Can LLMs Consistently Describe Programs Across Source Code, Assembly, and Binary Representations?

Evaluating the Quality of Generated High-Level Descriptions of Benign and Malware Programs

Matīss Bērziņš[1]
Supervisors: Soham Chakraborty[1]**, Przemyslaw Pawelczak**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Matīss Bērziņš
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Przemyslaw Pawelczak, Arie van Deursen

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Large language models (LLMs) are increasingly used to summarize and reason about software artifacts. This is especially true in cybersecurity, where analysts must often interpret low-level code such as assembly or binary. If LLMs describe the same program differently depending on its representation, analysts may therefore receive inconsistent or incomplete explanations. This paper evaluates whether an LLM generates descriptions that are both consistent across representations of the same program and aligned with human reference descriptions. Using a balanced subset of SBAN, a dataset which provides aligned high-level source code, disassembled assembly, and a raw-hexadecimal binary representation of the same programs together with a natural-language reference, we generate high-level descriptions for every representation with Qwen3.5-2B using a fixed prompt and low-temperature stochastic decoding, repeated over five runs for 75000 descriptions in total. To prevent context-level reference leakage and support reproducibility, each description is generated independently, without conversation history, the reference description, dataset labels, or the other representations. We measure cross-representation consistency and reference alignment with complementary metrics: sentence-transformer cosine similarity and ROUGE-L over the full dataset, BERTScore against the references, and Prometheus, an independent LLM judge, on a fixed 600-sample subset. Source-code descriptions align best with the references and assembly–source descriptions are the most consistent, while binary–source is the least consistent. A Friedman test confirms a statistically significant representation effect on reference-based quality. The absolute differences are small, however, and cross-representation consistency is only moderate across all metrics. These results indicate that representation choice measurably affects both the quality and consistency of LLM-generated descriptions, likely because each representation exposes a different level of semantic information.

1 Introduction

Large language models (LLMs) are increasingly used to summarize and reason about software artifacts. In cybersecurity and reverse engineering, this includes artifacts at different abstraction levels, such as source code, assembly, and binary code. This raises a reliability problem: the same program may expose different information depending on its representation. Source code can include names, comments, types, and high-level structure, while assembly and binary representations expose lower-level operations and control flow but remove much of this semantic context.

Prior work shows that code summaries can depend on source-level cues such as identifiers and function names [1], and identifier-aware models such as CodeT5 treat identifiers as important semantic carriers [2]. It remains unclear, however, whether an LLM describes the same program consistently when the input changes from source code to assembly or binary. This matters because analysts often do not have access to source code, and inconsistent descriptions may cause relevant behaviour, such as persistence, file-system modification, or network communication, to be missed.

This paper investigates the following research question:

> *Can an LLM provide consistent high-level descriptions of the same program when represented as source code, assembly code, and binary code?*

We decompose this into two sub-questions:

SQ1 Are descriptions generated from different representations of the same program consistent with each other?

SQ2 Do those descriptions align with the reference descriptions provided by the SBAN dataset [3]?

The three representations form an abstraction gradient: source code exposes the most semantic cues (identifiers, types, comments, and high-level structure), disassembled assembly retains instruction mnemonics and recognizable calls but loses naming, and the raw-hexadecimal binary exposes the fewest. This motivates two directional hypotheses, which make the expectations implied above explicit and testable:

H1 (consistency) Cross-representation consistency decreases as the abstraction gap between two representations widens, so the assembly–source pair is the most consistent and the binary–source pair the least.

H2 (quality) Reference alignment decreases from source to assembly to binary, with source-code descriptions matching the references best.

The main contribution of this paper is a controlled, context-isolated evaluation of representation-induced variation in LLM-generated program behaviour descriptions. The study combines automatic metrics over 75000 generated descriptions with an LLM-as-a-judge evaluation on a fixed subset. The evaluation design prevents context-level reference leakage during generation, that is, the generator never sees the reference, the dataset labels, or the other representations of a program. It keeps the same programs aligned across all representations, and preserves the scripts and outputs needed to inspect the pipeline. This controls leakage within the prompt, but it does not, and cannot, rule out that the public SBAN programs or their descriptions were seen during model pre-training (Section 7). Our main finding is that representation choice measurably affects both how consistently the model describes a program and how well its descriptions match human references: descriptions are most consistent between assembly and source code and least consistent between binary and source code, and source-code descriptions align best with the references, although the absolute differences are small. All scripts, prompts, plotting code, logs, and final result files are available in the project repository: `https://github.com/JstBlood/CSE3000-Research-Project`. The raw dataset and model weights are not included.

2 Background and Related Work

2.1 LLMs for Code Summarization

Automatic code summarization generates natural-language descriptions of program behaviour. Prior work shows that LLM-generated summaries can depend on source-level cues such as identifiers and function names [1]. CodeT5 similarly highlights identifiers as semantic carriers in pre-trained code

models [2]. These findings suggest that source-code summaries may benefit from information unavailable in assembly or binary code. However, this work mainly studies source-level summarization, not whether descriptions remain stable across representations of the same program.

2.2 Assembly and Binary Code Representation

Assembly and binary representations expose low-level instructions, registers, memory accesses, and control flow, while removing names, comments, types, and other source-level structure. Prior work reflects the need for specialized modelling at this level. PalmTree learns instruction embeddings for assembly language [4], while jTrans uses jump-aware Transformer modelling for binary code similarity [5]. These works address low-level code representation, but not whether an LLM produces equivalent natural-language descriptions for source, assembly, and binary versions of the same program.

