MULTI-LANGUAGE NEURAL DECOMPILATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Deterministic decompilers and state of the art neural decompilers often support binary to C translation yet there is a lack of decompilers for modern high level languages (HLLs) such as Swift, Go and Rust. Current benchmarks for decompilation fixate on binary to C decompilation at the function level, limiting the realistic decompilation tasks to evaluate decompilers on. We investigate using LLMs to translate binaries at the repository level for multiple HLLs, and introduce BLANKSPACEBENCH, a multi-language decompilation benchmark. We also develop BLANKSPACE, a multi-language agentic decompilation framework, which uses existing binary analysis tools to translate binaries from multiple instruction set architectures (ISAs) to HLLs, with support for coding agents Claude Code and Codex. We compare BLANKSPACE, frontier models, and deterministic decompilers on top open source projects, evaluating decompilation for arm64 to Swift, x86-64 to C, and x86-64 to Go.

1 INTRODUCTION

Decompilation is the process of translating an executable file back to source code (Cifuentes & Gough), often used in reverse engineering tasks such as analyzing malicious software (Yakdan et al., 2016), auditing for backdoors (Shoshitaishvili et al., 2015), and analyzing binary-only patches (Brumley et al., 2008). Current state-of-the-art decompilers such as Ghidra (NSA) Hex-Rays (Hex-Rays SA, a), and Binary Ninja (Vector 35, 2026) decompile using traditional program analysis and refinement techniques (Brumley et al., 2013).

Deterministic decompilers typically target C as their high-level output language, and their capability to decompile binaries back to C is strong due to decades of investment and in general the close mapping of C concepts to binary-level semantics (Kernighan & Ritchie, 1988). Support for other high-level languages (HLLs) such as Go, Swift, and Rust, is mixed at best. Often, the **only** output supported by the decompiler is C. For example, a Swift program is represented as a C program, implementing low-level Swift semantics. This is because re-implementing a reverse-compiler to a new HLL is a massive undertaking, because current HLLs are much more semantically distant from binary code than C (Hosseini & Dolan-Gavitt, 2022).

Recently deep learning has been used to assist with decompilation, including usage of models trained specifically for a given decompilation task and general purpose large language models (LLMs) (Fu et al., 2019; Tan et al., b). Neural decompilers such as LLM4Decompile (Tan et al., b) and Idioms (Dramko et al., 2026), which finetune LLMs for C decompilation, are trained for decompilation specifically, using data specific to C programs. Generalized LLMs are also capable. There is recent adoption of using Claude Code and Codex to rapidly assist in the decompilation of multiple video games, to C and C++ (Lewis, 2025).

We propose to expand the usage of LLMs for decompilation. Conceptually, the decompilation problem is more of a *translation* problem than a reverse-compilation problem. The original language is the source code and the new language is binary code. As with human languages, some semantic concepts and idioms from the source language will not match the binary. the neural translator can learn this given enough examples. Fortunately for this particular problem, data can be generated effectively at-will given the existing breadth of open source code and the ability of LLMs to write software applications in HLLs. This approach liberates us from complex re-targeting for each source HLL and destination instruction-set architecture (ISA) pair. The mapping can simply be learned given enough examples. Many frontier LLMs have already encountered enough examples of common language/ISA pairs, and already excel at the task.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```swift
func read() throws -> Data {
  do {
    return try
      Data(contentsOf:  url)
  } catch {
    throw ReadError(
      path: path,
      reason: .readFailed(
        error))
  }
}
```

(a) Original Swift Source.

```
...
result_1 = Foundation
  .Data.__allocating_init(
    contentsOf:options:)(
    contentsOf, 0, error);
 ...
if (error)
{
 ...
  x0_11 = _swift_allocError(
    sub_40de58(&data_418160,
      &data_411458),
    sub_410354(0x418168,
      &data_418160,
      &data_411458,
      ....
}
return result;
```

(b) Binary Ninja Decompilation (excerpt).

