models and Neurosymbolic Chain-of-Thought Decompilation

Weike Fang [1] **Zhejian Zhou** [1] **Junzhou He** [1] **Weihang Wang** [1]

Abstract

WebAssembly enables near-native execution in web applications and is increasingly adopted for tasks that demand high performance and robust security. However, its assembly-like syntax, implicit stack machine, and low-level data types make it extremely difficult for human developers to understand, spurring the need for effective WebAssembly reverse engineering techniques. In this paper, we propose StackSight, a novel neurosymbolic approach that combines Large Language Models (LLMs) with advanced program analysis to decompile complex WebAssembly code into readable C++ snippets. StackSight visualizes and tracks virtual stack alterations via a static analysis algorithm and then applies chain-of-thought prompting to harness LLM's complex reasoning capabilities. Evaluation results show that StackSight significantly improves WebAssembly decompilation. Our user study also demonstrates that code snippets generated by StackSight have significantly higher win rates and enable a better grasp of code semantics.

1. Introduction

WebAssembly (or WASM) is a low-level, portable bytecode language designed for high-performance computations at near-native speeds and broad portability across devices and platforms. It was first introduced for web applications (Haas et al., 2017a) and is now supported by all major browsers, including Chrome, Firefox, Safari, and Edge (McConnell, 2017). It was then applied to a wide range of applications such as mobile devices (Pop et al., 2022), smart contracts (McCallum, 2019), and Internet of Things (Gurdeep Singh & Scholliers, 2019; Liu et al., 2021).

[1]Department of Computer Science, University of Southern California, Los Angeles, CA, United States. Correspondence to: Weihang Wang <weihangw@usc.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

Facilitating WebAssembly understanding is crucial, given its exploitation for malicious purposes such as cryptojacking, where it is executed secretly in browsers to mine cryptocurrencies (Konoth et al., 2018; Kharraz et al., 2019; Musch et al., 2019b; Romano et al., 2020). While WebAssembly serves as a compilation target for high-level languages such as C, C++, Go, and Rust, challenges arise when WebAssembly code is shipped as third-party modules without access to high-level source codes (Musch et al., 2019a; Romano & Wang, 2023). Despite adopting a text format equivalent to WASM binary code, manual comprehension remains challenging. Different from register-based native binaries, WebAssembly manages a virtual stack machine, necessitating the tracking of stack behaviors to comprehend its operations. Furthermore, WebAssembly employs only four numeric data types (`i32`, `i64`, `f32`, and `f64`), concealing variable types and semantics. To make matters worse, 28.8% of WebAssembly binaries are minified (Hilbig et al., 2021) with the variable names obfuscated and web test suites can get flaky (Liu et al., 2024), making it hard to interpret and test. Such challenges motivate *automatic decompilation* for better understanding.

Although *decompilation* for WebAssembly could be naturally formulated as an end-to-end sequence translation task that takes assembly code as input and outputs the respective C++ source code, we find this naive formulation unsatisfactory in terms of performance and insight. The dissatisfaction arises from the relative scarcity of WebAssembly pretraining corpus and the fuzziness of neural networks.

In light of the trend in Natural Language Processing that encourages intermediate reasoning steps, we approach the decompilation problem in a step-by-step fashion. We identify intermediate tasks humans would perform in order to decompile WebAssembly code. In correspondence with WebAssembly specifications, we propose three key stages: (1) explicitly tracking the virtual stack, (2) recovering semantically meaningful names of variables, and (3) summarizing code functionality in natural language.

In this work, we propose *StackSight*, the first approach utilizing LLM with advanced program analysis for WebAssembly decompilation. StackSight breaks decompilation down into multiple smaller tasks: the program analysis component

tracks the virtual stack as additional contextual information, and the chain-of-thought prompting ensures that the model can progressively interpret binary codes. This dual approach aids in accurately predicting variable semantics, deciphering function objectives, and effectively decompiling WebAssembly code. Besides, the necessity for such an advanced methodology arises partly from the scarcity of WebAssembly data in the datasets typically used to pretrain large language models, making them fail to grasp the nuances of this assembly-level language. We demonstrate that without StackSight, large language models perform poorly on WebAssembly comprehension and decompilation.

Existing literature has extensively demonstrated the capabilities of LLMs in various code reasoning tasks. However, our research uniquely directs its focus towards binary decompilation, which remains largely understudied within the community (Fan et al., 2023). Beyond these primary tasks, our approach can be extended to downstream tasks such as malware detection and code review.

Specifically, our work makes the following contributions:

- We develop a robust static analysis tool that explicitly tracks the state of the virtual stack, accounting for complex control and data flows. Equipped with the tool, we propose a novel Chain-of-Thought pipeline catered for WebAssembly decompilation.
- We perform comprehensive experiments on StackSight. Results show that StackSight increases the amount of functionally correct decompiled codes by 70% and produces code that is notably more favorable from the perspective of human developers.
- Case studies show that the static analysis tool mitigates logical errors and hallucinations in LLM outputs.

2. Related Work

2.1. WebAssembly Decompilation

In the realm of WebAssembly decompilation, significant efforts have been made towards recovering data and function types. Lehmann and Pradel leveraged LSTM neural networks to recover precise, high-level parameter and return data types for WebAssembly functions (2022). Romano and Wang generated semantics-aware intermediate representations of WebAssembly functions and applied machine learning classifiers to understand module and function purposes (2023). These methods can significantly enhance human understanding of WebAssembly code, yet they fall short in unveiling the variety of code functionalities or enabling decompilation into high-level C/C++ codes.

Benali was among the first to train a WebAssembly decompiler based on transformer and LSTM models (2022). However, their method only works on artificial, small code snippets designed to evaluate different C grammars, lacking in real-world semantic complexity. Our approach, on the other hand, can work for more complicated WebAssembly code, identify diverse purposes, and decompile them to near-equivalent C/C++ code. This represents a significant leap forward in WebAssembly decompilation.

2.2. Decompilation Using Static Analysis

Researchers have also utilized traditional static analysis approaches to decompile binaries to C/C++ (Fokin et al., 2011; Yakdan et al., 2015; Wang et al., 2017), Java (Desnos & Gueguen, 2011; Harrand et al., 2020), and Python (Ahad et al., 2023). A large body of work have focused on recovering Control Flow Graph (CFG) (Cifuentes, 1993; Cifuentes & Gough, 1995; Fokin et al., 2011; Yakdan et al., 2015; Wang et al., 2015; 2017; Gussoni et al., 2020) and type inference from binary code (Lee et al., 2011; ElWazeer et al., 2013; Noonan et al., 2016; Xu et al., 2017; Lehmann & Pradel, 2022). Liu and Wang performed a comprehensive study to investigate decompilation correctness of C code decompilers, finding that while modern decompilers have been progressively improved, challenges such as type recovery and optimization still hinder well-formed outputs (2020). Our approach distinctively integrates static analysis with a focus on the virtual stack machine to infer variable types and relationships, thus tackling the inherent complexities of WebAssembly. Unlike traditional static analysis methods that may not fully consider WebAssembly's implicit memory mechanisms, our approach provides a detailed understanding of these elements to reconstruct high-level data types and semantics from the binary code.

2.3. Neural Decompilation

Prior work has leveraged Neural Machine Translation (NMT) models for binary decompilation (Katz et al., 2019; Liang et al., 2021). Katz *et al*. presented a Recurrent Neural Network (RNN) based method to decompile machine code to high-level C code (Katz et al., 2018; Fu et al., 2019). Cao *et al*. targeted compiler-optimized binaries and leveraged graph neural network (GNN) model to convert binaries to an intermediate representation (IR), which is then used to generate high-level code (2022).

Recently, Large Language Models have shown impressive performance on code-related tasks, particularly in binary reverse engineering (Pearce et al., 2022; Xu et al., 2023; Wong et al., 2023; Al-Kaswan et al., 2023). All these efforts aim to decompile C/C++ executable binaries back to C/C++. Our work stands at the intersection of these neural decompilation methods and the unique challenges presented by WebAssembly binaries. This focus on WebAssembly sets our work apart, as we address the specific nuances and

complexities inherent in these binaries, contributing novel insights and methodologies to the field of decompilation.