2.3 Evaluation of Generated Descriptions

Generated descriptions can be difficult to evaluate because similar meanings may be expressed with different wording. ROUGE-L measures lexical overlap using longest common subsequence matching [6], while BERTScore uses contextual embeddings to compare generated and reference text [7]. ROUGE-L captures surface similarity, whereas BERTScore is more tolerant of paraphrasing. However, both may miss program-specific behavioural differences, especially in security contexts. SimLLM makes a similar observation for code summaries and motivates LLM-based semantic similarity evaluation [8]. This paper therefore uses multiple metrics rather than treating any single score as ground truth.

2.4 LLM-as-a-Judge Evaluation

LLM-as-a-judge methods provide flexible rubric-based evaluation of generated text. G-Eval proposes structured GPT-based evaluation for natural-language generation [9], and Prometheus 2 is an open evaluator model for direct assessment and pairwise ranking [10]. This paper uses Prometheus as an independent evaluator, separate from the generator model. However, LLM judges can exhibit position bias, verbosity bias, and self-enhancement bias [11]. Prometheus scores are therefore treated as complementary evidence alongside lexical and embedding-based metrics.

2.5 Positioning of This Work

Table 1 positions this work against the most relevant prior approaches. Existing work either summarizes code at a single abstraction level or models low-level code for tasks other than natural-language description, and evaluation work targets generated-text quality in general rather than cross-representation stability. This study instead holds the program fixed and varies only its representation, then measures both inter-representation consistency and reference alignment.

Work	Input level	Task	Cross-rep.?
LLM code summarization [1]	Source	Summarization	No
CodeT5 [2]	Source	Generation / understanding	No
PalmTree [4]	Assembly	Instruction embedding	No
jTrans [5]	Binary	Code similarity	No
G-Eval [9], Prometheus [10]	Text	LLM-as-judge evaluation	No
This study	**Source + assembly + binary**	**Description consistency & quality**	**Yes**

Table 1: Qualitative comparison with related work. "Input level" is the program abstraction level consumed. "Cross-rep.?" indicates whether the work compares natural-language descriptions across source, assembly, and binary representations of the *same* program.

3 Methodology

3.1 Representation-Induced Variation

A program has a single underlying behaviour, yet its source, assembly, and binary representations expose different semantic cues that prior work shows can shift generated summaries [1, 2, 4, 5]. We formalize this representation-induced variation as follows.

Let $P = \{p_1, \ldots, p_N\}$ be a set of programs. For each program p_i, the dataset provides three aligned representations r_i^b, r_i^a, and r_i^s, denoting binary, assembly, and source code respectively, as well as a reference description n_i. A generator model M, given a prompt P_{gen}, produces:

$$d_{i,j}^x = M(P_{\text{gen}}, r_i^x),$$

where $x \in \{b, a, s\}$ and $j \in \{1, \ldots, 5\}$ is the generation run.

The core problem is that $d_{i,j}^b$, $d_{i,j}^a$, and $d_{i,j}^s$ should ideally describe the same behaviour, because they refer to the same program. This study operationalizes the research question through two measurements: cross-representation consistency,

$$\text{sim}(d_{i,j}^b, d_{i,j}^a), \quad \text{sim}(d_{i,j}^b, d_{i,j}^s), \quad \text{sim}(d_{i,j}^a, d_{i,j}^s),$$

and reference-based quality,

$$\text{sim}(d_{i,j}^x, n_i).$$

These are distinct: descriptions can agree with each other while both miss the reference behaviour, or one representation can align with the reference while disagreeing with the others.

3.2 Dataset and Representations

SBAN is selected because it is, to our knowledge, the only readily available dataset that provides aligned binary, assembly, source-code, and natural-language-description fields for the *same* programs [3], which is exactly the alignment required to vary representation while holding behaviour fixed.

The study uses a 5000-sample SBAN evaluation subset, with samples matched by program identifier. The experiment keeps only samples for which all three input representations and the reference description are available.

Each representation is consumed as the text that SBAN provides for it, since a text LLM cannot ingest raw executable bytes directly. Concretely, the *source* field is the high-level source code, the *assembly* field is the disassembled instruction listing (mnemonics and operands), and the *binary* field is the Raw Hexadecimal Representation (RHR) of the compiled executable used by SBAN [3], that is, the program serialized as a hexadecimal byte string rather than a strings dump or a separate disassembly. Throughout the paper, "binary" refers to this hexadecimal byte projection, and "assembly" to the disassembly. The two are distinct SBAN layers and are never merged.

The subset is balanced between benign and malware-labeled samples. These labels are used only for sampling and are not included in generation or judge prompts. Since each program is evaluated under three representations and five repeated generation runs, the generation stage produces 75000 descriptions:

$$5000 \times 3 \times 5 = 75000.$$

3.3 Context-Isolated Description Generation

Description generation is performed with Qwen3.5-2B, an open-weights instruction-tuned model [12]. It is chosen because its open weights make the pipeline fully reproducible, and because a 2B-parameter model is small enough to generate all 75000 descriptions within the four-hour GPU wall-time limit available on the cluster (Section 4) while still following the formatting constraints of the prompt. The same prompt is used for all representations to isolate the effect of representation rather than prompt variation. The prompt contains only the current representation text and excludes the reference description, malware/benign label, dataset metadata, sample identifier, and the other two representations. The exact prompt is listed in Appendix A.

Both the raw model output and the cleaned generated description are stored, which keeps the output auditable. Generation is repeated five times with controlled stochastic decoding using temperature 0.30, top-p 0.90, repetition penalty 1.05, and a maximum of 160 new tokens, which is sufficient for the single-sentence descriptions the prompt requests. The representation text is passed in full: the pipeline applies no truncation and instead aborts if a prompt plus the 160-token generation budget would exceed the model context window, so every retained sample is one whose three representations each fit within the context window without truncation. Each generation call is independent and receives no conversational history.

