Benchmarking Binary Type Inference Techniques in Decompilers

Vedant Soni
Arizona State University
Tempe, Arizona, USA
vsoni14@asu.edu

Audrey Dutcher
Arizona State University
Tempe, Arizona, USA
dutcher@asu.edu

Tiffany Bao
Arizona State University
Tempe, Arizona, USA
tbao@asu.edu

Ruoyu Wang
Arizona State University
Tempe, Arizona, USA
fishw@asu.edu

Abstract

Decompilation is the process of translating low-level, machine-executable code back into a high-level representation. Decompilers–tools that perform this translation–are essential for reverse engineers and security professionals, supporting critical tasks within their workflows. However, due to the inherent loss of information during compilation as a result of optimizations, inlining, and other compiler-specific transformations, decompiled output is often incomplete or inaccurate.

A central challenge in decompilation is accurate type inference: the reconstruction of high-level type information for variables based on low-level code patterns and memory access behaviors. Despite ongoing advancements in decompilation research, there is a notable lack of comprehensive comparative studies evaluating the type inference capabilities of existing decompilers.

This paper presents a benchmark study of five decompilers, focusing on their ability to infer types at both the function and variable levels. We conduct the evaluation on a dataset of binaries compiled from the Nixpkgs collection at both `-O0` and `-O2` optimization levels, allowing us to assess decompiler performance across unoptimized and optimized executables. The results highlight the relative strengths and weaknesses of each decompiler and identify recurring scenarios in which incorrect type information is produced.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Decompilation, Type Inference, Benchmark

ACM Reference Format:
Vedant Soni, Audrey Dutcher, Tiffany Bao, and Ruoyu Wang. 2025. Benchmarking Binary Type Inference Techniques in Decompilers. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3733822.3764675

1 Introduction

Binary decompilers, or *decompilers* for short, are the software that translate compiled executables ("binaries") into high-level human-readable pseudocode. Decompilers rely only on the information present in a binary to generate the pseudocode. Because compilation is itself a lossy process, the code that decompilers reconstruct is often incorrect or incomplete. A critical step of this reconstruction is binary type inference, i.e. determining the type of a variable or a function that was compiled into a binary. Binary type inference (or *type inference* for short) is crucial for interpreting memory layouts, pointer relationships, and calling conventions. Because type inference is a critical part of binary decompilation, and that decompiled programs are frequently used for vulnerability research, issues in the type inference process can have real-world impact, resulting in missed bugs and vulnerabilities during static analysis [18].

Although there have been considerable advances in the field of type inference through static, dynamic, and learning-based techniques [1, 3, 7, 15, 16, 20, 21, 23, 24, 30, 31], the community still lacks a systematic understanding—and, more importantly, a rigorous and comprehensive evaluation—of existing type inference algorithms and their implementations. This is caused by the following reasons. First, as pointed out by prior work [3], many published techniques do not make their prototypes publicly available, let alone release their source code, making it difficult to reproduce results or compare across approaches. Second, there is no standard benchmark dataset of binaries with ground-truth type information for evaluating type inference capabilities. As a result, existing studies often construct and evaluate their techniques on custom datasets, which may introduce limitations in evaluation or even biases in the results. Last but not least, most studies report the strengths of individual algorithms and tools under specific subcategories of types. However, such subcategories may not be representative in practice, and thus these strengths may not contribute as much as expected benefit to the real-world use case of binary analysis and software understanding, for example, binary code decompilation. For example, Retypd reports 98% accuracy in recovering `const` qualifiers, but this metric reveals little about its overall type-inference capability, because distinguishing `const` from non-`const` types rarely matters for real-world binary reverse engineering. It is essential to conduct a comprehensive evaluation of decompilers' type-inference capabilities for the community to understand the state of the art and promote more rigorous advancement.

This paper aims to benchmark the type inference capabilities of the decompilation tools used in practice. Specifically, we evaluate four widely used decompilers: Hex-Rays [13], Binary Ninja [27], Ghidra [19], and angr [22]. Beyond the native type inference capabilities that these decompilers implement, we also benchmark Retypd [20], which provides a Ghidra plugin that implements the algorithm [12]. Our study targets `x86-64` ELF binaries compiled from C, because most decompilers prioritize these binaries when developing new features. We compile every program at `-O0` and `-O2` to get wider understanding of the type inference capabilities of decompilers across optimization levels.

Guided by these choices, we build a standard dataset of programs compiled from the NixOS package set, Nixpkgs. Each binary is first built with full debug information, from which we extract ground-truth variables and types. We then strip the binaries, apply each testing decompiler to decompile the binaries, normalize the outputs, and finally collect the types inferred by each decompiler. We measure the following four metrics:

- *Coverage*: Percentage of the ground-truth variables successfully recovered by the testing decompiler,
- *Accuracy*: Percentage of ground-truth variables with correctly recovered type,
- *Precision*: Percentage of decompiler-identified variables that exist in ground truth and have a correct type, and
- *False Positive Rate*: Percentage of variables identified by the decompiler that are not present in the ground truth.