3. Methodology

3.1. Motivations behind Step-By-Step Decompilation

In our methodology, we draw inspiration from recent advancements in eliciting reasoning abilities in LLMs. One major advancement in the NLP community is the Chain-of-Thought (CoT) Prompting approach, first introduced by (Wei et al., 2022b). CoT prompting focuses on guiding large language models through a series of intermediate reasoning steps resembling the thinking process of a human, significantly improving the ability of LLMs to perform complicated reasoning.

Our work is the first to propose using CoT reasoning in reverse engineering tasks like decompilation. Previous work (Benali, 2022) has shown that a direct sequence-to-sequence model from WebAssembly binaries to high-level C code fails to grasp such complex data flow and generate high-quality source code when code complexity increases. It is intuitive to decompose the decompilation process into multiple phases as a large amount of complex reasoning is needed to interpret, abstract, and translate binary code to human-readable languages. We reflect on how humans would approach the WebAssembly decompilation task and strategically decompose the process into multiple steps, eliciting LLMs' reasoning abilities to decompile like a human. We identify the following three intuitions that are critical to unveil the assembly-like nature of WebAssembly code.

Explicitly Tracking the Virtual Stack As specified in the original paper (Haas et al., 2017b), WebAssembly code operates on an implicit stack. An example WebAssembly Text Format file is provided in Listing 1.

As shown, WebAssembly code loads variables onto the stack (`local.get, i32.const`), operates on the top stack elements and writes back the answer (`i32.add, i32.mul , i32.div_s`). However, the text format lacks explicit representation tracking the history states of the stack.

While this approach can achieve a more compact code size, it also imposes great difficulties for humans to understand the code. We concur that keeping an explicit history of what is on the virtual stack and the operations that interact with the stack would serve as a useful intermediate step. More than helping to identify the semantics of WebAssembly code, this step also reveals patterns in `.wat` files that correspond to specific behavior in C++ source code.

```
(func (;1;) (type 1) (param i32) (result
    i32)
    local.get 0
    i32.const 1
    i32.add
    local.get 0
    i32.mul
    i32.const 2
    i32.div_s)
```

Listing 1. Example of WebAssembly Text file (WAT).

Recovering Semantically Meaningful Names of Variables Also specified in the original paper (Haas et al., 2017b), WebAssembly code only assigns indexes to used variables, posing another major obstacle to understanding WebAssembly code. Human developers typically encode semantic information in variable names. To recover variable semantics in WebAssembly code, we first prompt the LLM to predict the variable types of input and return values based on stack information. Then, the model gradually identifies the purpose of each local variable by inspecting operation histories and assigns meaningful names to each variable.

Summarize in Natural Language Once all variable semantics are recovered, the next logical step for human developers to interpret the WebAssembly code is to identify high-level purposes of the code. This is crucial in preventing the verbatim transcription of WebAssembly instructions into potentially incorrect C/C++ code. Our approach focuses on a natural language interpretation of each part of the code, which provides a more intuitive understanding of the code's functionality, aligning with how developers typically conceptualize complex coding structures.

3.2. StackSight: WebAssembly Decompilation Pipeline

To facilitate stepwise reasoning, we build a pipeline shown in Figure 1 for WebAssembly decompilation, where each component focuses on one of the above decompilation phases. This structured pipeline ensures that LLM can progressively build upon its understanding.

3.2.1. Stack Visualization

In StackSight, we address the inherent complexities of WebAssembly's virtual stack machine through a specialized static analysis tool. This tool is adept at parsing the WebAssembly text format (WAT) code, thereby explicitly visualizing and tracking the behavior of the stack machine.

WebAssembly diverges from traditional native binaries in that it permits the definition of any number of local variables and effectively operates with an unlimited number of virtual registers on a virtual stack machine. This dependency obfuscates the understanding of the code's functionality, both for humans and computational models, without prior knowledge of the stack's configuration.

To overcome this challenge, StackSight includes a static analysis algorithm to visualize the stack after each operation. This approach mirrors how a human reverse engineer

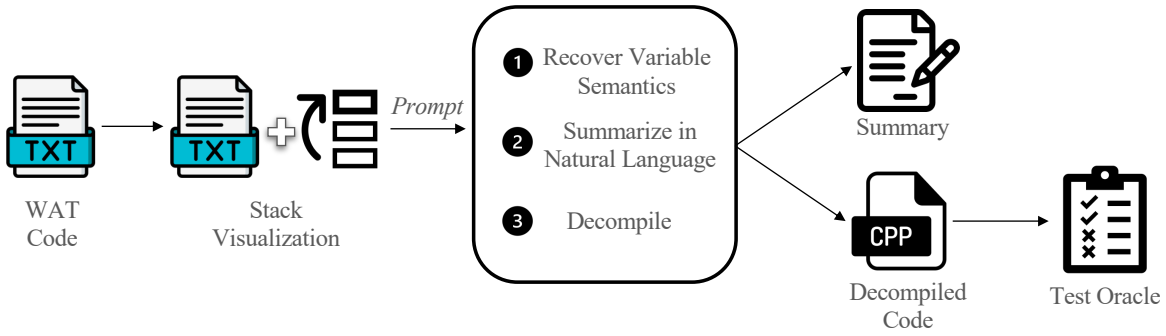

Figure 1. StackSight: The decompilation pipeline.

may jot down the stack and interpret operations. The algorithm supports all 172 WebAssembly opcodes, including memory operations, function calls, type conversions, control flow, and branching operations. As each opcode has its specific rules on how it manipulates the stack, StackSight simulates such manipulations to precisely determine the stack's resulting state from any given instruction and its current state. Moreover, it applies symbolic execution to account for control flow branches: When there are divergent execution paths that may result in different stack states, it will use symbolic notation to record different branches.

Our static analysis tool can annotate each line of WAT code with the precise state of the stack. The exact state of the stack at each line of code is represented with a simplified notation. This approach mirrors the steps a human reverse engineer might take, such as manually noting the stack's state to interpret its operations. An example can be seen in Figure 2. We show empirically that this program analysis tool can significantly bridge the gap between WASM code and its high-level source code counterparts.

3.2.2. CoT Prompting

The annotated WebAssembly code (`.wat`) is then included in the prompt to a large language model in a step-by-step manner. First, based on the stack visualization, we ask the LLM to comprehend variable relationships, predict the data types for input parameters and return types (*pivot variables*). The model is then prompted to recover the semantic meaning of each local variable based on the pivot variables. Given the context and usage of each local variable, the model is asked to summarize the function operations in natural language to derive a high-level understanding of what the WebAssembly code is doing. Finally, based on the above knowledge, the LLM is prompted to decompile the piece of WebAssembly code to C or C++. To evaluate the outputs from the LLM, we implement a test oracle that incorporates a series of test cases to evaluate the correctness of the decompiled code.

4. Experiment Setup

4.1. Datasets

We list our main considerations when choosing evaluation datasets. First, we need to ensure that the datasets are compilable to WebAssembly, to obtain input-output pairs. Second, we need to ensure that the datasets come with test cases and natural language descriptions to assess functional correctness of decompiled code snippets and the semantic similarity of model-generated summaries. Finally, we minimize data contamination concerns by verifying that the datasets are human-written. To this end, we use the two code benchmark datasets. Both datasets are originally targeted for Natural Language to Code generation. Each sample comes with test cases and a natural language task description that satisfies our needs. For examples of the two datasets, see Appendix A

HumanEval-X HumanEval-X (Zheng et al., 2023) is a widely-used Natural Language to Code dataset which has a split in C++. It is adapted from HumanEval (Chen et al., 2021) by human experts. The HumanEval dataset is also written by human experts, therefore reducing the risk of data contamination.

MBXP MBXP (Athiwaratkun et al., 2023) is also a Natural Language to Code dataset that contains a C++ split. It is created using a parsing-based conversion framework that generates C++ code from Python snippets in MBPP (Austin et al., 2021). The MBPP dataset is crowdsourced, therefore also deemed to be original.