```swift
public func read() throws
    -> Data {
  do {
    return try
      Data(contentsOf:  url)
  } catch {
    throw FilesError(
      path: path,
      reason: ReadErrorReason
        .readFailed(error))
  }
}
```

(c) BLANKSPACE Decompilation.

Figure 1: Decompiling `File.read()` from the Swift library Files (1a). Binary Ninja (1b) emits pseudo C that exposes runtime calls and pointer writes, while BLANKSPACE (1c) recovers Swift close to the original, including `throws` and `do`/`catch`. Boxes mark the file read (red) and the thrown error (blue).

To evaluate decompliation-as-translation, we present BLANKSPACE, a framework for multi-language neural decompilation which supports a variety of instruction set architectures (ISAs) to target HLLs. Instead of providing the burden of training models for each target HLL to users, BLANKSPACE uses pretrained LLM coding agents paired with extracted metadata from the provided binary and feedback from the HLL's compiler and binary similarity scoring with respect to the original input binary. We provide support for arm64 to Swift, x86-64 to C, and x86-64 to Go decompilation. We evaluate each of the target HLL decompilation configurations on BLANKSPACEBENCH, our decompilation evaluation consisting of 50 top open source Swift projects, 20 Go projects, and 20 C projects. We build BLANKSPACE on coding agent clients Claude Code and Codex, with frontier LLM model pairings. We also compare BLANKSPACE to state of the art deterministic decompilers and the same LLM and coding agent pairings without the BLANKSPACE framework.

Our main contributions are:

- **Multi-language Agentic Decompilation Framework.** Given a stripped binary of a target HLL, BLANKSPACE decompiles the binary to source code at the repository level. BLANKSPACE is an agentic framework supporting multiple ISA to HLL decompilation tasks due to easily porting between each ISA to HLL specification while maintaining a generalized decompilation pipeline that includes translation of binary, and a feedback loop that guides translation given target HLL compilation feedback and binary similarity scoring.
- **Benchmark on Decompilation at the Repository Level of Real World Projects.** BLANKSPACE decompiles entire projects, as opposed to decompiling at the only the function level. Due to existing benchmarks only providing function level decompilation evaluation, we curated our own benchmark, BLANKSPACEBENCH, to support repository level decompilation evaluation. We fetch top open source projects and their unit tests for Swift, Go, and C, and also collect open source downstream clients that import the target projects. We measure functional correctness by executing each project's upstream tests and downstream client tests.
- **Evaluation of Decompilation Approaches.** We evaluate BLANKSPACE, frontier models paired with coding agents, and deterministic decompilers on BLANKSPACEBENCH. We also include ablations of BLANKSPACE to examine the effects of binary analysis tool access and iterative feedback. We also evaluate on the DecBench (Basque, 2026) sample set.

2 MOTIVATING EXAMPLE

As of September 2026, Swift ranks a modest #18 on the Tiobe Language Index (TIOBE Software, 2026). For Apple platforms such as MacOS and iOS, Swift is the default development language. Malware for Apple platforms is also written in Swift (SentinelOne, 2025). Decompilation is used to rapidly assess application behavior statically, without running the application, and aids skilled reverse engineers in in-depth analysis. The current generation of decompilers translate machine code to C, or a C-like language. Swift is far from a C-like language.

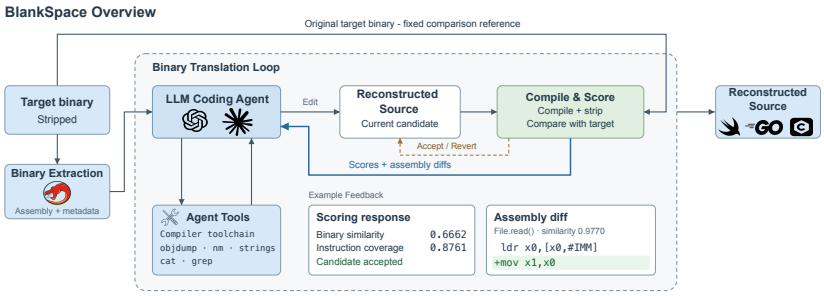

Figure 2: Overview of BLANKSPACE. The agent reconstructs source from an extracted binary, using compilation, similarity scores, and assembly diffs to guide revisions.

We show this behavior in a motivating example in Figure 1: `File.read()` from the Swift library Files (Sundell, 2017), a short method that reads a file into a `Data` value and throws a library error if the read fails. After stripping the binary, we provide it to Binary Ninja. Figure 1b is the C pseudo code output by Binary Ninja: the read becomes a call to a Foundation initializer, and the thrown error becomes a Swift runtime allocation. Figure 1c is the Swift source code recovered by BLANKSPACE, which expresses the same read and error handling with `throws` and `do`/`catch`, as in the original (Figure 1a). BinaryNinja is a state of the art disassembler/decompiler, and is representative of the apex of current decompilation tools' support for Swift. Decompilers that broadly implement a reverse-compilation pipeline cannot easily re-target to Swift or other HLLs.

3 BLANKSPACE

Given a stripped binary, BLANKSPACE uses language-specific analysis tools to extract assembly and retained metadata. An LLM coding agent uses the extracted facts to reconstruct source code through iterative compilation and binary similarity feedback. BLANKSPACE evaluates the coding agent's revisions, automates selecting program candidates, and provides function-level differences from revisions. The pipeline outputs reconstructed source in the target HLL. We describe the pipeline for Swift binaries below, followed by its adaptation to C and Go (3.7) and the design challenges encountered during development (B.2).

3.1 PROBLEM STATEMENT

We formalize the *decompilation problem*, the task of reconstructing source code from a given binary. Let B be a stripped target binary, L its source language, and $\mathcal{P}_L$ the set of source programs in L. Let θ specify the build environment used for reconstruction, including the compiler, target architecture, optimization settings, and dependencies. For a candidate program P that compiles in θ, let B_P be the resulting binary.

Let $\mathcal{X}_B$ denote the set of valid call sequences using the library interface of B, each with specified arguments and an external environment. For a binary b and $x \in \mathcal{X}_B$, let $\mathsf{Obs}(b, x)$ denote the observable behavior of b under x, including return values, exceptions, and termination. For nondeterministic executions, $\mathsf{Obs}(b, x)$ denotes the set of possible observable traces. **Binary to Source Code Reconstruction Task.** Given a stripped binary B, we seek to reconstruct a source program $\widehat{P}$ in language L satisfying:

1. $\widehat{P}$ **compiles.** $\widehat{P}$ compiles successfully in θ.
2. **Interface compatibility.** $\widehat{P}$ implements the same interface as the interface of $\mathcal{X}_B$.
3. **Behavioral equivalence.** The runtime behavior of the reconstructed binary of $\widehat{P}$ is equivalent to that of the runtime behavior of target B:

$$\forall x \in \mathcal{X}_B, \quad \mathsf{Obs}(B_{\widehat{P}}, x) = \mathsf{Obs}(B, x).$$

$\widehat{P}$ *compiles* when the first condition holds and is *behaviorally correct* when all three conditions hold. Multiple source programs may satisfy these requirements given the size of B.

3.2 BINARY ARTIFACT EXTRACTION

Given the stripped Swift binary B, we use Ghidra to disassemble and analyze the binary. Using custom Ghidra scripts, from the assembly we extract functions, function addresses and sizes,

and annotate instructions using recovered symbols and references. Mangling is when a compiler encodes declaration names and type information into symbols (Levine, 2000). Demangling decodes the symbols into human-readable names and type signatures (The Swift Project, a). We demangle retained Swift symbols using the command `swift demangle --compact` (The Swift Project, b) and filter compiler and runtime-generated assembly. Then, we format the extraction as a filesystem where each text file contains each function and its address, size, demanged signature, and assembly. We provide an index file for LLM agent navigation. BLANKSPACE also uses `swift-reflection-dump` to extract retained Swift metadata which includes type and field records. The filesystem is a resource of statically extracted information for the LLM agent to refer to as it translates the binary.

3.3 BINARY TRANSLATION LOOP

We provide the LLM coding agent its own workspace with the original stripped binary, the extracted references from 3.2 as a filesystem, and access to the following binary analysis tools: `otool`, `objdump`, `nm`, `swift demangle`, `strings`, `dyld_info`, and a compilation command consistent with the reconstruction environment θ. The agent edits a source directory to reconstruct the original Swift project. As the agent translates, it uses these tools to inspect the target, compile source code, and examine the generated assembly. BLANKSPACE provides a custom feedback tool that the agent can use to score its current reconstruction's binary similarity to the original binary (3.4). The custom feedback tool also compiles in environment θ and stores a snapshot of the current state of the agent's translation, and automatically determines whether to accept the current program state or revert to a better scoring program state (3.5). The custom feedback tool also generates a filesystem of function-level assembly differences between snapshots of program states (3.6). The loop of translation using tooling and feedback continues within a fixed timeout budget.

3.4 BINARY SIMILARITY SCORING

We compare the assembly of the target binary B with that of the compiled candidate translation B_P. We first disassemble both binaries into functions and normalize their instructions using architecture-specific rules. For ARM64, we also map selected registers to a smaller set of register names. These transformations reduce the score's sensitivity to differences in code layout and compiler choices. The custom scoring tool reports (Listing 2) the following components together to the agent, so that the agent can decide whether to reconstruct additional functions, improve existing functions, or reduce unmatched code.