3.4 Cross-Representation Consistency

To answer SQ1, descriptions generated for different representations of the same program and generation run are compared pairwise. Sentence-transformer cosine similarity [13] measures semantic proximity, while ROUGE-L provides a lexical-overlap baseline [6]. SQ1 inputs contain only generated descriptions and representation labels, reference descriptions are excluded.

Prometheus-7B-v2.0 is used as a complementary rubric-based judge. It is chosen because it is an open evaluator model purpose-built for rubric-based assessment, and because it is a different model family from the Qwen generator, which reduces same-model self-preference bias. Because LLM judging is substantially more expensive than automatic metrics, it is applied to a fixed random subset of 600 samples from generation run 1. This produces 1800 SQ1 Prometheus judgments:

$$600 \text{ programs} \times 3 \text{ representation pairs}.$$

Automatic SQ1 metrics are still computed over the full generated dataset.

Prometheus uses a feedback-first scoring design. The judge first writes brief rubric-based feedback without assigning a score. The script then scores constrained decimal candidates:

$$S = \{1.0, 1.1, \ldots, 5.0\}.$$

A prior prompt with omitted descriptions estimates baseline score preference, and this prior is subtracted from the candidate log-likelihoods:

$$\tilde{\ell}_s = \ell_s - \ell_s^{\text{prior}}.$$

The calibrated likelihoods are converted to probabilities,

$$p_s = \frac{\exp(\tilde{\ell}_s)}{\sum_{k \in S} \exp(\tilde{\ell}_k)},$$

and the reported score is the expected value:

$$E[s] = \sum_{s \in S} s p_s.$$

This preserves uncertainty across nearby rubric values. The hard maximum-probability score is retained only for inspection. The exact SQ1 judge prompts are listed in Appendix B.

3.5 Reference-Based Quality

To answer SQ2, each generated description is compared with the SBAN reference description for the same sample. BERTScore precision, recall, and F1 are computed over the full generated dataset because BERTScore is more tolerant of paraphrasing than exact lexical overlap [7]. BERTScore F1 is treated as the main automatic reference-based metric.

Prometheus is also used for SQ2 reference alignment. It is applied to the same fixed 600-sample subset from generation run 1 as SQ1, producing 1800 SQ2 judgments:

$$600 \text{ programs} \times 3 \text{ representations}.$$

This score measures alignment with the reference description, not independently verified behavioural correctness from the original program. The exact SQ2 judge prompts are listed in Appendix C.

3.6 Aggregation and Statistical Analysis

Automatic metrics are first computed at the generation-run level and then averaged per program before plotting and reporting summary statistics. For SQ1, each program contributes

one averaged score per representation pair. For SQ2, each program contributes one averaged score per representation. This avoids treating repeated generations as independent samples.

Because each program is evaluated under all three representations, representation-level quality scores are paired. The Friedman test is used to test for an overall representation effect over BERTScore F1 after averaging repeated generations per program and representation [14]. Because a large sample size can make even negligible differences significant, statistical significance is always interpreted together with the effect size (Kendall's W), absolute score differences, and the distributional plots.

Pair	Cos.	Cos. SD	R-L	R-L SD
Binary–Assembly	0.481	0.099	0.240	0.048
Binary–Source	0.374	0.094	0.192	0.035
Assembly–Source	**0.562**	0.114	**0.244**	0.057

Table 2: Cross-representation consistency between generated descriptions. Cos. is cosine similarity and R-L is ROUGE-L. SD is the standard deviation. Scores are computed per generation run and averaged per program (Section 3.4).

Pair	Mean	SD	Median	Count
Binary–Assembly	2.79	0.14	2.78	600
Binary–Source	2.75	0.12	2.72	600
Assembly–Source	**2.88**	0.16	2.88	600

Table 3: Prometheus cross-representation consistency scores on a 600-sample subset. Values are scored on a 1.0–5.0 scale. SD is the standard deviation and Count is the number of judged programs (Section 3.4).

Representation	Precision	Recall	F1
Binary	0.847	0.863	0.855
Assembly	0.856	0.883	0.869
Source	**0.865**	**0.897**	**0.881**

Table 4: Reference-based description quality measured with BERTScore against the SBAN reference descriptions. Scores are averaged per program over five generation runs (Section 3.5).

4 Experimental Setup

4.1 Hardware and Software

GPU jobs run on the DelftBlue cluster, TU Delft's institutional supercomputer. Each generation and judging job is a Slurm array task on the `gpu-a100-small` partition, requesting one GPU, 2 CPU cores, and 16 GB of system memory, under a four-hour wall-time limit. This partition provides a slice of an NVIDIA A100 with roughly 9.5 GB of GPU memory, which is the GPU budget that constrains the model and judge choices below. This four-hour limit is the main practical constraint that motivates incremental, resumable processing throughout the pipeline. The GPU environment is managed with `miniforge3`/Conda and uses PyTorch with the Hugging Face `transformers` library [15]. Prometheus is loaded in 4-bit precision (`bitsandbytes`, `bfloat16` compute) [16] so that the 7B judge fits on a single A100. The local metric stage uses Python with exact libraries and versions listed in the requirements files in the repository. Sentence-transformer cosine similarity uses `all-MiniLM-L6-v2` [13], BERTScore uses `roberta-large` (English) [17], and ROUGE-L is computed as a longest-common-subsequence F1 over whitespace tokens.

4.2 Generation and Validation

The generation stage processes 5000 programs under three representations and five repeated runs, with run j using seed $1000 + j$. Outputs are saved incrementally so interrupted jobs resume without regenerating completed descriptions. Each final JSONL file contains 15000 descriptions and the complete output contains 75000. Each row stores the raw model output, cleaned description, representation metadata, sample metadata, and generation status. After generation, validation checks that each run contains the expected sample–representation pairs, that no generated descriptions or references are empty, and that the same reference description is used for all three representations of a sample. Separate metric inputs are then built for SQ1 and SQ2 so that reference descriptions cannot leak into the cross-representation analysis.