4.2. Compilation Pipeline

We use Emscripten (The Emscripten project, 2024) (version 3.1.46) to compile C++ code to WebAssembly. We leverage the `wasm2wat` (version 1.0.33) tool from the WebAssembly Binary Toolkit repository (WebAssembly, 2024) to convert binary instructions (`.wasm`) into text format (`.wat`). To improve the readability of `.wat` files, we 1) compile function snippets as side modules 2) perform compiler op-

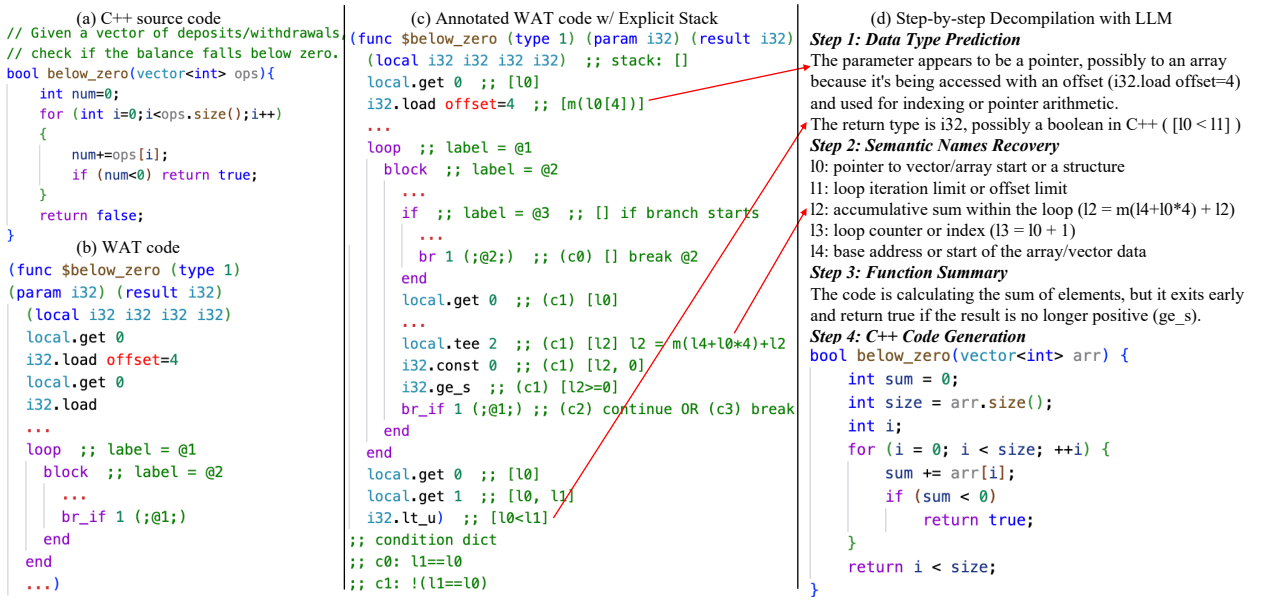

Figure 2. A Running example for StackSight. (a) C++ source code in HumanEval-X dataset. (b) WebAssembly Text Format (WAT) code compiled with Emscripten and translated with WABT. (c) Annotated WAT code with our proposed static analysis tool capturing the virtual stack alterations, variable assignments, control flow, and branching statements. (d) Chain-of-Thought prompting by breaking decompilation into multiple phases. LLM is able to utilize stack annotations, make reasonable predictions, summarize function purposes, and decompile WAT code to high-level C++.

Table 1. Data statistics for HumanEval-X and MBXP benchmarks from (Zan et al., 2023), with modifications. *Num.* denotes the number of instances in the benchmark, *Working Num.* denotes the number of instances that can compile, *S.PL* denotes code Solution's Programming Language (we are using the C++ split), *T.N.* denotes the average Number of Test cases per function, *W.C.* and *W.L.* for the average number of Characters and Lines in compiled WAT file, *S.C.* and *S.L.* for the average number of Characters and Lines in canonical Solution.

Benchmark	Num.	Working Num.	S. PL	T/N.	W/C.	W/L.	S/C.	S/L.	Scenario
HumanEval-X (2023)	164	161	C++	7.8	6784.9	295.2	252.5	10.4	Code Exercise
MBXP (2022)	974	773	C++	3.1	6063.9	160.1	192.9	9.2	Code Exercise

timization to reduce the size of generated code 3) preserve function names in C++ 4) present symbols in a readable way. The exact commands that we use are in Appendix C. For a summary on our compiled datasets, please refer to Table 1.

4.3. Models and Baselines

4.3.1. Baselines

A baseline approach is to elicit LLM's In-Context Learning (ICL) ability (Dai et al., 2023) by providing it with one or more examples as context. It is also known as one-shot or few-shot learning, which involves presenting LLMs with one or a few examples to provide context and understanding of a specific task. Such a method is widely applied for its simplicity and ability to quickly adapt to new tasks without the need for training or fine-tuning.

We construct in-context learning examples by concatenating three parts. We first start with an instruction part to explicitly state that the task to perform is to decompile WebAssembly files into the corresponding C++ file. We then append one or several round(s) of user and AI system interaction to serve as an in-context learning example. The user input is an exemplar `.wat` file, while the system output is the corresponding C++ file it is compiled from. Finally, we append the target `.wat` file of interest. For an example prompt, please refer to Appendix B. Other WebAssembly decompilers, such as `wasm2c` by WABT and (Benali, 2022), fail to generate compilable codes for the two datasets and are thus not included in the evaluation.

As the static analysis step involves constantly tracking the virtual stack machine and emulating branching operations, we find that the annotated code is nearly twice as large as the original code in average. For a fairer comparison, we include more shots in the ICL baselines to match the prompt size.

We include one shot in StackSight. As the annotated code is twice as large, 2-shot or 3-shot ICL baseline will match the prompt size of StackSight. To compare our method against stronger baselines, we randomly select examples from both datasets, and conduct the same set of experiments for 3-shot, 5-shot, and 10-shot ICL in our evaluation.

4.3.2. Models

We conduct experiments on three large language models: gpt-3.5-turbo-1106 (OpenAI, 2024a), gpt-4-0125-preview (OpenAI, 2024b), and Code Llama-7b-Instruct (Rozière et al., 2023). All three models have exhibited strong code interpretation and complex reasoning abilities. As WebAssembly is low-resource, and its decompilation is especially intricate, choosing less advanced models might compromise the quality and accuracy of the decompilation process. Besides, their easy accessibility ensures that our research can be replicated and applied by a broader community of developers.

5. Evaluation

We examine the effectiveness of our pipeline by addressing the following research questions:

RQ1: To what extent do the decompiled codes generated by our method match their original codes?

RQ2: Can our method more effectively help developers comprehend WebAssembly code in the real world?

RQ3: How does each component of our pipeline contribute to the overall performance of decompilation?

5.1. RQ1: Quality of Decompiled Code and Summary

In addressing this research question, our evaluation is twofold: assessing the quality of the generated summaries and examining the correctness of the decompiled code.

Evaluating Summary Quality We measure the quality of the summaries produced by our pipeline using both automated text quality metrics and human judgment.

Assessing Decompiled Code Correctness To evaluate the quality of the decompiled code, we choose **functional correctness** as our metric. Previous works on decompilation (Benali, 2022; Fu et al., 2019) have focused on metrics such as token-level accuracy, template-level accuracy and also complete match. Such metrics capture textual and lexical similarities. (Chen et al., 2021) suggests that text-similarity based metrics do not correlate with functional correctness, and proposes to directly evaluate correctness through expert-written test cases. We follow this setup to quantify the quality of decompiled code. We employ a test oracle to incorporate C/C++ test cases into the decompiled code and report the pass rate to assess the effectiveness of our decompilation pipeline.

5.1.1. Summary Evaluation

Evaluating the degree of alignment between the predicted summaries and ground-truth summaries provided in the dataset presents a challenge because they may be written in different styles or formats but are semantically similar to each other (Fang & Jiang, 2022). In order to capture semantic similarity, we propose BERTScore (Zhang et al., 2020) and Sentence-BERT (SBERT) (Reimers & Gurevych, 2019) for evaluation. BERTScore leverages the contextual embeddings from BERT to compute the similarity between predicted and reference sentences. SBERT is designed for creating dense vector representation of sentences. Similarity scores are then computed using cosine similarity.

5.1.2. Decompilation Correctness Evaluation

Using automatic parsers to extract desired model output and automate the evaluation process is a common practice in evaluating LLMs, as showcased in MetaMathQA (Yu et al., 2023) and MMIQC (Liu & Yao, 2024). We implement the test oracle to automate the evaluation, including five steps:

Code Sanitization The test oracle parses the outputs from LLMs and extracting the functions under test (FUT). In some cases, the generated code may include an extraneous `main()` function intended to validate the usage of the FUT. In such instances, the test oracle removes it, since test cases will be integrated into a new main function automatically.