We also divide the evaluation on various dimensions to identify specific cases where decompilers struggle. In particular, we find that the presence of *complex types*, that is, structures and arrays, makes certain decompilation tasks easier and others harder.

Contributions. This paper makes the following contributions.

- We create a comprehensive benchmark suite for evaluating the performance of binary type inference techniques as implemented in binary decompilers. Researchers can extend this benchmark suite to support more decompilers in the future.
- We evaluate five binary type inference algorithms and decompilers using our benchmark suite and report our findings.
- We quantify through our survey that decompilers have much room for improvement when compared to a ground truth of source-accuracy.

In the spirit of open science, we make our research artifacts, including evaluation scripts and raw data available at https://github.com/sefcom/decompiler-types-benchmark.

2 Background

Before delving into the details of benchmarking, we first introduce the key concepts that are essential for understanding this paper.

2.1 Compilation and Decompilation

Compiling the source code to a binary is a multi-step process as shown in Figure 1.

- **Pre-processing** removes comments, resolves macros and expands included files.
- **Compilation and Assembly** parses the pre-processed code and converts it to an intermediate representation. This intermediate representation is then optimized for size, speed, and other metrics, and then converted to platform specific assembly code. This assembly is then converted to binary machine code by the assembler.
- **Linking** resolves all external references and links all function calls to their respective definitions.

A decompiler attempts to reverse this process and generate source code, usually a close imitation of C syntax, from the compiled binary. This is also a multi-stage process as shown in Figure 2.

- **Disassembly and IR Lifting** disassembles the executable to assembly code and converts the instructions to an *Intermediate Representation* (IR) which is used for further analysis.
- **Program Recovery** processes the lifted IR to generate function boundaries, reconstruct control flow graph and recover variables and types using static analysis and inference techniques. In addition, some decompilers also optimize the code for readability, for example, by removing dead logic and unnecessary assignments.
- **Code generation** converts the processed IR to C-like source code.

2.2 Debug Information

Debugging With Attributed Record Format (DWARF) is the standard debugging information format embedded in ELF binaries compiled with debug symbols. It stores the source code details for functions, variables, and types through a hierarchy of *Debugging Information Entries* (DIEs). Each DIE captures various attributes such as the name, size, and storage location as well as the underlying type definition, including arrays and structures [9].

2.3 Binary Type Inference

While type inference on source code is a well-studied problem, it becomes significantly more challenging when applied to binaries. The challenge of binary type inference has been widely explored. Existing research primarily focuses on using static and dynamic analysis techniques and machine learning approaches infer types from binaries.

2.3.1 Static Analysis Approaches. Static type inference techniques aim to recover high-level type information solely from a program's binary or intermediate representation (IR) without executing the code. TIE [15] infers variable types by generating constraints from data flows and function calls and solving them using a custom solver. DIVINE [1] employs Value-Set Analysis to group memory accesses into variable-like locations and iteratively refines these groupings to identify aggregate structures, focusing on recovering variables rather than performing full type inference. Osprey [30] uses a probabilistic approach by assigning likelihood scores to possible types in order to determine the most probable type assignments. Retypd [20] and BinSub [24] both generate type constraints and solve them to infer types for variables. Unlike all prior techniques that are based on unification, Retypd and BinSub relies on subtyping, which allows them to handle polymorphic types in binary code.

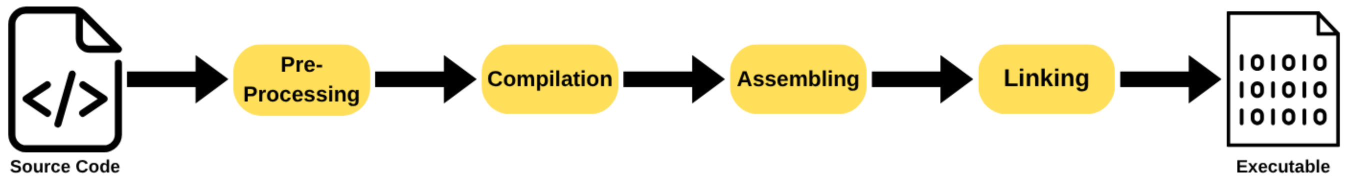


Figure 1: The compilation process for a C program.

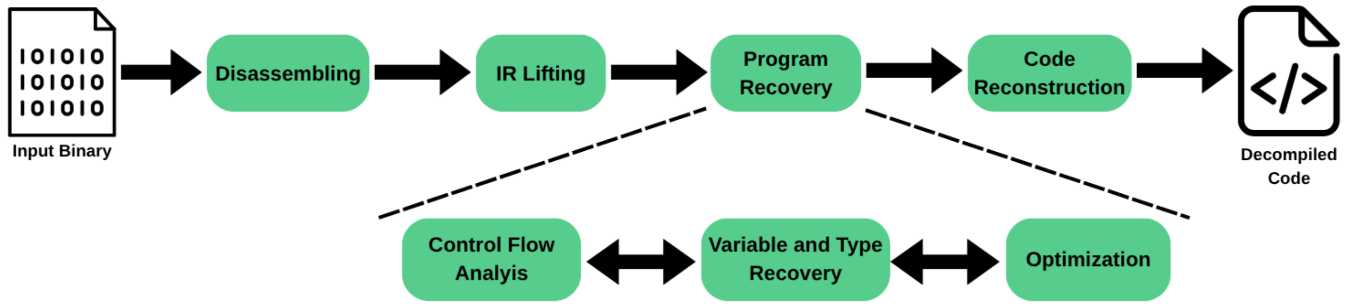


Figure 2: The decompilation process for a compiled binary.