4.3 Prometheus Judging

Prometheus judging runs as a resumable batch process on the fixed 600-sample subset from generation run 1. The SQ1 judge input contains 1800 description pairs and the SQ2 judge input contains 1800 generated-description/reference pairs, for 3600 judgments in total. Feedback is decoded with temperature 0.20, top-p 0.90, and at most 128 tokens, after which the 41 decimal score candidates are scored in batches. Inputs are split into chunks processed as Slurm array jobs, and each chunk writes its own JSONL output so timed-out jobs resume safely. Each output row stores the original judge input, metadata, judge prompt version, seed, brief rubric-based feedback, calibrated expected score, hard maximum-probability score, score-bucket probabilities, and the full decimal score probability distribution. The main analysis uses the calibrated expected score. All scripts, prompts, Slurm files, logs, and final metric outputs are released in the repository (Appendix D).

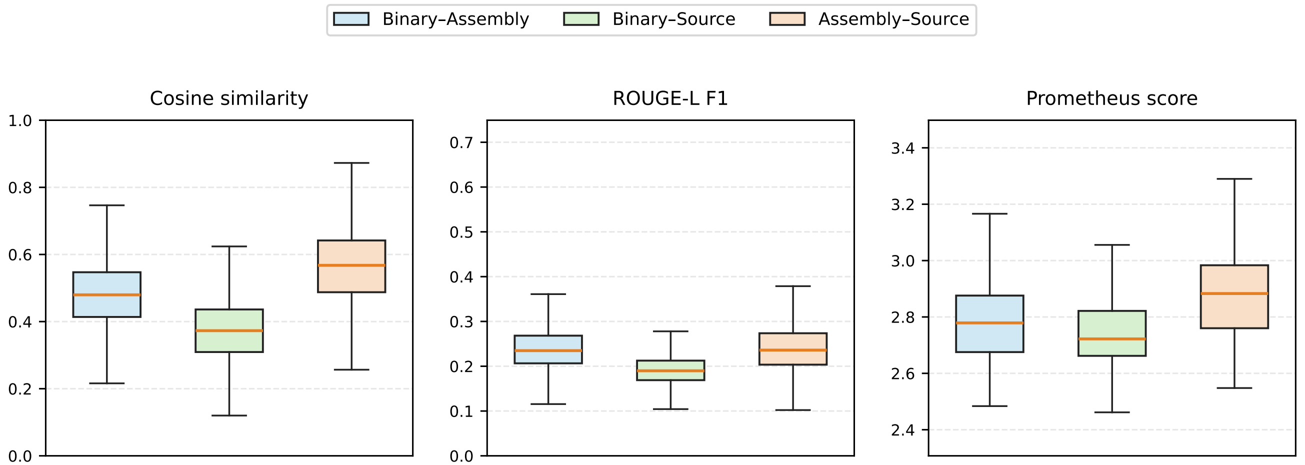

Figure 1: Cross-representation consistency (SQ1) distributions between generated descriptions, by representation pair, for sentence-transformer cosine similarity, ROUGE-L (longest-common-subsequence F1), and the Prometheus calibrated expected score (1.0–5.0 rubric scale). The automatic metrics aggregate per-program scores averaged over the five generation runs on the full dataset, while Prometheus is computed on the fixed 600-sample subset (Section 3.4).

Representation	Mean	SD	Median	Count
Binary	2.79	0.22	2.70	600
Assembly	2.99	0.24	3.05	600
Source	**3.09**	0.19	3.17	600

Table 5: Prometheus reference-alignment (SQ2) scores on the fixed 600-sample subset, by representation. Values are calibrated expected scores on a 1.0–5.0 scale. SD is the standard deviation and Count is the number of judged programs (Section 3.5).

Representation	Mean F1	SD	Mean rank
Binary	0.855	0.018	1.22
Assembly	0.869	0.019	2.09
Source	**0.881**	0.018	**2.69**
Friedman $\chi^2 = 5456.91$, $p < 0.001$, $N = 5000$, $W = 0.55$			

Table 6: Friedman test for a representation effect over F1 values, averaged per program across the five generation runs (Section 3.6).

5 Results

5.1 Cross-Representation Similarity

Table 2 reports the full-dataset SQ1 metrics, and Figure 1 shows their distributions. Table 3 reports Prometheus SQ1 scores on the fixed 600-sample subset.

On the fixed 600-sample subset, the Prometheus SQ1 scores (Table 3, Figure 1) show that the Binary–Source pair has the lowest cross-representation consistency (mean 2.75), while the Assembly–Source pair has the highest (mean 2.88), the Binary–Assembly pair lies between them (mean 2.79). All three means cluster near the middle of the 1.0–5.0 rubric scale, indicating only partial agreement between descriptions generated from different representations, and that representation-induced variation is strongest between binary and source code. The full-dataset automatic metrics in Table 2 and Figure 1 reproduce this ordering on all 5000 programs and make the gap clearer: sentence-transformer cosine similarity is highest for Assembly–Source (0.56), intermediate for Binary–Assembly (0.48), and lowest for Binary–Source (0.37), and ROUGE-L follows the same ordering, with Binary–Source again clearly lowest (0.19 versus 0.24 for the other two pairs). The embedding metric separates the pairs far more sharply than Prometheus does, which is expected because cosine similarity reacts directly to the divergent surface content of binary-derived descriptions, whereas the rubric-anchored judge compresses its scores toward the middle of the scale.

5.2 Reference-Based Description Quality

Table 4 reports the full-dataset BERTScore results against the SBAN reference descriptions, and Figure 2 shows their distributions.