Function Name Alignment Function names from decompiled code may divert from those expected by the test cases because WebAssembly compilers such as Emscripten may strip away function identifiers. The test oracle will rename the function names from the decompiled code to match those expected by the test cases, utilizing regular expression

Test Case Integration Subsequently, the test oracle will insert test cases pertinent to this function into the decompiled code in a new `main()` function. This step assesses whether the decompiled code meets the functional requirements as specified by the test suite.

Compilation Error Rectification We address compilation errors that do not impact the underlying functionality.

Function argument pass-by-reference: Generated code from LLM often uses pass-by-reference for function arguments to minimize copying overhead. However, in the tests within both evaluation datasets, constant vectors are often passed to function arguments. To address this issue, we manually modify the function argument to use pass-by-value or prefix the `const` modifier when it does not alter the function's intended behavior.

Namespace issues: In some instances, the tests presuppose the inclusion of `using namespace std;` directive. This line is automatically inserted where necessary

Test Case Execution Finally, we compile and execute the test cases against the decompiled code. The test oracle will output the percentage of decompiled files that can compile and those that can pass all test cases.

Discussion This correctness evaluation focuses on functional equivalence rather than syntactic similarity, which is a deliberate choice to match the intended use cases. In real use scenario, users are only presented with the WebAssembly code but not the original source code, and during compilation, identifiers and logic may change depending on the compiler, compilation flags, etc. Syntax may change and a different set of variables may be used, while the core functionality should be invariant. By using a test-driven approach, we can objectively measure the extent to which the decompiled code represents the functionality of the original WebAssembly.

Among the five steps, the only manual intervention arises in Step 4 for fixing function argument pass-by-reference. We deliberately choose not to employ scripts to fix it automatically because altering function arguments may inadvertently modify function semantics. We manually analyze each case to ensure that adjustments preserve the original logic.

5.1.3. Results

Our experiment results, detailed in Table 2, reveal that StackSight with one-shot still achieves the highest performance in the majority of the metrics, despite longer prompts and additional demonstration examples in the ICL baselines. These results demonstrate that our approach exhibits a superior ability to accurately capture and reflect the true purposes of the WebAssembly code, thus yielding more successful decompilation outcomes.

Regarding decompilation correctness, StackSight demonstrates the highest scores across all models and baselines. More specifically, we notice that StackSight brings twice as much improvement on HumanEval-X as on MBXP. Further analysis reveals that the discrepancy may be because many code snippets in MBXP are synthesized by a code generation model (Athiwaratkun et al., 2023). These snippets, while passing test cases, often fail to align with the original functional objectives. Despite this challenge, StackSight still manages to achieve a 5 to 10 percentage increase in functional correctness. This indicates that StackSight significantly improves the interpretation of WebAssembly code, leading to more accurate decompilation. In addition, larger models (GPTs) benefit more from StackSight than Code Llama-7b, which is consistent with previous observations that complex reasoning abilities are more prominent in larger models ($\geq$ 100B) (Wei et al., 2022a; Fu et al., 2023).

StackSight also exhibits competitive performance in BERTScore and SBERT metrics for GPT-3.5 and highest performance for Code Llama. This suggests that StackSight-equipped LLMs excel at grasping the high-level functionalities of the code and generating natural language summaries that are semantically closer to the actual intentions of the original code.

In addition, we observe that increasing the number of demonstration examples improves the summarization and decompilation quality in most cases. This suggests that further enhancing StackSight's performance is possible by providing more, higher-quality examples.

5.1.4. Case Study

We present two case studies that exemplify the effectiveness of StackSight. The first case study, illustrated in Figure 2, showcases the capabilities of our static analysis algorithm. It accurately represents stack operations, delineates variable relationships, and navigates if-else branches. This detailed representation provides the LLM with rich contextual information, enabling it to successfully predict data types, restore variable semantics, succinctly summarize functional objectives, and generate a functionally equivalent version of the C++ code.

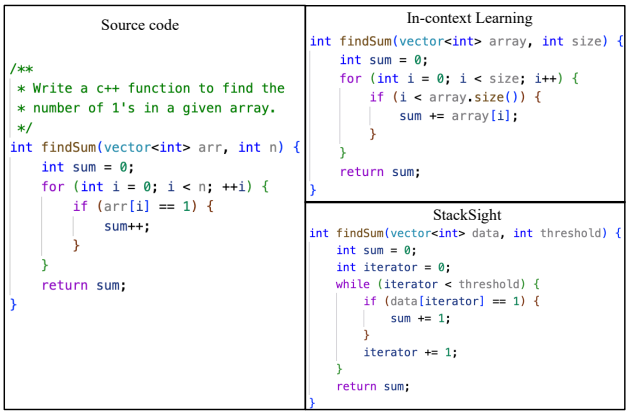

Figure 3. Hallucination case study.

The second case study, shown in Figure 3, focuses on StackSight's ability to counter hallucination, especially in scenarios with misleading function names. In this example, the function named `findSum` is actually intended to count the occurrence of 1's in a given array. We present two versions of decompiled code generated by gpt-3.5-turbo with ICL and with StackSight. The ICL version falsely generates a function that calculates the sum of all elements in the array, misled by the function name. In contrast, the StackSight version accurately discerns the true purpose of the function. This case study highlights that StackSight helps elicit

Table 2. Summary evaluation and comparison of pass rate of different methods. For summary evaluation, we report BERTScore and SBERT. For decompilation quality, we report the percentage of outputs that can compile successfully and pass all test assertions.

	HumanEval-X				MBXP			
Method	Compile	Correct	BERTScore	SBERT	Compile	Correct	BERTScore	SBERT
GPT-3.5								
1-shot	38.2	13.4	80.9	38.3	48.0	19.6	83.3	36.3
3-shot	49.2	18.3	82.7	42.8	41.5	21.1	86.3	45.0
5-shot	50.7	22.1	**84.5**	50.0	38.5	18.8	**88.4**	**52.0**
10-shot	51.6	23.4	84.1	50.7	23.9	10.2	86.9	50.6
StackSight	**58.1**	**31.4**	83.6	**51.8**	**54.9**	**29.5**	84.6	51.3
Code Llama								
1-shot	47.7	13.5	79.5	37.1	38.8	10.3	81.5	37.0
3-shot	42.6	10.8	79.9	40.4	21.2	5.5	82.1	40.9
5-shot	46.0	12.7	80.3	41.7	31.1	8.2	82.1	39.2
10-shot	50.0	10.7	80.6	43.7	29.2	9.0	82.4	40.4
StackSight	**54.2**	**24.6**	**83.1**	**53.5**	**41.8**	**15.4**	**85.4**	**55.0**
GPT-4								
1-shot	46.5	19.7	78.75	33.9	63.6	42.4	80.7	34.5
StackSight	60.6	36.5	82.05	46.5	85.9	59.6	84.0	47.2
Δ	+14.1	+16.8	+3.30	+12.6	+22.3	+17.2	+3.3	+12.7

complex reasoning capabilities in LLMs and its potential to correct misinterpretations in decompilation tasks.

5.2. RQ2: User Study

To rigorously evaluate the efficacy of our decompiling methodology, we conduct a structured user study. We begin by randomly selecting twenty code snippets from each dataset for the user study. For every code snippet, we initiated the decompilation process utilizing two distinct approaches: the baseline In Context Learning (ICL) method and the StackSight approach, employing both Code Llama and GPT-3.5-turbo models. This procedure yielded four decompiled versions for each snippet.

Subsequently, we carefully developed three multiple-choice questions per code snippet based on the source code, aligning with three distinct objectives: functionality comprehension, input-output relationship understanding, and edge case handling. These objectives range from a general grasp of the code's purpose to intricate details. Correct answers to these evaluation questions indicate that the decompiled code effectively captures and conveys the original functionality in a manner that is both accurate and human-readable. See C.1 for more details.

Functional Comprehension Participants are presented with three multiple-choice questions to answer based on the StackSight version and the baseline version. It aims to test how well participants can comprehend the given code snippet, thereby assessing whether our decompilation approach aids in producing code that is not only functionally equivalent but also easily comprehensible.

Code Similarity Following the initial comprehension evaluation, participants are presented with the original source code and its two decompiled versions. They are tasked with choosing which decompiled version most closely resembles the semantic content of the original source code, through comparison. This stands in parallel to the test-driven approach, as the decompiled code may capture the essence of the binary code but fails to pass the test cases due to some edge case handling. This process provides an additional layer of evaluation as to how well StackSight manages to recover high-level code semantics.