Function similarity. For a target function f_i and candidate function g_j, let t_i and c_j denote their normalized instruction counts. We align their instruction sequences using `difflib.SequenceMatcher` from Python's standard library (Python Software Foundation). Instructions in identical blocks receive full credit. Within regions where the instruction sequences differ, we greedily pair instructions with the same mnemonic and assign partial credit based on agreement between their destination and remaining operands. Unpaired instructions receive no credit. Let m_{ij} denote the total instruction credit. The function similarity is

$$s_{ij} = \frac{2m_{ij}}{t_i + c_j}.$$

We first pair functions with identical retained symbol names. A shared name establishes correspondence but contributes no similarity credit. We match the remaining functions one-to-one, considering pairs with similarity above above a minimum threshold and normalized instruction sequences differing in length by at most a factor of two.

Aggregate similarity. Let s_i denote the similarity assigned to target function f_i, with $s_i = 0$ for unmatched target functions. Let $T = \sum_i t_i$ be the total target instruction count and E the total instruction count of unmatched candidate functions. We compute

$$S(B, B_P) = \frac{\sum_i t_i s_i}{T + E}.$$

This weighs each function by its size in the target. Missing target functions contribute zero credit, while unmatched candidate functions increase the denominator.

Coverage feedback. A similarity score averaged only over matched functions can favor accurate reconstruction of a small subset of the target while failing to translate the remaining functions. We

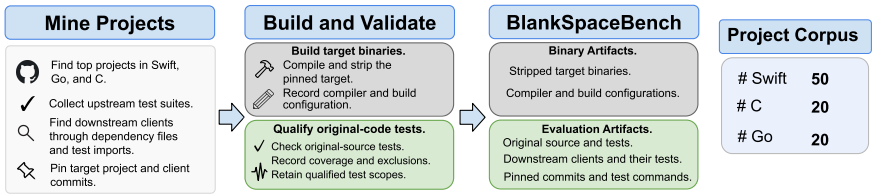

Figure 3: BLANKSPACE construction. Projects are mined with their upstream tests and downstream clients. The corpus has 50 Swift, 20 C, and 20 Go top open source projects.

include coverage to encourage a broader scope of reconstruction. We also report the fraction of target instructions belonging to functions that receive nonzero similarity:

$$C(B, B_P) = \frac{\sum_{i:s_i>0} t_i}{T}.$$

This distinguishes how much of the target has received a partial reconstruction from how closely those reconstructed functions match. The feedback additionally reports the instruction weighted mean similarity among these functions, the excess-code ratio E/T, and changes in similarity and coverage relative to the previously accepted candidate. The same matching and aggregation rules apply to Swift, C, and Go. Function recovery and architecture-specific normalization for C and Go are discussed in 3.7.

3.5 TRANSLATION CANDIDATE SELECTION

Every time the LLM agent calls the BLANKSPACE custom scoring tool, a snapshot of the current state of the agent's source reconstruction is saved. BLANKSPACE keeps track of the last accepted snapshot, P_{acc}. At each feedback request, BLANKSPACE evaluates the submitted candidate P. If compilation and build validation succeed, the candidate is accepted when $S(B, B_P) \geq S(B, B_{P_{\text{acc}}})$, where S is the aggregate similarity score defined in 3.4. The accepted program snapshot P_{acc} is then updated to P. Equal scoring candidates are accepted, allowing revisions that preserve the current score. If the score decreases or compilation fails, BLANKSPACE automatically restores the agent's editable source directory to P_{acc}. The feedback response reports this decision, and subsequent decompilation continues from the restored source. At completion, BLANKSPACE returns the source from the last accepted program snapshot. The output preserves the highest aggregate score attained among successfully evaluated candidates, with ties resolved in favor of the most recently accepted candidate.

3.6 ASSEMBLY COMPARISON FEEDBACK

After a translation candidate is scored, BLANKSPACE generates function-level assembly diffs between the target and candidate binaries using the function pairings from 3.4. Along each function diff, we include similarity scores between the target and candidate functions and the change in similarity score with respect to the previously accepted candidate snapshot. The diff files are saved in a read-only directory for each snapshot. An index lists the functions and the line ranges of their diffs. The scoring tool returns the index path in its JSON response, enabling retrieval of individual functions. The diffs remain available when a candidate is reverted, so the agent can still inspect the rejected candidate's assembly.

3.7 PORTING BLANKSPACE TO OTHER LANGUAGES

Supporting Go and C required changes in binary extraction, build configuration, and ISA-specific instruction comparisons. The binary translation loop (3.3) remained the same across languages. For C we extract disassembly with Ghidra and identify exported function s for candidate program scoring using ELF dynamic symbols. For Go, we use GoReSym (Mandiant, 2022) to recover function names and boundaries from runtime metadata, and extract assembly with `objdump`. For each ISA and language pairing, we compile and strip programs using the same configuration used to compile the target binary. Both of the C and Go ports target x86-64 and share the same instruction normalization and operand-handling rules, while using the same Swift function matching and scoring formulas.

4 BLANKSPACEBENCH: REPOSITORY-LEVEL HLL DECOMPILATION

Existing neural decompilation benchmarks such as HumanEval-Decompile (Tan et al., b) and ExeBench (Armengol-Estapé et al., 2022) are from 2024 or earlier, test decompilation at the func-

tion level, and only provide assembly to C decompilation evaluation. Thus to mitigate testing on old datasets possibly in the knowledge cutoff date and to test decompilation in real world scenarios, we source from top open source projects. Functional correctness tests executable behavior of the decompiled source code. For each project, we source functional correctness tests from the project's own test suites (*upstream tests*) and from dependent projects (*downstream tests*). We chose to mine downstream client tests instead of only using a target project's own tests to ensure the decompilation outputs are evaluated in realistic settings. Although we only support Swift, Golang, and C, the mining and validation techniques for curating the benchmark can be generalized to other languages.

4.1 Mining for Projects and Functional Correctness Tests

We search on GitHub for top open source projects determined by number of GitHub stars, in Swift, Golang, and C. Upstream tests are from each project's existing test suite. We record the repository commit of the fetched source code and tests, along with the build commands used for evaluation. Downstream clients are found by searching for references to the target project in other repositories. For Swift, we search for the target repository's URL in `Package.swift` and `Package.resolved`, and for import statements of the target library's modules. For Go, we search for the target's module path in `go.mod`. For C, we search for dependency references in submodule declarations and build files. For evaluating BlankSpace, we chose to have pairs of target projects and single downstream client test suites, though for multiple projects we were able to find more than one dependent client test suite.

4.2 Validating BlankSpaceBench Functional Correctness Tests

A downstream client may depend on a target project without its tests executing the target project's code. Thus, we validate each target project and downstream client pair by linking the original target's source code and executing the client's tests. We exclude tests that fail with the original source code. We run the selected client's tests with the pinned version of the original target, and require successful execution without test failures. We enable code coverage instrumentation to record which parts of the target project were executed during the tests. For instrumenting Swift we used `swift test --enable-code-coverage`, for Go `go test -coverpkg` with `-coverprofile`, and `llvm-cov` for C.

4.3 Functional Correctness Execution Modes

We test functional correctness of decompiled code in two modes. Whole module execution measures whether the reconstruction works as a replacement for the target module. However, an incompatible type or missing declaration can prevent the module from compiling. This can leave out measuring the functional correctness of individual functions. We also use function-level execution to test decompiled functions that we substitute into the original source code. We use both modes since only testing individual functions doesn't test the complete decompiled project's behavior.

5 Evaluation

Our evaluation answers the following research questions:

- **RQ 1** What is the decompilation performance of BlankSpace?
- **RQ 2** How effective is each component of BlankSpace?
- **RQ 3** How does BlankSpace affect decompilation quality across coding agents and models?
- **RQ 4** How generalizable is BlankSpace to supporting multiple target HLLs?

5.1 Experiment Setup

We evaluate the BlankSpace framework with coding agents Claude Code (Anthropic) with high effort and Opus 5 and Codex (OpenAI) with high effort and models GPT5.6 Sol and GLM 5.3. For baselines we compare BlankSpace against each of the coding agent and model configurations, providing each agent access to the same set of binary analysis tools (3.3), the target HLL toolchain, and include the same prompts to decompile the given binary. Experiments were run on a MacBook Pro with a 18-core Apple M5 Max and 64 GB of memory for Swift decompilation, and 2x Digital Ocean general purpose droplets with 32 virtual CPUs and 128 GB of RAM for Go and C decompilation. Each experiment run is given a timeout of 30 minutes. For Swift we used Apple Swift 6.3.3 compiler with the macOS 26.5 SDK, targeting ARM64 with `-O`. For C we used Clang 21.1.8 targeting Linux x86-64 with `-O2` and `-O3`. For Go we used Go 1.26.4 targeting Linux amd64 with default optimization and inlining. Swift, C, and Go inputs are stripped before decompilation. For DecBench we used GCC 11.4.0 with `-O0`, `-O2`, and `-O2 -fno-inline`.