Table 5 reports Prometheus SQ2 reference-alignment scores on the fixed 600-sample subset. The Prometheus SQ2 scores (Table 5, Figure 3) show that Source descriptions achieve the highest reference alignment (mean 3.09), followed by Assembly (mean 2.99) and Binary (mean 2.79). The absolute differences are small (a 0.30-point spread on the 1.0–5.0 scale), suggesting that source code aligns best with the reference descriptions while leaving only a measurable but modest gap over the lower-level representations. The full-dataset BERTScore results in Table 4 and Figure 2 confirm the same ranking on all 5000 programs: Source has the highest F1 (0.881), followed by Assembly (0.869) and Binary (0.855). The gain comes mainly from recall (Binary 0.863 rising to Source 0.897) rather than precision, indicating that source-based descriptions cover more of the reference content while all three representations stay similarly on-topic. The absolute BERTScore differences are small, partly because BERTScore assigns a high baseline similarity to fluent English text, so the consistent direction of the effect matters more than its magnitude.

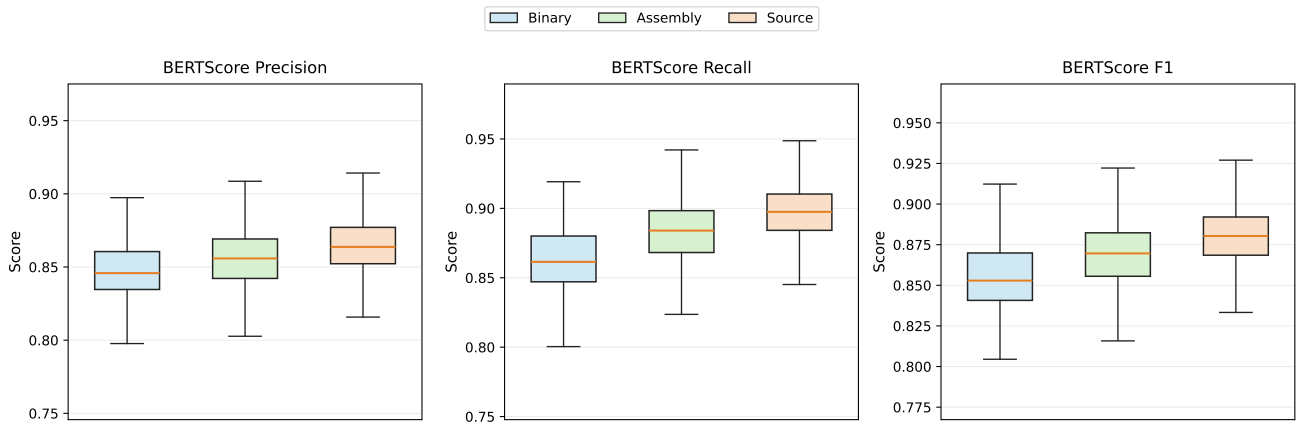

Figure 2: Reference-based quality (SQ2) distributions, by representation, for BERTScore precision, recall, and F1 against the SBAN reference descriptions. Each box aggregates per-program scores averaged over the five generation runs; see Section 3.5.

5.3 Statistical Test Results

Table 6 reports the Friedman test over the paired, per-program BERTScore F1 values.

The Friedman test indicates a statistically significant representation effect on BERTScore F1 ($\chi^2(2) = 5456.91$, $p < 0.001$), with mean ranks ordering the representations as source (2.69) $>$ assembly (2.09) $>$ binary (1.22). The effect size is moderate-to-large (Kendall's $W = 0.55$), so the ordering is consistent across programs. Since the sample size is large, this result is interpreted together with the small absolute score differences (Table 4) and the distributional plots: the representation effect is consistent and well-ordered, but modest in magnitude.

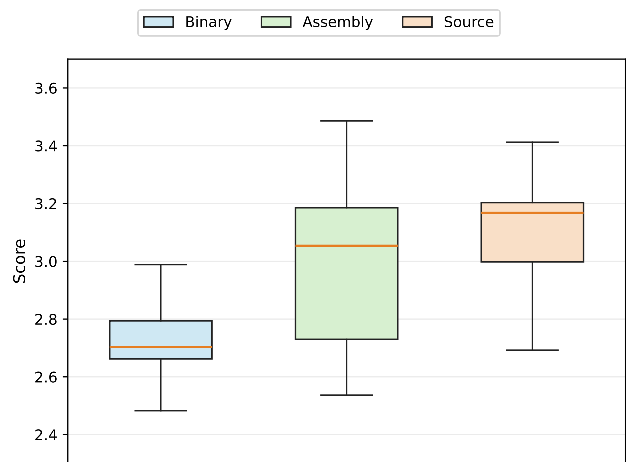

Figure 3: Prometheus reference-alignment (SQ2) scores on the fixed 600-sample subset, by representation. Scores are calibrated expected values on a 1.0–5.0 rubric scale (Section 3.5).

6 Responsible Research

6.1 Ethical Considerations

This work uses a malware-analysis dataset, which makes the research dual-use. The study is designed to evaluate descriptions of program behaviour, not to improve malware generation, evasion, or deployment. The prompts ask for high-level behavioural descriptions and do not ask the model to classify, modify, exploit, or execute programs. The results should therefore be interpreted as evidence about explanation reliability, not as an operational malware-analysis tool.

The study uses an existing research dataset and does not collect personal data. The generated outputs are summaries of program representations rather than user information. Still, LLM-generated descriptions can be misleading if used without expert review, especially in cybersecurity contexts where incorrect summaries may cause analysts to miss relevant behaviour.