5.2.1. User Study Statistics

We recruited 15 participants in our user study. To better mimic a real-world scenario, our participants are either graduate students in engineering or professional developers. All participants are proficient in C/C++. The total time for completing our questions ranges from 45 to 60 minutes. We ensure that for each designed question, there are two to three responses from the participants to reduce personal bias.

Figure 4 presents the results of our study on functionality comprehension. We report the average percentage of correct responses from participants given the decompiled code. Models equipped with StackSight markedly outperform the

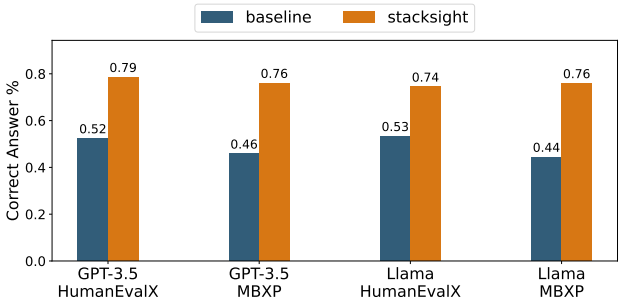

Figure 4. Comparison of correct answer percentages.

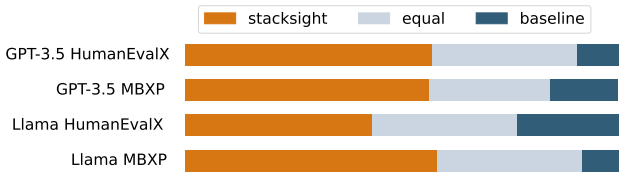

Figure 5. Comparison of similarity ratios.

baseline, achieving at least a 20% higher rate of correct answers. Figure 5 depicts the results of our code similarity evaluation, where each bar represents the proportion of participants who identified code snippets as closely resembling the source code. Remarkably, StackSight approach exhibited a higher resemblance ratio in comparison to the baseline.

5.3. RQ3: Ablation Study

To understand the contribution of each component within StackSight, we conduct an ablation study on the following elements: stack visualization, recovery of variable semantics, and generation of natural language summaries. All prompts can be found in Appendix B.3.

5.3.1. Results

Table 3. Ablation study.

% Correct	HumanEval-X	MBXP
GPT-3.5	13.4	19.6
+StackSight	31.4	29.5
No stack visualization	23.6	26.5
Basic CoT	22.2	26.2
No variable semantics	24.8	22.7
No function summary	20.7	24.3

As detailed in Table 3, we report the percentage of decompiled code snippets that can pass all test cases by removing each component for the two datasets. Given that the annotated binary code is nearly twice as large as the original version, we include another example for "No stack visualization" ablation to match the prompt size.

Removing either the explicit stack visualization or the variable semantics recovery significantly reduces the contextual information, leading to a degradation of decompilation accuracy. The removal of stack visualization leads to a decrease in the successful decompilation rate by 7.8% and 3% for the two datasets, and the absence of variable semantics recovery results in a performance drop by 6.6% and 6.8%. These results underscore the importance of translating low-level WebAssembly operations and data types to high-level counterparts.

The omission of the natural language summary component also significantly impacts performance, with a decrease of 10.7% and 5.2%. Removing the step of summarization causes LLM to resort to a more literal transcription of the WebAssembly code, resulting in verbatim output and incorrect C++ code. These findings underscore the integral role each component plays in our decompilation pipeline.

We also include the basic chain-of-thought prompting by using a short system prompt, while keeping everything else the same as the "No stack visualization" ablation. We observe that the extended system prompt leads to marginal improvement of 1.4% and 0.3%.

6. Conclusion

In this paper, we propose StackSight, a three-stage neural-symbolic method that decompiles WebAssembly into C++ code. Each stage of our pipeline is carefully designed to tackle a challenge in understanding WebAssembly. We have shown empirically that StackSight significantly enhances the functional correctness of decompiled C++ snippets and improves the semantic relevance of model generated summaries. We have also conducted case studies to showcase how StackSight mitigates the risks of misinterpretation by LLMs in ambiguous contexts. Furthermore, the notably higher win-rate of StackSight generated C++ code in our user study further underscores the effectiveness of StackSight. Our work pushes forward the capabilities of current WebAssembly decompilation toolkits. Hopefully, our work can shed light upon eliciting the power of large language models for low-resource code languages.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback. This work was partially supported by the US National Science Foundation under Grant No. 2321444. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

Impact Statement

Our work will enhance the comprehension of WebAssembly code distributed on the Internet. Binary codes are frequently minified or obfuscated, and they may serve unintended or malicious functional purposes. Our work proposes StackSight, which proves extremely useful by decompiling these binaries into high-level, human-readable programs complemented by natural language summaries. We hope that our work can reduce the prevalence of WebAssembly-based malware, fostering a more transparent environment for third-party WebAssembly code.

References

Ahad, A., Jung, C., Askar, A., Kim, D., Kim, T., and Kwon, Y. Pyfet: Forensically equivalent transformation for python binary decompilation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 3296–3313, 2023. doi: 10.1109/SP46215.2023.10179370.

Al-Kaswan, A., Ahmed, T., Izadi, M., Sawant, A. A., Devanbu, P., and van Deursen, A. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 260–271, 2023. doi: 10.1109/SANER56733.2023.00033.

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., Gonugondla, S. K., Ding, H., Kumar, V., Fulton, N., Farahani, A., Jain, S., Giaquinto, R., Qian, H., Ramanathan, M. K., Nallapati, R., Ray, B., Bhatia, P., Sengupta, S., Roth, D., and Xiang, B. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021.

Benali, A. An initial investigation of neural decompilation for webassembly, 2022.

Cao, Y., Liang, R., Chen, K., and Hu, P. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pp. 508–518, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

Cifuentes, C. and Gough, K. J. Decompilation of binary programs. *Software: Practice and Experience*, 25(7): 811–829, 1995.

Cifuentes, C. G. A structuring algorithm for decompilation. 1993. URL https://api.semanticscholar.org/CorpusID:17905992.

Dai, D., Sun, Y., Dong, L., Hao, Y., Ma, S., Sui, Z., and Wei, F. Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers, 2023.

Desnos, A. and Gueguen, G. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 1:1–24, 2011.

ElWazeer, K., Anand, K., Kotha, A., Smithson, M., and Barua, R. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pp. 51–60, 2013.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

Fang, W. and Jiang, M. Investigating relationships between accuracy and diversity in multi-reference text generation. 2022.

Fokin, A., Derevenetc, E., Chernov, A., and Troshina, K. Smartdec: approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pp. 347–356. IEEE, 2011.

Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., and Zhao, J. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.

Fu, Y., Peng, H., Ou, L., Sabharwal, A., and Khot, T. Specializing smaller language models towards multi-step reasoning. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Gurdeep Singh, R. and Scholliers, C. Warduino: a dynamic webassembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pp. 27–36, 2019.

Gussoni, A., Di Federico, A., Fezzardi, P., and Agosta, G. A comb for decompiled c code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, pp. 637–651, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200, 2017a.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pp. 185–200, New York, NY, USA, 2017b. Association for Computing Machinery. ISBN 9781450349888.

Harrand, N., Soto-Valero, C., Monperrus, M., and Baudry, B. Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*, 168: 110645, 2020.

Hilbig, A., Lehmann, D., and Pradel, M. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the web conference 2021*, pp. 2696–2708, 2021.

Katz, D. S., Ruchti, J., and Schulte, E. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 346–356. IEEE, 2018.

Katz, O., Olshaker, Y., Goldberg, Y., and Yahav, E. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M., and Bailey, M. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, WWW '19, pp. 840–852, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366748.

Konoth, R. K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., and Vigna, G. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pp. 1714–1730, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930.

Lee, J., Avgerinos, T., and Brumley, D. Tie: Principled reverse engineering of types in binary programs. 2011.

Lehmann, D. and Pradel, M. Finding the dwarf: recovering precise types from webassembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 410–425, 2022.

Liang, R., Cao, Y., Hu, P., He, J., and Chen, K. Semantics-recovering decompilation through neural machine translation. *arXiv preprint arXiv:2112.15491*, 2021.

Liu, H. and Yao, A. C.-C. Augmenting math word problems via iterative question composing. *arXiv preprint arXiv:2401.09003*, 2024.