```
int max(int a, int b, int c, int *p)
{
    if (a > b) { *p = 0; return a; }
    else       { *p = c; return b; }
}

// I/O pair (p truncated to 4 of 32)
in:  a=125, b=79, c=97,
     p=[57, 24, 41, 22, ...]
out: ret=125,
     p=[0, 24, 41, 22, ...]
```

(a) ExeBench function and I/O pair (test-real, #316).

```
// cJSON/tests/cjson_add.c
static void cjson_add_null_should_add_null(void) {
    cJSON *root = cJSON_CreateObject();
    cJSON *null = NULL;
    cJSON_AddNullToObject(root, "null");
    TEST_ASSERT_NOT_NULL(null =
        cJSON_GetObjectItemCaseSensitive(root, "null"));
    TEST_ASSERT_EQUAL_INT(null->type, cJSON_NULL);
    cJSON_Delete(root);
}
```

(b) BLANKSPACEBENCH upstream test (`cJSON`).

```
// pg-status/test/tests/test_pg_status_api.c
static void test_missing_master_json(void) {
    const TestHost hosts[] = { fixture_pg_status_dead_host("dead") };
    PgStatusApiFixture api =
        fixture_pg_status_start(hosts, sizeof(hosts) / sizeof(hosts[0]), -1);
    TestHTTPResponse response =
        http_test_get(api.port, "/master", "Accept: application/json\r\n");
    fixture_pg_status_expect_json(&response, 404,
        "{\"host\":null,\"dc\":null,\"geo\":null}");
    http_test_response_free(&response);
    fixture_pg_status_stop(&api);
}
```

(c) BLANKSPACEBENCH downstream test (`pg-status`, a client that depends on the target library `cJSON`).

Figure 4: Functional correctness comparisons. ExeBench (a) checks a single function against input/output pairs. BLANKSPACEBENCH runs the project's own tests (b) and tests from dependent packages (c) against the decompiled project.

Dataset. We used BLANKSPACEBENCH, consisting of top open source projects in Swift, C, and Go. For the C projects, we decompiled 26 of DecBench's open source C projects, and evaluated on its x86-64 sample set targets for optimization levels `O0`, `O2`, `O2-noline`. This subset of the sample set consists of 158 sampled functions, of the full 250 sample set.

5.2 EVALUATION METRICS

Instruction Similarity. We compile the original and decompiled code under the same compiler and build settings. We pair functions using unique linker symbols and compare their normalized instruction sequences. We align instructions using `difflib.SequenceMatcher`. Let M be the number of aligned identical instructions, and let U_r and U_c be the numbers of unaligned instructions in the compiled ground truth source code and compiled decompilation output. We report

$$I = 100 \cdot \frac{M}{M + U_r + U_c}.$$

We sum the quantities across compared functions before computing the score. We use unique compiler linkage symbols to establish function correspondence. This avoids selecting function pairs because their assembly looks similar, but excludes functions without an unambiguous symbol match, including potentially correct reconstructions with renamed symbols.

BinDiff Similarity. BinDiff (Dullien & Rolles, 2005) is a binary comparison tool. We use BinDiff's whole binary similarity score to compare the target and recompiled binaries. BinDiff uses heuristics such as call graph structure to pair functions and basic blocks, then combines their matches into a whole binary similarity score. BinDiff can compare functions without matching symbols, but may not catch operand differences that affect behavior.

Normalized Edit Distance to Source. To measure resemblance to original source, we use the Levenshtein edit distance between the decompiled code and the original source code, divided by the length of the longer sequence. Comments and whitespace are excluded. Lower values indicate sloer textual similarity. However, this metric penalizes naming and implementation changes even if behavior is preserved.

Function-Level Functional Success. For each project we report the percentage of functions' corresponding decompiled functions compile, execute, and pass the selected tests when substituted individually into the original module.

Whole-Module Functional Success. Whole-module functional success is achieved when a reconstructed module builds and all of its tests pass. For upstream tests, we rebuild the target package with the reconstructed module and for downstream tests, we rebuild the downstream client. We report upstream and downstream success separately.

Table 1: Swift Decompilation Evaluation (50 Swift Projects).

	Static Metrics $N = 50$			Whole-module Success (%) ↑		Function-level Success (%) ↑
Method	**Instruction Sim. (%) ↑**	**BinDiff Sim. (%) ↑**	**Source Edit Distance ↓**	**Upstream** $N = 38$	**Downstream** $N = 33$	**Upstream Project-average** $N = 22$
BLANKSPACE	40.19	47.99	0.803	5.26	24.24	28.09
Ghidra	26.34	75.87	0.9752	–	–	–
Binary Ninja	25.21	84.05	0.9655	–	–	–
Codex, GPT-5.6 Sol	36.47	43.08	0.798	7.89	36.36	29.28

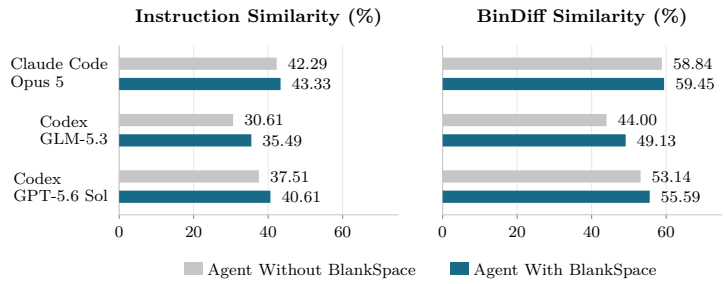

Figure 5: Cross-agent comparison on 20 Swift projects. Gray bars show each agent/model without BLANKSPACE, and blue bars show the same agent/model with BLANKSPACE.

5.3 RQ1: BLANKSPACE PERFORMANCE

Table 1 compares decompilation performance on 50 Swift projects. With the same GPT5.6 Sol model, BLANKSPACE achieves higher instruction similarity than plain Codex (40.19% versus 36.47%) and higher BinDiff similarity (47.99% versus 43.08%), gains of approximately +3.7% and +4.9%. Source edit distance slightly favors Codex (0.798). Both LLM methods have higher instruction similarity and lower source edit distance than Ghidra and Binary Ninja while the deterministic decompilers have higher BinDiff similarity. Functional evaluation reveals a different ranking. On the 38 projects with evaluated upstream tests, whole-module success is 2/38 for BLANKSPACE and 3/38 for Codex. On the 33 projects with measured downstream tests execution for both methods, success is 8/33 for BLANKSPACE and 12/33 for Codex. Across the 22 upstream projects' function level execution, the project average pass rate is 28.09% for BLANKSPACE and 29.28% for Codex. Thus, the observed binary similarity increases do not translate into higher functional correctness success on these evaluated subsets.

5.4 RQ2: ABLATION EXPERIMENTS

We evaluate four BLANKSPACE ablation configurations on the same 20 Swift projects using Codex with GPT-5.6 Sol, high reasoning effort, and a 30-minute budget per run. We measure instruction similarity, BinDiff similarity, and normalized source edit distance, averaging each metric equally across projects. Each ablation configuration removes a component while keeping all other components of BLANKSPACE. Removing *assembly diffs* removes function-level assembly diffs. Without *binary extraction*, the agent has no access to Ghidra extracted disassembly. No *binary analysis tools* removes the compiler and binary analysis tools from the agent's allowed tool usage. Removing the *scoring loop* disables the scoring feedback, assembly diffs, and automatic reverting of translation candidates that drop scoring. Table 2 shows that the configuration without assembly diffs has the highest mean instruction and BinDiff similarity among the four ablations (42.92% and 56.60%). The configuration without the scoring loop has the lowest binary similarity means (37.53% and 51.91%), and the lowest source edit distance (0.7525).

5.5 RQ3: CROSS AGENT AND MODEL COMPARISON

Figure 5 compares three agent–model configurations with and without BLANKSPACE on 20 Swift projects. Adding BLANKSPACE increases both instruction similarity and BinDiff similarity in all three configurations. Codex with GLM-5.3 shows the largest gains: +4.9% in instruction similarity and +5.1% in BinDiff. Codex with GPT-5.6 Sol improves by +3.1% and +2.4%, while Claude Code

with Opus 5 improves by +1% and +0.6%. Normalized source edit distance (see Table 6) decreases for GLM-5.3, from 0.8167 to 0.8156, but increases for Opus 5 and GPT-5.6 Sol. Thus, the binary similarity improvements extend across the tested agents and models, while closer recovery of the original source text is not consistent across configurations.

5.6 RQ4: Multi-language Support

On the 20 Go projects (see Table 3), BlankSpace obtains an average instruction similarity of 14.01%, compared with 10.73% for Codex. Table 4 reports results on DecBench's x86-64 sample subset. BlankSpace achieves a union score of 24.1% (38/158), compared with 21.5% (34/158) for plain Codex. The largest gain is in recompile: BlankSpace produces 14 perfect matches (8.9%), compared to seven (4.4%) for plain Codex. Structure improves from 31 to 32 perfect matches, and Types from six to seven.