6.2 Reproducibility

Reproducibility is supported by fixing the evaluation subset and seeds, using a single shared prompt, and validating every generated output before metric computation (Sections 3–4). The prompts, scripts, and result files are released (Appendix D). Because the same 5000 programs are evaluated under all three representations, the comparison is paired rather than across unrelated samples, and the context-isolation controls described above also serve as validity controls. The raw dataset and model weights are excluded for storage and licensing reasons, so a reproducer must supply them separately. The main residual threat to exact reproduction is that LLM inference depends on model versions, hardware, and quantization. The Prometheus judge in particular is quantized to 4-bit to fit a single GPU, which is one reason its scores are reported only as complementary evidence.

6.3 Use of Generative AI

Generative AI tools were used during both writing and implementation, and their role is described here for transparency. For writing, large language models (accessed through ChatGPT) were used mainly to revise existing author-written text for grammar, sentence structure, conciseness, and clarity. For implementation, ChatGPT was used to debug and revise the

scripts for generation, metric computation, validation, plotting, and Prometheus scoring. All generated code was read, tested, and corrected by the author, and the experimental logic and parameters were set deliberately rather than accepted blindly.

No generative AI tool was used as an unverified source of scientific claims. Claims about related work are supported by cited literature, and all experimental results are derived from the described pipeline.

7 Discussion

7.1 Representation Effects

The overall SQ1 scores show that the Binary–Source pair has the lowest cross-representation consistency, while the Assembly–Source pair has the highest. This ordering is what one would expect if consistency tracks how much semantic information two representations share, because the three representations form a clear abstraction gradient. Source code is the richest, exposing identifiers, types, comments, and high-level structure. Assembly is intermediate: it discards source-level naming but still preserves human-readable instruction mnemonics, recognizable library and system calls, and explicit control flow, so it retains many of the behavioural cues that source code also makes available. The binary is the poorest: as a raw hexadecimal byte string it exposes almost no symbolic context, reducing largely to opaque byte values from which behavioural cues must be inferred indirectly.

It therefore makes sense that the Assembly–Source pair is the most consistent: assembly and source sit adjacent on this gradient and expose overlapping behavioural cues, so the model tends to infer similar high-level behaviour from both. By the same logic, Binary–Source is the least consistent because binary and source are the two endpoints of the gradient, separated by the largest gap in available semantic information. With little symbolic context to anchor the binary description, the model is more likely to produce a generic or divergent summary that disagrees with the source-derived one. In other words, representation-induced variation grows with the abstraction distance between representations, which is strongest between binary and source code.

For reference-based quality, the overall SQ2 scores order the representations as source $>$ assembly $>$ binary. This supports the expectation that source code exposes identifiers and high-level structure that help the model infer program intent. The margins are small (within roughly 0.3 points on the rubric scale), so the result should be interpreted as a measurable but not uniformly large representation effect.

7.2 Qualitative Example

A concrete example from the judged subset illustrates why these scores stay near the middle of the scale rather than approaching the top. For one program, the source-derived description is specific and behaviour-rich: *"The function checks if a data source exists at a specified address and attempts to invoke a custom handler via a library function. If the source is valid, it calls another internal function to examine mailboxes, otherwise it returns an error code."* This detail is possible because source code preserves the identifiers and structure naming the handler, the mailbox-examining routine, and the error path. The assembly-derived description keeps the right shape but loses the naming: *"... initializes a memory buffer, performs a series of system calls including file operations and library invocations, and then cleans up allocated resources... "*. It still recovers the library calls and cleanup that overlap with source, but "mailboxes" has flattened into a generic "library invocation". The binary-derived description is the most generic and hedged: *"... execute a series of Windows API calls and memory manipulations, likely involving process creation, service registration, or network communication... "*. With the symbols stripped, the model can only enumerate plausible Windows behaviours, none matching the specific mailbox check. The judge consequently rates Assembly–Source higher, since both mention library calls and cleanup, than Binary–Source, which pits a specific behaviour against an unrelated generic guess. This mirrors the aggregate scores and shows that the moderate ratings stem less from random noise than from a systematic loss of semantic detail as the representation moves away from source code.

7.3 Prometheus Judge Interpretation

Prometheus scores are rubric-based evidence, not ground truth: the judge is a model rather than a human analyst and can exhibit known LLM-judge biases [11]. Because they are computed on a fixed 600-sample subset and on a smoothed expected-value scale, they complement, but do not replace, the full-dataset automatic metrics.

7.4 Implications for Malware Analysis

For malware and reverse-engineering workflows, these results suggest that LLM-generated descriptions should be interpreted as representation-dependent. A description generated from binary or assembly may still be useful, but it may emphasize different behaviour than a source-based description. Agreement between descriptions from different representations can increase confidence, while disagreement can indicate that the model is relying on representation-specific cues or missing relevant behaviour.

7.5 Metric Agreement

Each metric captures a different facet of similarity, so agreement across them is more informative than any single score. All three SQ1 metrics agree on the ordering: sentence-transformer cosine, ROUGE-L, and Prometheus all rank Assembly–Source as the most consistent pair and Binary–Source as the least, and the SQ2 metrics (BERTScore F1 and Prometheus) both rank source $>$ assembly $>$ binary. The metrics disagree mainly in how strongly they separate the representations: the embedding-based cosine similarity shows the widest spread, ROUGE-L a narrower one, and the Prometheus rubric the narrowest, because lexical overlap and the rubric-anchored judge are both less sensitive than embeddings to the surface divergence of binary-derived descriptions. This convergence across independent metrics increases confidence that the representation effect is real rather than an artefact of any single measure. The distributional plots matter because mean values alone hide the substantial program-level variation in how consistently the model describes different representations.