Liu, R., Garcia, L., and Srivastava, M. Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 94–105. IEEE, 2021.

Liu, X., Song, Z., Fang, W., Yang, W., and Wang, W. Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests. In *Proceedings of the ACM on Web Conference 2024*, pp. 3043–3052, 2024.

Liu, Z. and Wang, S. How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 475–487, 2020.

McCallum, T. Diving into ethereum's virtual machine(evm): the future of ewasm, 2019.

McConnell, J. Webassembly support now shipping in all major browsers, 2017. URL https://blog.mozilla.org/en/mozilla/webassembly-in-browsers/. The Mozilla Blog.

Musch, M., Wressnegger, C., Johns, M., and Rieck, K. New kid on the web: A study on the prevalence of webassembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pp. 23–42. Springer, 2019a.

Musch, M., Wressnegger, C., Johns, M., and Rieck, K. Thieves in the browser: Web-based cryptojacking in the wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450371643.

Noonan, M., Loginov, A., and Cok, D. Polymorphic type inference for machine code. *SIGPLAN Not.*, 51(6):27–41, jun 2016. ISSN 0362-1340.

OpenAI. Gpt-3.5 turbo, 2024a. URL https://platform.openai.com/docs/models/gpt-3-5-turbo. Accessed on 2024-06-03.

OpenAI. Gpt-4 turbo and gpt-4, 2024b. URL https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo. Accessed on 2024-06-03.

Pearce, H., Tan, B., Krishnamurthy, P., Khorrami, F., Karri, R., and Dolan-Gavitt, B. Pop quiz! can a large language model help with reverse engineering? *arXiv preprint arXiv:2202.01142*, 2022.

Pop, V. A. B., Niemi, A., Manea, V., Rusanen, A., and Ekberg, J.-E. Towards securely migrating webassembly enclaves. In *Proceedings of the 15th European Workshop on Systems Security*, pp. 43–49, 2022.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

Romano, A. and Wang, W. Automated webassembly function purpose identification with semantics-aware analysis. In *Proceedings of the ACM Web Conference 2023*, pp. 2885–2894, 2023.

Romano, A., Zheng, Y., and Wang, W. Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1129–1140, 2020.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2023.

The Emscripten project. Emscripten, 2024. URL https://github.com/emscripten-core/emscripten. Accessed on 2024-06-03.

Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., and Vigna, G. Ramblr: Making reassembly great again. In *NDSS*, 2017.

Wang, S., Wang, P., and Wu, D. Reassembleable disassembling. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pp. 627–642, USA, 2015. USENIX Association. ISBN 9781931971232.

WebAssembly. The webassembly binary toolkit, 2024. URL https://github.com/WebAssembly/wabt. Accessed on 2024-06-03.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022a. ISSN 2835-8856. Survey Certification.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022b.

Wong, W. K., Wang, H., Li, Z., Liu, Z., Wang, S., Tang, Q., Nie, S., and Wu, S. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

Xu, X., Zhang, Z., Feng, S., Ye, Y., Su, Z., Jiang, N., Cheng, S., Tan, L., and Zhang, X. Lmpa: Improving decompilation by synergy of large language model and program analysis. *arXiv preprint arXiv:2306.02546*, 2023.

Xu, Z., Wen, C., and Qin, S. Learning types for binaries. In *Formal Methods and Software Engineering: 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*, pp. 430–446. Springer, 2017.

Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

Yu, L., Jiang, W., Shi, H., Jincheng, Y., Liu, Z., Zhang, Y., Kwok, J., Li, Z., Weller, A., and Liu, W. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Yongji, W., and Lou, J.-G. Large language models meet NL2Code: A survey. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics.

Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., and Artzi, Y. Bertscore: Evaluating text generation with bert. In *International Conference for Learning Representation (ICLR)*, 2020.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pp. 5673–5684, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701030.

Appendix

A. Original Dataset Examples

A.1. HumanEval-X Example

```
// prompt:
/*
Check if in given vector of numbers, are
    any two numbers closer to each other
    than
given threshold.
>>> has_close_elements({1.0, 2.0, 3.0},
    0.5)
false
>>> has_close_elements({1.0, 2.8, 3.0, 4.0,
     5.0, 2.0}, 0.3)
true
*/
#include<stdio.h>
#include<vector>
#include<math.h>
using namespace std;
bool has_close_elements(vector<float>
    numbers, float threshold){
// canonical solution:
    int i,j;

    for (i=0;i<numbers.size();i++)
    for (j=i+1;j<numbers.size();j++)
    if (abs(numbers[i]-numbers[j])<
        threshold)
    return true;

    return false;
}
// test:
#undef NDEBUG
#include<assert.h>
int main(){
    vector<float> a={1.0, 2.0, 3.9, 4.0,
        5.0, 2.2};
    assert (has_close_elements(a, 0.3)==
        true);
    assert (has_close_elements(a, 0.05) ==
        false);

    assert (has_close_elements({1.0, 2.0,
        5.9, 4.0, 5.0}, 0.95) == true);
    assert (has_close_elements({1.0, 2.0,
        5.9, 4.0, 5.0}, 0.8) ==false);
    assert (has_close_elements({1.0, 2.0,
        3.0, 4.0, 5.0}, 2.0) == true);
    assert (has_close_elements({1.1, 2.2,
        3.1, 4.1, 5.1}, 1.0) == true);
    assert (has_close_elements({1.1, 2.2,
        3.1, 4.1, 5.1}, 0.5) == false);

}
```

A.2. MBXP Example

```
// prompt:
#include <bits/stdc++.h>
using namespace std;


/**
 * Write a c++ function to check whether
     the two numbers differ at one bit
     position only or not.
 * > differAtOneBitPos(13, 9)
 * true
 * > differAtOneBitPos(15, 8)
 * false
 * > differAtOneBitPos(2, 4)
 * false
 */
bool differAtOneBitPos(int a, int b) {
// canonical solution:
    return (a == b) || (a == 13 && b == 9);
}

// test:
template <typename T> bool compare(T a, T b
    ){
    return a == b;
}

int main(int argc, char* argv[]) {
    bool x0 = differAtOneBitPos(13, 9);
    if (!(compare(x0, true))) {
        throw runtime_error("Exception --
            test case 0 did not pass.");
    }

    bool x1 = differAtOneBitPos(15, 8);
    if (!(compare(x1, false))) {
        throw runtime_error("Exception --
            test case 1 did not pass.");
    }

    bool x2 = differAtOneBitPos(2, 4);
    if (!(compare(x2, false))) {
        throw runtime_error("Exception --
            test case 2 did not pass.");
    }

    return 0;
}
```

```
(module
  (type (;0;) (func))
  (type (;1;) (func (param i32) (result i32
     )))
  (func (;0;) (type 0))
  (func (;1;) (type 1) (param i32) (result
     i32)
    local.get 0
    local.get 0
    i32.mul)
  (export "__wasm_call_ctors" (func 0))
  (export "__wasm_apply_data_relocs" (func
     0))
  (export "_Z18car_race_collisioni" (func
     1)))

AI: ```cpp

#include<stdio.h>
#include<math.h>
using namespace std;
#include<algorithm>
#include<stdlib.h>
int car_race_collision(int n){
    return n*n;
}

```
Human:

(module
(type (;0;) (func))
(type (;1;) (func (param i32) (result i32))
    )
(func (;0;) (type 0))
(func (;1;) (type 1) (param i32) (result
    i32)
    local.get 0
    i32.const 1
    i32.add
    local.get 0
    i32.mul
    i32.const 2
    i32.div_s)
(export "__wasm_call_ctors" (func 0))
(export "__wasm_apply_data_relocs" (func 0)
    )
(export "_Z8sum_to_ni" (func 1)))
```

B. Prompts

B.1. In-Context Learning Prompt

```
System: You are a professional
    developer who translates
    WebAssembly text into C++ code.
Given the following WebAssembly text
    file, please provide the equivalent
     C++ code.
Human:
```
```