6 Discussion

An advantage of BlankSpace is its reusable decompilation workflow. By delegating source recovery to a coding agent, the framework combines existing binary analysis tools with compilation and similarity feedback across languages, reducing the need to implement a separate language-specific decompiler. Our Swift, C, and Go evaluations demonstrate this reuse across different language and architecture settings. Extending support still requires appropriate binary extraction, build configuration, and extending evaluations. Furthermore, improved binary similarity does not consistently translate into functional success, motivating further work on decompiling entire projects that compile and pass their test suites.

7 Related Work

HLL Decompilation. Deterministic decompilers that specialize in binary to C translation such as Hex-Rays and Binary Ninja offer extensions to translate to other HLLs. Hex-Rays offers a Swift plugin (Hex-Rays SA, b) which demangles Swift names. Binary Ninja features a Pseudo-Rust view (Wiens, 2024), and GhidRust (DMaroo, 2023) is an open source plugin that translates Ghidra's C output to Rust pseudo-code. Oxidizer (Liu et al., 2026) is a deterministic binary to Rust decompiler, which uses angr (Shoshitaishvili et al., 2016) for control-flow graph (CFG) and functional-level CFG construction, then uses its own Rust-specific type inference.

Neural Decompilation. LLM4Decompile (Tan et al., b) was the first open-weight LLM series for x86 to C decompilation, which used ExeBench (Armengol-Estapé et al., 2022) as the training corpus and trained on two types of tasks including disassembled binary to C translation and Ghidra pseudo-code to C translation. Idioms (Dramko et al., 2026) is a neural decompiler finetuned to predict user-defined type definitions alongside decompiled code, taking Hex-Rays output as initial input. Decaf (Shypula et al., 2026) samples candidate decompilations, discards the candidates that fail to compile, then uses a trained reranking model to rank each candidate's recompiled assembly. SK2Decompile (Tan et al., a) decompiles using two models by first recovering program structure as an IR with placeholder identifiers, then predicting the names of the IR. Beyond the C (Hosseini & Dolan-Gavitt, 2022) trains transformer models to decompile multiple languages by treating assembly and source languages as plain text translation.

8 Conclusion and Limitations

BlankSpace is a multi-language neural decompilation framework for decompiling stripped binaries at the repository level, and BlankSpaceBench, a decompilation benchmark spanning 90 Swift, C, and Go projects. Our results show improvements in binary similarity while functional correctness remains challenging. We will open source BlankSpace and BlankSpaceBench. In future work we plan to explore using BlankSpace for assisting with model training, improving functional correctness, and extending BlankSpace support for other HLLs.

Limitations. To support other HLL translations, BlankSpace relies on existing decompilation-related tooling for initial assembly and metadata to guide translation. The knowledge cutoff date for GPT5.6 Sol is Febuary 16, 2026 (OpenAI, 2026). The knowledge cutoff date for Opus 5 is May 2026 (Anthropic, 2026). Of our 90 pinned target project commits, 49 projects precede or are within May 2026 and 41 projects are after May 2026.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Anthropic. Claude code. https://github.com/anthropics/claude-code. Accessed: 2026-09-26.

Anthropic. System card: Claude opus 5. Technical report, Anthropic, July 2026. URL https://www-cdn.anthropic.com/b514064af1408018e64b1ad24e7d5e75850b4ffd/Claude%20Opus%205%20System%20Card.pdf. Accessed: 2026-09-26.

Apple Inc. Swift HTTP Types. https://github.com/apple/swift-http-types, 2023.

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 50–59, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534867. URL https://doi.org/10.1145/3520312.3534867.

Asana. locheck: Validate iOS, Android, and Mac localizations. https://github.com/Asana/locheck. Accessed: 2026-09-26.

Zion Leonahenahe Basque. DecBench: A decompiler benchmark. https://decbench.com/about/, 2026.

David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. 2008.

David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 353–368, 2013.

Cristina Cifuentes and K. John Gough. Decompilation of binary programs. 25 (7):811–829. ISSN 1097-024X. doi: 10.1002/spe.4380250706. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250706. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250706.

DMaroo. GhidRust: Rust binary analysis extension for Ghidra. https://github.com/DMaroo/GhidRust, 2023. Accessed: 2026-09-26.

Luke Dramko, Claire Le Goues, and Edward J Schwartz. Idioms: A simple and effective framework for turbo-charging local neural decompilation with well-defined types. In *NDSS*, 2026.

Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

Asdine El Hrychy. Storm: Simple and powerful toolkit for BoltDB. https://github.com/asdine/storm. Accessed: 2026-09-26.

etcd Authors. bbolt: An embedded key/value database for Go. https://github.com/etcd-io/bbolt. Accessed: 2026-09-26.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/093b60fd0557804c8ba0cbf1453da22f-Abstract.html.

Jean-loup Gailly and Mark Adler. zlib: A massively spiffy yet delicately unobtrusive compression library. https://github.com/madler/zlib. Accessed: 2026-09-26.

Hex-Rays SA. Hex-Rays decompiler. https://hex-rays.com/decompiler/, a.

Hex-Rays SA. Swift plugin. IDA Documentation, https://docs.hex-rays.com/add-ons/plugins/concepts/plugins-shipped-with-ida/swift-plugin, b. Accessed: 2026-09-26.

Iman Hosseini and Brendan Dolan-Gavitt. Beyond the C: Retargetable decompilation using neural machine translation. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2022. URL https://www.ndss-symposium.org/wp-content/uploads/bar2022_23009_paper.pdf.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

Latchset. José: C-language implementation of JavaScript object signing and encryption. https://github.com/latchset/jose. Accessed: 2026-09-26.

John R Levine. *Linkers and loaders*. Morgan Kaufmann, 2000.

Chris Lewis. The unexpected effectiveness of one-shot decompilation with Claude. Chris' Blog, November 2025. URL https://blog.chrislewis.au/the-unexpected-effectiveness-of-one-shot-decompilation-with-claude/. Accessed: 2026-09-26.

Yibo Liu, Zion Leonahenahe Basque, Arvind S Raj, Chavin Udomwongsa, Chang Zhu, Jie Hu, Changyu Zhao, Fangzhou Dong, Adam Doupé, Tiffany Bao, et al. Oxidizer: Toward concise and high-fidelity rust decompilation. In *2026 IEEE Symposium on Security and Privacy (SP)*, pp. 3433–3451. IEEE, 2026.

Mandiant. GoReSym: Go symbol recovery tool. https://github.com/mandiant/GoReSym, 2022.

NSA. Ghidra. URL https://github.com/NationalSecurityAgency/ghidra. original-date: 2019-03-01T03:27:48Z.

OpenAI. Codex. https://github.com/openai/codex. Accessed: 2026-09-26.

OpenAI. GPT-5.6 Sol model. OpenAI API Documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol. Model identifier: gpt-5.6-sol. Accessed: 2026-09-26.

Python Software Foundation. difflib — helpers for computing deltas. Python 3 Documentation, https://docs.python.org/3/library/difflib.html. Accessed: 2026-09-26.

SentinelOne. 2024 macOS malware review: Infostealers, backdoors, and APT campaigns targeting the enterprise. SentinelOne Blog, 2025. URL https://www.sentinelone.com/blog/2024-macos-malware-review-infostealers-backdoors-and-apt-campaigns-targeting-the-e

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pp. 138–157. IEEE, 2016.

Alexander Shypula, Osbert Bastani, and Edward Schwartz. Decaf: Improving neural decompilation with automatic feedback and search. *arXiv preprint arXiv:2605.11501*, 2026.

John Sundell. Files: A nicer way to handle files & folders in Swift. https://github.com/JohnSundell/Files, 2017.

Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. SK2decompile: LLM-based two-phase binary decompilation from skeleton to skin, a. URL http://arxiv.org/abs/2509.22114.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487, b. doi: 10.18653/v1/2024.emnlp-main.203. URL http://arxiv.org/abs/2403.05286.