7.6 Limitations

This study has several limitations. Many of them stem from a constrained compute budget: all GPU work ran on a shared cluster under a four-hour wall-time limit on a single A100 per job (Section 4), which directly shaped three design choices. First, the generator is a relatively small 2B-parameter model, Qwen3.5-2B, chosen because it could describe all 75000 representations within budget. The results may therefore not generalize to larger or more capable models. Second, the Prometheus judge had to be quantized to 4-bit and applied to only a fixed 600-sample subset rather than the full dataset, which is why its scores are reported as complementary evidence rather than as a primary metric. Third, reference-based quality is measured against the SBAN natural-language descriptions, which are treated as a proxy for quality rather than as perfect behavioural ground truth, and automatic text metrics may miss security-relevant behavioural differences even when descriptions appear semantically similar.

A further limitation concerns validity rather than budget. The context isolation in Section 3.3 removes prompt-level leakage, but it cannot address *data contamination*: SBAN is a public dataset, so we cannot exclude that the generator or the judge encountered these programs, or their descriptions, during pre-training. Contamination is a recognized and difficult-to-eliminate threat in public-benchmark LLM evaluation, and simple decontamination heuristics are often insufficient, so our claims are framed as relative comparisons across representations of the same program rather than as absolute quality measurements.

These results are also drawn from a single generator model, a single judge model, and a single dataset, so they should be read as evidence about this setting rather than as a general law about all LLMs. Together, these constraints affect the strength and generality of the conclusions, but not their direction within this setting: the representation effect is consistent across every metric we computed.

8 Conclusion and Future Work

This paper asked whether an LLM describes the same program consistently across its source-code, assembly, and binary representations (SQ1), and how well those descriptions match human references (SQ2). Using a controlled, context-isolated pipeline, Qwen3.5-2B produced 75000 descriptions over 5000 programs, evaluated with cosine similarity, ROUGE-L, BERTScore, and an independent LLM as a judge.

Conclusions. SQ1. Descriptions are only moderately consistent and clearly not representation-invariant: every metric ranks the Assembly–Source pair most consistent and Binary–Source least, so agreement falls as the abstraction gap widens, supporting hypothesis H1. **SQ2.** Source-code descriptions align best with the references, ahead of assembly and binary, a difference the Friedman test confirms as significant ($\chi^2(2) = 5456.91$, $p < 0.001$, $W = 0.55$) but small in absolute terms, supporting hypothesis H2. **Main question.** The evaluated model does not *fully* describe a program consistently across representations: representation choice affects both inter-representation agreement and reference alignment, most plausibly because lower-level code exposes fewer semantic cues. While these results come from a single 2B model, judge, and dataset and should not be over-generalized, the practical implication is that LLM summaries of binaries or assembly should be treated as representation-dependent, and disagreement across representations is a useful signal to inspect a program more closely.

Future work. The study was constrained by a four-hour wall-time limit and a single GPU with roughly 9.5 GB of memory, which forced a small 2B model and limited the LLM judge to 600 programs. The most valuable next steps are to repeat it with large, recent state-of-the-art models (both frontier and code-specialized) and to run every stage, including the LLM judge, over the full dataset of close to 3 million samples. Further directions include adding human expert judgments, testing representation-specific prompts, and analysing which behaviours are most often lost when describing assembly or binary.

Acknowledgements

I thank my research project group as well as the responsible professors and supervisors for their feedback and support throughout the project.

References

[1] R. Haldar *et al.*, “Analyzing the Performance of Large Language Models on Code Summarization,” 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.08018

[2] Y. Wang *et al.*, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2109.00859

[3] H. Jelodar *et al.*, “SBAN: A Framework & Multi-Dimensional Dataset for Large Language Model Pre-Training and Software Code Mining,” 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2510.18936

[4] X. Li *et al.*, “PalmTree: Learning an Assembly Language Model for Instruction Embedding,” 2021. [Online]. Available: https://doi.org/10.1145/3460120.3484587

[5] H. Wang *et al.*, “jTrans: Jump-Aware Transformer for Binary Code Similarity,” 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.12713

[6] C.-Y. Lin, “ROUGE: A Package for Automatic Evaluation of Summaries,” 2004. [Online]. Available: https://aclanthology.org/W04-1013/

[7] T. Zhang *et al.*, “BERTScore: Evaluating Text Generation with BERT,” 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1904.09675

[8] X. Jin *et al.*, “SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language

Model-Based Approach," 2024. [Online]. Available: https://doi.org/10.1145/3660769

[9] Y. Liu *et al.*, "G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.16634

[10] S. Kim *et al.*, "Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models," 2024. [Online]. Available: https://doi.org/10.18653/v1/2024.emnlp-main.248

[11] L. Zheng *et al.*, "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.05685

[12] A. Yang *et al.*, "Qwen3 Technical Report," 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.09388

[13] N. Reimers *et al.*, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," 2019. [Online]. Available: https://doi.org/10.18653/v1/D19-1410

[14] J. Demšar, "Statistical Comparisons of Classifiers over Multiple Data Sets," 2006. [Online]. Available: https://www.jmlr.org/papers/v7/demsar06a.html

[15] T. Wolf *et al.*, "Transformers: State-of-the-Art Natural Language Processing," 2020. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-demos.6

[16] T. Dettmers *et al.*, "The Case for 4-bit Precision: k-bit Inference Scaling Laws," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2212.09720

[17] Y. Liu *et al.*, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1907.11692