B.2. CoT Learning Prompt

```
System: You are a professional reverse
    engineer tasked with translating
    WebAssembly text (WAT) into C++
    code. The process involves
    interpreting annotated WAT code,
    understanding its stack state and
    conditional branches, predicting
    data types and semantics,
    summarizing in natural language,
    and finally generating equivalent C
    ++ code. Let's do it step by step.

Step 1: Data Type Prediction
Given annotated WAT code, examine each
    line's comments that visualize the
    virtual stack state.
Identify and predict the high-level C/C
    ++ data types for input parameters
    and return values based on their
    operations and usage in the code.

Step 2: Semantic Variable Recovery
Analyze stack visualizations to extract
     variable relationships.Recover and
     describe the semantic meaning of
    each variable, including parameters
    , returns, and local variables. Use
     'l0', 'l1', etc., to represent
    variables corresponding to local.
    get 0, local.get 1, and so on.

Step 3: Function Summary
Revisit the source WAT function based
    on your understanding of variable
    semantics. Summarize the function's
     purpose and operation in a few
    lines, focusing on the overall
    functionality.

Step 4: C++ Code Generation
Going back to the original WAT code,
    decompile it into equivalent C++
    code based on the variable
    semantics and function summary.
    Write the C++ code in a standard
    format, avoiding direct line-by-
    line translation from WAT. Enclose
    C++ code within "cpp" and "".

Format for Output:
Clearly indicate each step's output
    with "Step 1:", "Step 2:", etc.
Provide explanations and reasoning for
    each step's conclusion."
Human:
```

```
(module $code_3.wasm
  (type (;0;) (func))
  (type (;1;) (func (param i32) (result i32
     )))
  (import "env" "memory" (memory (;0;) 0))
  (func $__wasm_call_ctors (type 0))
  (func $below_zero_std::__2::vector<
     int__std::__2::allocator<int>>_ (type
      1) (param i32) (result i32)
    (local i32 i32 i32 i32)  ;; stack: []
    local.get 0  ;; [l0]
    i32.load offset=4  ;; [(l0[4])_mem]
    local.get 0  ;; [(l0[4])_mem, l0]
    i32.load  ;; [(l0[4])_mem, (l0)_mem]
    local.tee 4  ;; [(l0[4])_mem, l4] ;; l4
       = (l0)_mem
    i32.sub  ;; [(l0[4])_mem-l4]
    i32.const 2  ;; [(l0[4])_mem-l4, 2]
    i32.shr_s  ;; [(l0[4])_mem-l4/4]
    local.set 1  ;; [] ;; l1 = (l0[4])_mem-
       l4/4
    loop  ;; label = @1
      block  ;; label = @2
        local.get 1  ;; [l1]
        local.get 3  ;; [l1, l3]
        local.tee 0  ;; [l1, l0] ;; l0 = l3
        i32.eq  ;; [l1==l0]
        if  ;; label = @3  ;; [] branch
           starts
          local.get 1  ;; (c0) [l1] ;; c0:
             l1==l0
          local.set 0  ;; (c0) [] ;; l0 =
             l1
          br 1 (;@2;)  ;; (c0) [] break out
               of the block at @2
        end
        local.get 0  ;; (c1) [l0] ;; c1: !(
           l1==l0)
        i32.const 1  ;; (c1) [l0, 1]
        i32.add  ;; (c1) [l0+1]
        local.set 3  ;; (c1) [] ;; l3 = l0
           +1
        local.get 4  ;; (c1) [l4]
        local.get 0  ;; (c1) [l4, l0]
        i32.const 2  ;; (c1) [l4, l0, 2]
        i32.shl  ;; (c1) [l4, l0*4]
        i32.add  ;; (c1) [l4+l0*4]
        i32.load  ;; (c1) [(l4+l0*4)_mem]
        local.get 2  ;; (c1) [(l4+l0*4)_mem
           , l2]
        i32.add  ;; (c1) [(l4+l0*4)_mem+l2]
        local.tee 2  ;; (c1) [l2] ;; l2 = (
           l4+l0*4)_mem+l2
        i32.const 0  ;; (c1) [l2, 0]
        i32.ge_s  ;; (c1) [l2>=0]
        br_if 1 (;@1;)  ;; (c2) [] continue
             the loop at @1 OR (c3) [] move
             on to next line ;; c2: c1 &&
            l2>=0 ;; c3: c1 && !(l2>=0)
      end
```

```
    end
    local.get 0  ;; [l0]
    local.get 1  ;; [l0, l1]
    i32.lt_u)  ;; [l0<l1]
  (export "__wasm_call_ctors" (func
     $__wasm_call_ctors))
  (export "__wasm_apply_data_relocs" (func
     $__wasm_call_ctors))
  (export "
     _Z10below_zeroNSt3__26vectorIiNS_
  9allocatorIiEEEE" (func $below_zero_std::
     __2::vector<int__std::__2::allocator<
     int>>_)))
;; condition dict
;; c0: l1==l0
;; c1: !(l1==l0)
;; c2: c1 && l2>=0
;; c3: c1 && !(l2>=0)
```

AI:
Step 1:Data Type Prediction

The function's parameter is used in memory operations with an offset (i32.load offset=4), suggesting it's likely a pointer to an array or a complex data structure. Shift operations (i32.shr_s / i32.shl) by 2 places are observed, so it is likely pointer arithmetic of an array of 4-byte objects, such as an integer array. The return value is the result of a comparison.Thus, the return type is likely a boolean (True/False) in C++.

Step 2: Semantic Variable Recovery

l0: Initially loaded from memory and later used as a loop counter. It likely represents a pointer to the start of an array or a similar structure, based on memory access patterns (l4 = l0_mem). As it's incremented in the loop (l0=l3 and l3 = l0+1), it also serves as an index or counter within the array.

l1: Calculated as a shifted and subtracted value from l0, likely representing a boundary or limit within the data structure, possibly related to the size or an upper limit of an array (l1 = l0[4]_mem - l4 / 4 where l4 = l0_mem)

l2: Appears to accumulate values in a loop, suggesting it's used for a running total or sum. The way it's updated in each iteration indicates an accumulative operation (l2 = (l4+l0<<2_mem)+l2)

l3: Incremented in the loop, closely following l0, it seems to function as an auxiliary counter or a modified index within the loop (l3 = l0 + 1)

l4: Loaded directly from memory (l0_mem) and used in arithmetic operations related to other variables. It likely represents a base address or a reference point within the data structure, possibly the starting address of the array's data.

Step 3: Function Summary

The function iteratively processes elements of an integer array or vector. It involves accumulating values and checking if this cumulative sum falls below zero. The function terminates the iteration early if this condition is met, which is a typical pattern for functions that perform a check or calculation until a certain condition is no longer satisfied when the sum l2 is no longer positive The final comparison (i32.lt_u), comparing l0 and l1, suggests a boundary check, likely ensuring the loop does not exceed the array's limits.
In essence, this function appears to be designed to check whether the cumulative sum of elements in an integer array ever becomes negative. If so, it returns true, indicating this condition has been met. Otherwise, the function returns false.

Step 4: C++ Code Generation

```cpp
```

```
#include<stdio.h>
#include<vector>
using namespace std;
bool below_zero(vector<int> operations){
    int num=0;
    for (int i=0;i<operations.size();i++)
    {
        num+=operations[i];
        if (num<0) return true;
    }
    return false;
}
```

```
Human:

(module $code_2.wasm
  (type (;0;) (func))
  (type (;1;) (func (param f32) (result f32
     )))
  (func $__wasm_call_ctors (type 0))
  (func $truncate_number_float_ (type 1) (
     param f32) (result f32)
    local.get 0  ;; [l0]
    block (result i32)  ;; label = @1
      local.get 0  ;; [l0, l0]
      f32.abs  ;; [l0, abs(l0)]
      f32.const 0x1p+31 (;=2.14748e+09;)
          ;; [l0, abs(l0), 0x1p+31]
      f32.lt  ;; [l0, abs(l0)<0x1p+31]
      if  ;; label = @2  ;; [l0]
        local.get 0  ;; [l0, l0]
        i32.trunc_f32_s  ;; [l0, trunc(l0)]
        br 1 (;@1;)  ;; [l0] Break Out of
            The Loop at @1
      end
      i32.const -2147483648  ;; [l0,
         -2147483648]
    end
    f32.convert_i32_s  ;; [l0, f32(trunc(l0
       ))]
    f32.sub)  ;; [l0-f32(trunc(l0))]
  (export "__wasm_call_ctors" (func
     $__wasm_call_ctors))
  (export "__wasm_apply_data_relocs" (func
     $__wasm_call_ctors))
  (export "_Z15truncate_numberf" (func
     $truncate_number_float_)))
```

B.3. Ablation Study Prompts

B.3.1. No Stack Visualization

We remove static analysis annotation from each of the Wasm binaries, while everything else remains the same as Appendix B.2.