The Swift Project. Mangling. Swift ABI documentation, https://github.com/swiftlang/swift/blob/main/docs/ABI/Mangling.rst, a. Accessed: 2026-09-26.

The Swift Project. The Swift programming language toolchain. https://github.com/swiftlang/swift, b.

TIOBE Software. TIOBE programming community index. https://www.tiobe.com/tiobe-index/, 2026. September 2026 edition. Accessed: 2026-09-26.

Vector 35. Binary ninja. https://binary.ninja/, 2026. Accessed: 2026-09-26.

Jordan Wiens. Binary ninja 4.2 frogstar. Binary Ninja Blog, Vector 35, November 2024. URL https://binary.ninja/2024/11/20/4.2-frogstar.html. Accessed: 2026-09-26.

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177. IEEE, 2016.

A EVALUATION DETAILS

We report DecBench's function-level perfect match rates (Basque, 2026). *Structure* counts functions whose reconstructed and reference control-flow graphs have a graph edit distance of zero. *Types* counts functions with a type match score of one. *Recompile* counts functions whose recompiled, normalized assembly matches the reference. *Union* counts functions perfect on at least one metric.

Table 2: BLANKSPACE Ablations (20 Swift projects).

Configuration	Instruction Similarity (%) ↑	BinDiff Similarity (%) ↑	Source Edit Distance ↓
w/o Assembly Diffs	42.92	56.60	0.7627
w/o Binary Extraction	40.56	55.14	0.7751
w/o Binary Analysis Tools	42.13	54.96	0.7980
w/o Scoring	37.53	51.91	0.7525

Table 3: Multi-language Evaluation.

(a) BlankSpace and Codex

		Binary similarity		Whole-module Success		Function-level Success	
Setting	Method	Instruction	BinDiff	Upstream	Downstream	Upstream	Downstream
Go	BLANKSPACE	14.01	–	0.00	5.00	–	–
	Codex	10.73	–	0.00	5.00	–	–
C `-O2`	BLANKSPACE	36.00	56.82	30.00	45.00	24.12	21.70
	Codex	25.60	48.46	30.00	45.00	24.01	21.56
C `-O3`	BLANKSPACE	35.34	56.70	25.00	45.00	31.78	24.32
	Codex	27.25	48.26	40.00	45.00	29.58	24.94

Table 4: DecBench x86-64 functions, sample set.

Approach	Union	Structure	Types	Recompile
BLANKSPACE	**24.1%**	**20.6%**	**4.6%**	**8.9%**
Codex, GPT-5.6 Sol	21.5%	20.0%	4.0%	4.4%

Table 5: Per-project Swift decompilation results for BLANKSPACE and Codex, both using GPT5.6 Sol. Dashes indicate unavailable function measurements.

	Instruction Sim. (%) ↑		BinDiff Sim. (%) ↑		Source Edit Distance ↓		Function Success (%) ↑	
Project	BLANKSPACE	Codex	BLANKSPACE	Codex	BLANKSPACE	Codex	BLANKSPACE	Codex
AEXML	61.84	64.09	76.69	83.40	0.5170	0.3278	92.50	95.00
AsyncAlgorithms	50.11	45.35	37.85	33.93	0.8538	0.8532	9.96	8.94
AudioKit	13.49	15.00	46.68	45.88	0.8486	0.8568	–	–
CombineSchedulers	46.19	44.15	38.49	38.36	0.7197	0.6394	7.84	5.88
ComposableArchitecture	24.83	22.24	21.41	21.44	0.8508	0.8437	–	–
ContainerizationEXT4	70.61	58.06	2.08	1.51	0.8166	0.8706	–	–
Dependencies	35.78	36.01	47.23	27.64	0.8411	0.8016	–	–
Files	55.47	58.00	71.19	79.33	0.6879	0.7462	55.29	41.18
GRDB	16.58	17.75	22.67	18.40	0.8932	0.9041	–	–
HTTPTypes	31.38	27.63	56.90	54.56	0.7507	0.7697	50.00	50.00
Hummingbird	34.44	24.38	5.77	5.11	0.8223	0.8136	–	–
Kanna	53.50	40.05	56.27	57.41	0.8203	0.8343	–	–
KeyboardShortcuts	41.29	36.37	61.30	57.37	0.8102	0.7667	–	–
Logging	40.23	48.82	74.75	77.55	0.7834	0.7224	32.34	40.72
Markdown	34.15	20.68	37.80	29.06	0.8336	0.8261	–	–
MarkdownUI	54.33	46.67	48.72	28.10	0.8221	0.8216	34.59	27.67
Metrics	83.49	47.51	73.14	72.77	0.9301	0.9353	22.22	33.33
Moya	28.61	29.73	52.11	22.13	0.8642	0.8440	–	–
NIOCore	19.72	17.18	25.17	21.70	0.8943	0.9004	–	–
NIOHTTP2	52.63	34.36	5.77	5.10	0.8965	0.9288	–	–
NIOSSL	21.57	16.96	6.43	5.84	0.8717	0.8833	9.22	6.31
OrderedCollections	14.86	15.86	50.46	44.47	0.8097	0.8098	–	–
PerceptionCore	46.36	62.10	67.12	49.01	0.8846	0.7740	–	–
PromiseKit	35.44	36.05	64.30	65.48	0.7933	0.7454	–	–
Quick	14.71	17.33	51.28	45.13	0.7769	0.8289	13.70	7.31
Rainbow	52.06	34.99	71.73	54.14	0.7987	0.6421	47.76	60.70
Reachability	68.39	47.99	76.98	74.00	0.7254	0.6721	20.00	40.00
ReactiveSwift	35.88	27.65	51.26	50.63	0.7972	0.8203	–	–
RealModule	59.30	58.67	68.91	70.03	0.5395	0.4581	13.73	37.25
RxSwift	20.81	20.76	48.60	40.53	0.7889	0.8219	–	–
ServiceContextModule	92.05	88.97	68.76	79.19	0.4605	0.4548	40.00	40.00
ServiceLifecycle	58.34	53.14	11.26	10.79	0.7846	0.7646	38.98	28.81
ShellOut	88.04	64.68	77.33	73.33	0.6655	0.6965	52.38	57.14
SnapshotTesting	35.73	35.93	50.62	37.76	0.8611	0.8786	–	–
SourceKittenFramework	15.05	14.39	37.76	20.97	0.8433	0.8917	–	–
Splash	63.22	59.37	67.61	62.13	0.8370	0.8433	16.00	12.00
SQLite	13.77	15.77	55.10	49.45	0.7861	0.7585	–	–
Stencil	37.34	41.32	53.47	48.76	0.8472	0.8333	–	–
SwiftCLI	40.69	34.38	60.42	61.45	0.7075	0.7598	–	–
SwiftCSV	60.13	58.38	71.81	61.18	0.7439	0.7903	10.71	10.71
SwiftDate	16.31	13.60	53.84	52.78	0.8730	0.8713	–	–
SwiftProtobuf	21.98	18.63	37.10	36.17	0.8803	0.8803	–	–
SwiftSoup	4.30	3.26	28.71	19.68	0.8812	0.9031	–	–
SwiftSyntax	27.46	25.73	27.36	8.02	0.8796	0.9623	–	–
SwiftTerm	26.34	34.17	25.37	20.22	0.9523	0.9681	–	–
SystemPackage	44.44	32.14	58.59	48.37	0.8771	0.8437	7.18	5.95
Valet	46.81	41.85	62.18	60.15	0.7525	0.7857	–	–
XMLCoder	44.33	64.14	49.37	42.38	0.8140	0.8253	6.05	6.05
Yams	26.45	28.59	41.78	43.62	0.8362	0.8752	29.26	25.00
ZIPFoundation	28.54	22.47	41.74	37.78	0.8418	0.8316	8.22	4.11
Project average	40.19	36.47	47.99	43.08	0.8033	0.7976	28.09	29.28
Projects measured	50	50	50	50	50	50	22	22

Table 6: Cross-agent/model results on 20 Swift projects.

		Instruction Similarity (%) ↑		BinDiff Similarity (%) ↑		Source Edit Distance ↓	
Agent	Model	-	BLANKSPACE	-	BLANKSPACE	-	BLANKSPACE
Claude Code	Opus 5	42.29	43.33	58.84	59.45	0.6008	0.6507
Codex	GLM-5.3	30.61	35.49	44.00	49.13	0.8167	0.8156
Codex	GPT-5.6 Sol	37.51	40.61	53.14	55.59	0.7521	0.7618

B BLANKSPACE IMPLEMENTATION

B.1 DESIGN CHALLENGES

B.2 BLANKSPACE DESIGN CHALLENGES

While iterating through versions of BLANKSPACE, the following observations greatly influenced our final design. We include these examples as potential considerations when developing neural decompilation for HLLs.

Finetuning Models to Decompile Swift. We finetuned `gpt-oss:20b` (Agarwal et al., 2025) to translate XML disassembly chunks exported from Ghidra, to Swift source code. However, functions in source code and binary are not one to one, but many to many, which led to challenges in generating training data with complex Swift features. Furthermore, to support other languages besides Swift we would have to create language-specific training data and validation for each target language. However, the benefit is the latency of using the finetuned model. In a comparison of 40 chunks from the Swift packages Files and HTTPTypes (Apple Inc., 2023), GPT-5.6 Sol with high reasoning effort received identical XML inputs and its request latency was 6.4-7.4x slower than the finetuned model's median request latency.

Removing Constraints Designed for Older Models. Earlier versions of BLANKSPACE included dividing up the binary into translation chunks, and making parallel single shot LLM API calls to translate the chunks. Then with coding agents, we added a step of handing the chunks to a workspace where an LLM agent would start translating the chunks. BLANKSPACE also had multiple long prompts with many constraints on each HLL's idiomatic practices and detailed guidance on when and what tools to use, but we observed that with GPT-5.6 Sol providing tools and a simple prompt on how to use the tools sufficed. BLANKSPACE used to store program facts in a database that the agent could query, but we transitioned to a simple file system with an index file that an agent could use to lookup functions.

Agent Reward Hacking. Abuse of tool usage in the LLM agent sandbox led to reward hacking the translation task. For example, in earlier iterations when we allowed web search, once the agent recognized the name of the package it was decompiling, the agent cloned the package's repository on GitHub and provided the code as its "translation" instead of reading the assembly instructions and writing its translation in its workspace. The LLM agent also at times focused on maximizing the feedback from the provided scoring tool, leading to keeping unused functions that increased binary similarity scoring.

B.3 ARTIFACT EXTRACTION

BLANKSPACE generates Ghidra extracted disassembly into address function listings containing recovered signatures, function sizes, and annotated disassembly. Listing 1 shows an example supplied to the coding agent for the stripped Files package.