A Description Generation Prompt

This prompt is used for all source, assembly, and binary description generation. Only the current representation text is inserted into the prompt. The prompt excludes the reference description, malware/benign label, sample identifier, dataset metadata, and the other two representations.

```
You will be given one program representation. The representation may be source code, assembly code, or binary/
     disassembly text.

Task:
Write one concise natural-language description of what the program or function does.

Strict output rules:
- Return only the final description.
- Do not include reasoning, analysis, explanations, bullet points, labels, or extra text.
- Do not output <think>, Thinking Process, or any hidden reasoning.
- One sentence only.
- Start with "The code" or "The function" when appropriate.
- Describe the main high-level behaviour, not the implementation line by line.
- Focus on actions such as file operations, registry access, process creation, service handling, networking, memory/
     string manipulation, configuration handling, or control-flow behaviour.
- Include important conditions when they affect the behaviour, for example: if a file exists, if a registry key is
     missing, if a process is found, or if an input is invalid.
- Mention important effects, such as creating files, writing registry values, downloading files, starting processes,
     modifying strings, allocating memory, or cleaning up resources.
- Do not over-generalize if a more specific behaviour is supported by the input.
- Do not describe irrelevant low-level details such as register names, stack offsets, temporary variables, addresses,
      labels, or compiler artifacts.
- Do not classify the program as malware, benign, suspicious, safe, harmful, or malicious.
- Do not mention the representation type.
- Do not invent behaviour that is not supported by the input.
- If the exact purpose is unclear, give the most specific cautious description supported by the code, using wording
     such as "appears to" only when necessary.

Program input:
{representation_text}
```

B SQ1 Prometheus Cross-Representation Judge Prompt

This prompt evaluates semantic similarity between two generated descriptions of the same program from different input representations. Prometheus first writes brief rubric-based feedback without assigning a score. The script then computes calibrated constrained likelihoods over decimal scores from 1.0 to 5.0.

SQ1 Base Rubric Prompt

```
###Task:
Rate the semantic similarity of two generated high-level program behaviour descriptions for the same program.

###Description A:
{description_a}

###Description B:
{description_b}

###Evaluation focus:
Compare the described program behaviour, not exact wording.
Consider whether both descriptions describe the same main action, affected object, condition, input/output, and
     effect.
Relevant behaviours may include file operations, registry access, process creation, service handling, networking,
     memory or string manipulation, configuration handling, cleanup, or control-flow behaviour.
Do not penalize harmless paraphrasing.
Penalize changed behaviour, missing main behaviour, unsupported extra behaviour, contradictions, and overly generic
     descriptions.
Use the full 1.0--5.0 scale. Choose the score that best reflects the rubric, using one decimal place when the quality
      falls between two rubric levels.

###Rubric:
1.0 = Different behaviour: the descriptions are unrelated, contradictory, or describe different main actions.
2.0 = Weak overlap: the descriptions share a small topic or object, but the main behaviour or effect is different.
3.0 = Partial similarity: the descriptions describe related behaviour, but one misses or changes an important action,
      condition, object, or effect.
4.0 = Mostly equivalent: the descriptions describe the same main behaviour, with only minor missing details or
     harmless abstraction differences.
5.0 = Equivalent: the descriptions express the same behaviour, including the main action and important effect, with
     no meaningful contradiction or unsupported extra behaviour.
```

SQ1 Feedback Prompt

```
###Instruction:
Write brief feedback explaining the comparison according to the rubric.
Mention only the most important reason for the score.
Do not give a numeric score yet.
Do not add information not present in the descriptions.

Feedback:
```

SQ1 Score Prompt

```
###Feedback:
{judge_feedback}

###Instruction:
Based on the rubric and feedback, choose one numeric score from 1.0 to 5.0.
The score may use at most one decimal place.
Return only the score.

[RESULT]
```

C SQ2 Prometheus Reference-Alignment Judge Prompt

This prompt evaluates how well a generated description matches the SBAN natural-language reference description. The resulting score is interpreted as reference alignment, not independently verified behavioural correctness.

SQ2 Base Rubric Prompt

```
###Task:
Rate how well a generated program behaviour description matches a reference program behaviour description.

###Generated description:
{generated_description}

###Reference description:
{reference_nld}

###Evaluation focus:
Use the reference description as the grading target.
Compare program behaviour, not exact wording.
Check whether the generated description captures the same main action, affected object, condition, input/output, and
     effect as the reference.
Relevant behaviours may include file operations, registry access, process creation, service handling, networking,
     memory or string manipulation, configuration handling, cleanup, or control-flow behaviour.
Do not penalize harmless paraphrasing.
Penalize missing important behaviour, changed behaviour, unsupported added behaviour, contradictions, and overly
     generic descriptions.
Do not reward a description for being plausible if the behaviour is not supported by the reference.
Use the full 1.0--5.0 scale. Choose the score that best reflects the rubric, using one decimal place when the quality
      falls between two rubric levels.

###Rubric:
1.0 = Incorrect: unrelated, contradictory, meaningless, or describes a different main behaviour than the reference.
2.0 = Weak match: mentions a similar topic or object, but misses or changes the main action or effect.
3.0 = Partial match: captures part of the reference behaviour, but omits, adds, or changes an important action,
     condition, object, or effect.
4.0 = Mostly correct: captures the main reference behaviour, with only minor omissions, harmless abstraction, or
     small wording differences.
5.0 = Equivalent: semantically matches the reference behaviour, including the main action and important effect, with
     no unsupported additions or contradictions.
```

SQ2 Feedback Prompt

```
###Instruction:
Write brief feedback explaining the comparison according to the rubric.
Mention only the most important reason for the score.
Do not give a numeric score yet.
Do not add information not present in the descriptions.

Feedback:
```

SQ2 Score Prompt

```
###Feedback:
{judge_feedback}

###Instruction:
Based on the rubric and feedback, choose one numeric score from 1.0 to 5.0.
The score may use at most one decimal place.
Return only the score.

[RESULT]
```

D Repository and Reproduction Artifacts

The accompanying repository contains the scripts, prompts, plotting code, logs, and final result files used in the experiment:

`https://github.com/JstBlood/CSE3000-Research-Project`

The repository excludes the raw SBAN dataset and model weights. A reproducer must provide the prepared dataset and model files separately, or adapt the dataset and model paths in the scripts.