B.3.2. No Stack Visualization + Basic CoT

We replace the extended system prompt with a short system prompt, while keeping everything else the same as the "No stack visualization" ablation.

```
System: You are a professional reverse
    engineer tasked with translating
    WebAssembly text (WAT) into C++
    code. Let's do it step by step.
```

B.3.3. No Variable Semantics

We remove Step 2 from the CoT learning prompt in Appendix B.2, resulting in a decrease of 70 tokens in the prompt, which is negligible.

B.3.4. No Function Summary

We remove Step 3 from the CoT learning prompt in Appendix C.2, which leads to a decrease of 43 tokens in the prompt.

C. Compilation Command

```
emcc "$target_cpp" -o "$output_wasm" \
    --profiling-funcs \ # keep function
    # names
    -Wl,--demangle \
    -s SIDE_MODULE=1 \ # compile as
    # side module
    -Oz # optimize size
wasm2wat "$output_wasm" -o "$output_wat"
```

C.1. User Study

This section outlines the structure of our user study, describing the initial setup followed by an exemplification of a subset of the study.

C.2. Setup

Our study is conducted online, with a web app we developed to ensure a user-friendly and streamlined process. All data was collected anonymously to maintain participant privacy. Participants were first acquainted with the study through instructions and a visual tutorial within the application, which provided a clear understanding of the task requirements.

C.3. A Sample of the study

As depicted in Figure 6. Upon entering the evaluation system, participants will see the instructions first.

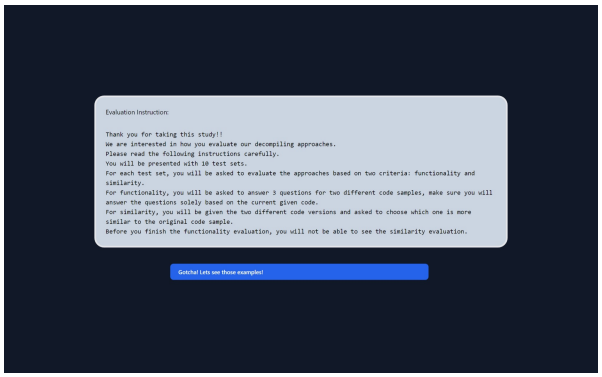

Figure 6. Task instructions.

Then by clicking the button below, they will enter example page and they will see a example set with detailed tutorials of the complete user study process. The tutorials is consisted of many steps, for simplicity we show the tutorial with Figures 7 and 8 illustrating a selection of these.

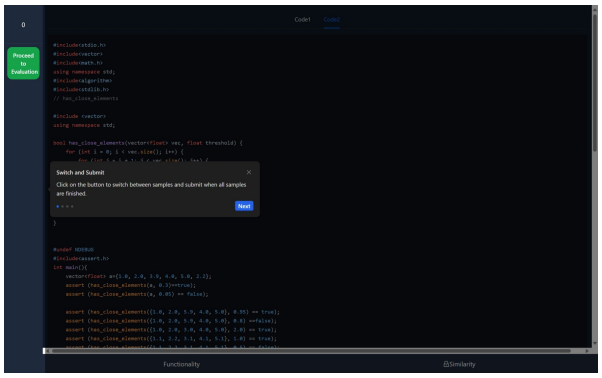

Figure 7. System tutorial sample 1.

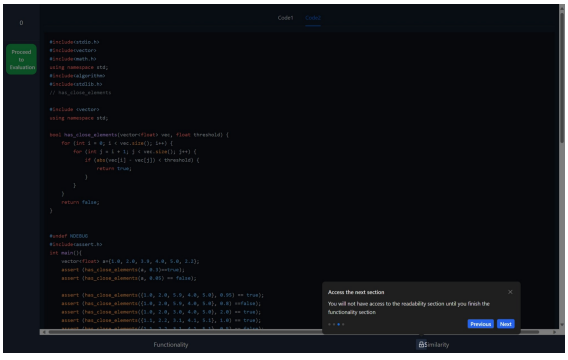

Figure 8. System tutorial sample 2.

This stepwise tutorial ensures that participants are well-informed about the study's progression, emphasizing that they must complete the functionality assessment before proceeding, thereby maintaining consistency throughout the study.

Next, we will show how we display our evaluation tasks.

Functional Comprehension

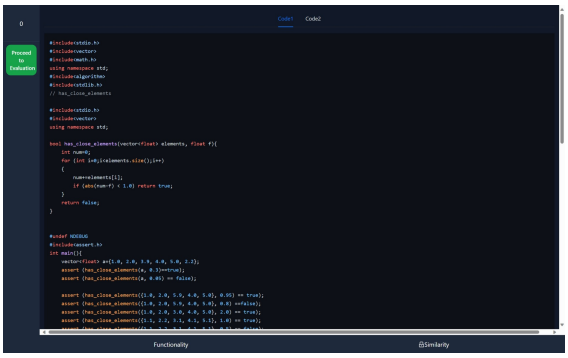

Figure 9. Functionality samples.

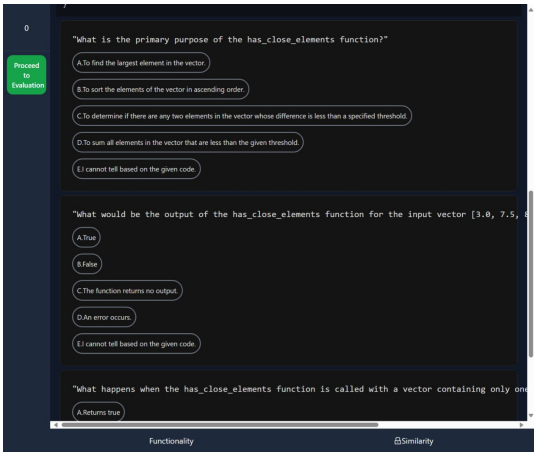

Figure 10. Functionality questions.

In figure 9 we can clearly see each set has two code samples—one derived from the baseline approach and the other from our StackSight approach. Accompanying these code samples, as presented in Figure 10, are three multiple-choice questions created based on source code. To address potential semantic variations between the source and generated codes, we include the option "I cannot tell based on the given code.". Participants' progress is visually restricted; a lock icon on the similarity section tab indicates that they must answer all questions before accessing the subsequent section.

Code Similarity

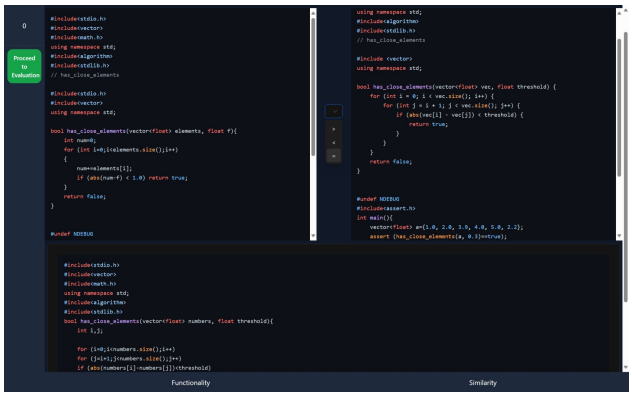

Figure 11. Similarity sample.

Figure 11 shows the section where participants compare two decompiled versions of code with the original, judging which decompiled code resembles the source more closely.

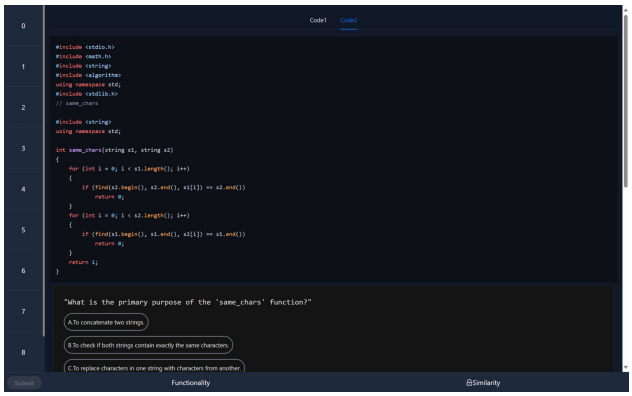

Figure 12. Evaluation page.

Then, in the actual evaluation page as shown in 12, each participant is provided with 10 sets of questions. The 'Submit' button remains inactive until all sections are completed, ensuring the completion of the study.