```
function @ 0xa79c  (224 bytes)
name: static Files.Folder.temporary.getter : Files.Folder

--- disassembly ---
pacibsp
sub sp,sp,#0x40
stp x22,x21,[sp,#0x10]
stp x20,x19,[sp,#0x20]
stp x29,x30,[sp,#0x30]
add x29,sp,#0x30
bl <_NSTemporaryDirectory>
```

```
.......

mov x0,x20
cbnz x21, <LAB_0000a858>
ldp x29,x30,[sp,#0x30]
ldp x20,x19,[sp,#0x20]
ldp x22,x21,[sp,#0x10]
add sp,sp,#0x40
retab
.....
```

Listing 1: Excerpt of Ghidra extracted artifact for `Files.Folder.temporary`.

B.4 Example of Binary Similarity Scoring Feedback

Below is the response the LLM agent receives when calling the custom BlankSpace scoring feedback tool.


```
  {
    "batch_delta": -0.000193479174682043,
    "budget_elapsed_seconds": 1540.1,
    "budget_remaining_seconds": 259.9,
    "calls_remaining": null,
    "candidate_excess_ratio": 0.017104420243433695,
    "compile_success": true,
    "coverage_delta": 0.0,
    "decision": "revert",
    "diagnostics": {
      "bytes": 1931143,
      "elapsed_seconds": 1.1070570410010987,
      "index": "diagnostics/checkpoint-39/INDEX.tsv",
      "status": "complete"
    },
    "estimated_target_byte_weighted_similarity": 0.6660469889016358,
    "reached_function_similarity": 0.7732398393383498,
    "reached_function_similarity_delta": -0.00022461749462021618,
    "target_byte_coverage": 0.8761050608584241
  }
```


Listing 2: Example JSON feedback for a Files decompilation source reconstruction candidate. Compilation succeeds, but the decrease in aggregate similarity triggers reverting back to the previously accepted source reconstruction.

B.5 Example of Assembly Comparison Feedback

After each scored source reconstruction snapshot, BlankSpace generates read-only assembly that compare each target function with its submitted counterpart. Listing 3 shows the normalized comparison for `Files.Folder.temporary`. Lines marked with a minus come from the target binary, and lines marked with a plus come from the submitted reconstruction. Normalization replaces immediates, offsets, and branch destinations with placeholders, so differences caused only by layout drop out. The agent can also read a raw companion diff that keeps literal addresses, operands, and symbol annotations. The reported similarity (0.853) is from BlankSpace's internal scoring (3.4).

```
--- target/42908
+++ submitted/33064
@@ -1,56 +1,51 @@
[... 6 unchanged instructions omitted ...]
 bl <TARGET>
 mov x5,x5
 bl <TARGET>
 mov x3,x0
 bl <TARGET>
-mov x5,x0
-mov x6,x1
+mov x4,x0
+mov x5,x1
 mov x0,x3
 bl <TARGET>

[... 12 unchanged instructions omitted ...]
```

```
 mov w1,#IMM
 mov w2,#IMM
 bl <TARGET>
-mov x4,x0
-stp x5,x6,[x0,#IMM]
-str x3,[x0,#IMM]
-mov x0,x3
-bl <TARGET>
 mov x3,x0
+mov x0,x4
+mov x1,x5
+mov x2,x3
+mov x4,x3
 mov x5,#IMM
 bl <TARGET>
-mov x0,x3
-bl <TARGET>
-mov x0,x4
 cbnz x5,<TARGET>
 ldp x5,x6,[sp,#IMM]
 ldp x4,x3,[sp,#IMM]
 ldp x6,x5,[sp,#IMM]
 add sp,sp,#IMM
 retab
-bl <TARGET>
 adrp x1,<TARGET>

[... 7 unchanged instructions omitted ...]
```

Listing 3: Excerpt of the normalized assembly diff between the target binary and a candidate reconstruction of `Files.Folder.temporary`.

C BLANKSPACEBENCH

C.1 FUNCTIONAL CORRECTNESS TESTS EXAMPLES

C.1.1 SWIFT

Below are examples of functional correctness tests for evaluating reconstructed source code for the Swift package Files (Sundell, 2017). The upstream example is from Files' own test suite, and the downstream example is from the package locheck (Asana), a client package that depends on Files and tests behavior that relies on Files.

```
func testReadingFileAsString() {
    performTest {
        let file = try folder.createFile(named: "string",
        contents: "Hello".data(using: .utf8)!)
        try XCTAssertEqual(file.readAsString(), "Hello")
    }
}
```

Listing 4: Upstream test from Files' own test suite. A file is created with the UTF-8 encoding of `Hello`, and the test checks that `readAsString()` returns the same string.

```
//
//  parseAndValidateAndroidStringsTests.swift
//
//
//  Created by Steve Landey on 9/1/21.
//

import Files
import Foundation
@testable import LocheckLogic
import XCTest

class ParseAndValidateAndroidStringsTests: XCTestCase {
    fileprivate let packageRootPath = URL(fileURLWithPath: #file)
        .pathComponents
        .prefix(while: { $0 != "Tests" })
        .joined(separator: "/")
```

```
        .dropFirst()

    func testParseAndValidateDemoFiles() {
        let problemReporter = ProblemReporter(log: false)
        parseAndValidateAndroidStrings(
            base: try! File(path: "\(packageRootPath)/Examples/strings-base.xml"),
            translation: try! File(path: "\(packageRootPath)/Examples/strings-translation.xml
     "),
            translationLanguageName: "demo",
            problemReporter: problemReporter)

        XCTAssertEqual(problemReporter.problems.count, 10)
        let problems = problemReporter.problems.map(\.problem)

        XCTAssertEqual(problems.map(\.kindIdentifier), [
            "duplicate_entries",
            "duplicate_entries",
            "key_missing_from_translation",
            "key_missing_from_base",
            "key_missing_from_translation",
            "key_missing_from_base",
            "phrase_has_missing_arguments",
            "string_has_invalid_argument",
            "string_has_missing_arguments",
            "string_array_item_count_mismatch",
        ])

        CastAndAssertEqual(problems[0], DuplicateEntries(context: nil, name: "duplicate_entry
     "))
        CastAndAssertEqual(problems[1], DuplicateEntries(context: nil, name: "duplicate_entry
     "))
        CastAndAssertEqual(problems[2], KeyMissingFromTranslation(key: "
     missing_from_translation", language: "demo"))
        CastAndAssertEqual(problems[3], KeyMissingFromBase(key: "missing_from_base"))
    }
}
```

Listing 5: A downstream Swift test for the Files package from Locheck. The test constructs File values for two Android localization XML fixtures and checks that validation produces ten problems with the expected identifiers and selected details.

C.1.2 GOLANG

Listings 6 and 7 show upstream and downstream tests for the Go package bbolt (etcd Authors). The upstream test checks key value insertion and retrieval within a transaction. The downstream test comes from Storm (El Hrychy), a client built on bbolt, and checks stored-byte retrieval and the missing-key error through Storm's API.

```
func TestBucket_Put(t *testing.T) {
    db := btesting.MustCreateDB(t)
    if err := db.Update(func(tx *bolt.Tx) error {
        b, err := tx.CreateBucket([]byte("widgets"))
        if err != nil {
            t.Fatal(err)
        }
        if err := b.Put([]byte("foo"), []byte("bar")); err != nil {
            t.Fatal(err)
        }

        v := tx.Bucket([]byte("widgets")).Get([]byte("foo"))
        if !bytes.Equal([]byte("bar"), v) {
            t.Fatalf("unexpected value: %v", v)
        }
        return nil
    }); err != nil {
        t.Fatal(err)
    }
}
```

Listing 6: Upstream bbolt test. The test creates a bucket, stores the key-value pair foo/bar, and checks that reading the key in the same transaction returns the stored value.

```
func TestGetBytes(t *testing.T) {
    db, cleanup := createDB(t)
    defer cleanup()
```

```

    err := db.SetBytes("trash", "a", []byte("hi"))
    require.NoError(t, err)

    val, err := db.GetBytes("trash", "a")
    require.NoError(t, err)
    require.Equal(t, []byte("hi"), val)

    _, err = db.GetBytes("trash", "b")
    require.Equal(t, ErrNotFound, err)
}
```

Listing 7: Downstream bbolt test from Storm. The test stores and retrieves raw bytes through Storm and checks that a missing key returns ErrNotFound.

C.1.3 C

Listings 8 and 9 show upstream and downstream tests for the C compression library zlib (Gailly & Adler). The upstream test compresses and decompresses a string and checks that the round trip restores it. The downstream test comes from jose (Latchset), whose DEF compression algorithm for JSON Web Encryption is implemented with zlib's deflate and inflate. It compresses and decompresses an RFC 7520 payload through jose's algorithm interface and checks that the output matches the input.

```
static void test_compress(Byte *compr, uLong comprLen, Byte *uncompr,
                          uLong uncomprLen) {
    int err;
    uLong len = (uLong)strlen(hello)+1;

    err = compress(compr, &comprLen, (const Bytef*)hello, len);
    CHECK_ERR(err, "compress");

    strcpy((char*)uncompr, "garbage");

    err = uncompress(uncompr, &uncomprLen, compr, comprLen);
    CHECK_ERR(err, "uncompress");

    if (strcmp((char*)uncompr, hello)) {
        fprintf(stderr, "bad uncompress\n");
        exit(1);
    } else {
        printf("uncompress(): %s\n", (char *)uncompr);
    }
}
```

Listing 8: Upstream zlib test. The test compresses a string with compress(), overwrites the output buffer, decompresses with uncompress(), and checks that the result equals the original string.

```
static void
test(const jose_hook_alg_t *a, bool iter,
     const uint8_t *i, size_t il)
{
    jose_io_auto_t *b = NULL;
    jose_io_auto_t *c = NULL;
    jose_io_auto_t *z = NULL;
    void *buf1 = NULL;
    void *buf2 = NULL;
    size_t blen = 0;
    size_t clen = 0;

    /* Test compression first. */
    b = jose_io_malloc(NULL, &buf1, &blen);
    assert(b);

    z = a->comp.def(a, NULL, b);
    assert(z);

    if (iter) {
        for (size_t j = 0; j < il; j++)
            assert(z->feed(z, &i[j], 1));
    } else {
        assert(z->feed(z, i, il));
    }

    assert(z->done(z));
```

```

    /* Test decompression now. */
    c = jose_io_malloc(NULL, &buf2, &clen);
    assert(b);

    z = a->comp.inf(a, NULL, c);
    assert(z);

    if (iter) {
        uint8_t *m = buf1;
        for (size_t j = 0; j < blen; j++)
            assert(z->feed(z, &m[j], 1));
    } else {
        assert(z->feed(z, buf1, blen));
    }

    assert(z->done(z));

    /* Compare the final output with the original input. */
    assert(clen == il);
    assert(memcmp(buf2, i, il) == 0);
}
```

Listing 9: Downstream zlib test from jose. The test compresses an input through jose's DEF compression algorithm, decompresses the result, and checks that the output length and bytes equal the input. The caller runs it on the RFC 7520 Section 5.9 payload, feeding data both in one call and one byte at a time.