2.3.2 Dynamic Analysis Approaches. Dynamic type inference approaches use programs' runtime information such as memory layouts and access patterns to determine the types of variables. REWARDS [16] instruments program execution to tag memory access with a timestamped type attribute and propagates resolved type information based on the tags. Howard [23] captures memory snapshots during execution and matches them to type templates, effectively recovering array or struct types. DSIbin [21] extends Howard by combining it with DSI [28], a data-structure identification tool, to improve the detection of complex types.

2.3.3 Learning-based Approaches. Learning-based type inference approaches leverage statistical models and learning based algorithms to recover types from binaries. EKLAVYA [7] trains a *RNN* (recurrent neural network) to recover function types. CATI [6] analyzes a contextual window of instructions around each variable, using a multi-stage neural classifier to predict its type. TyGr [31] uses a graph neural network (GNN) to encode data flow information and recover both basic and struct types. TypeFSL [25] applies few-shot learning with interprocedural slicing to generalize from just a handful of labeled examples. ReSym [29] fine-tunes pre-trained *Large Language Models* (LLMs) to produce types and names from decompiler IR and improves their outputs with logic programming.

2.3.4 Type Inference Techniques Used in Decompilers. Given the fact that decompilers must statically decompile binary functions, they typically use static type inference techniques. We examine the type inference techniques that popular open-source decompilers use and present them below. Ghidra conducts a unification-based type propagation and inference. Ghidra-Retypd implements the original retypd algorithm. angr implements and improves retypd for decompiled code. Unfortunately, it is impossible to determine the exact type inference algorithms that close-source decompilers, like Hex-Rays and Binary Ninja, use.

3 Overview

Our benchmark procedure includes the following key steps:

- **Dataset Creation** (Section 3.1). We build a dataset of C binaries, which serves as the basis for our evaluation.
- **Ground Truth Generation** (Section 3.2). We create a ground truth database of variable types from C-language binaries with debug information.
- **Data Extraction** (Section 3.3). We decompile all binaries in the dataset and extract variable and type information for each decompiled function.
- **Data Normalization** (Section 3.4). We normalize the extracted type information to ensure consistency across different decompilers.

3.1 Dataset Creation

For generating the ground truth database, we select 170 C-language binaries compiled from the 25.05 release of Nixpkgs. Nixpkgs was chosen as a dataset due to its highly reproducible and instrumentable nature, allowing us to customize the builds very precisely and easily while being assured that the compilation can be verified. Since Nixpkgs does not provide package categorizations, we picked by hand several packages based on the following criteria:

(1) Presence in the Gentoo Linux `dev-util` packageset was preferred due to this packageset being used as an evaluation dataset in prior work [31].
(2) Packages must have been written in C and produce binaries, as opposed to only libraries.
(3) Packages should compile under our instrumentation.

We built all the selected packages with debug information at both `-O0` and `-O2` optimization levels to account for an unoptimized baseline and a more real-world optimization scenario. We also used the `-fno-omit-frame-pointer` flag during the compilation of `-O2` binaries to preserve frame pointers.

From this point, all ELF executables which were under 800 KB were chosen. We found it necessary to discard files over 800 KB because some decompilers and binary type inference techniques, such as Retypd and angr, would require too much time when processing or decompiling binaries that are too large.

Preserving frame pointers in -O2 optimization. Our study uses stack offsets to identify variables (and their locations). In our experiments, we found that when binaries were compiled at `-O2`, base pointers were often optimized away, which makes it difficult to accurately track variable locations in terms of their stack offsets. To mitigate this issue, we employed the `-fno-omit-frame-pointer` flag when compiling binaries under `-O2` optimization, which preserves base pointers and allowed us to accurately and unambiguously match variables across ground-truth and decompiler outputs.

3.2 Ground Truth Generation

Current research [2] suggests that faithfulness to original source code is a useful metric for evaluating decompiler quality. Following this suggestion to its logical conclusion, we believe that faithfulness to DWARF debugging information corresponding to source-original variables is a good measure of variable recovery and type inference quality.

We additionally define the notion of a variable to evaluate as uniquely identified by an offset within a function's stack frame. This explicitly excludes register- and statically-allocated variables, while variables which share a stack offset at different points in a function into the same definition. We chose to exclude register-allocated variables and collapse stack variables by stack frame offset in order to simplify implementations of this benchmark, as there is not a widely available or agreed upon factor which can be used to disambiguate two such overlapping variables. We chose to exclude statically allocated variables because they cannot be associated with a function for our comparisons between different types of functions.

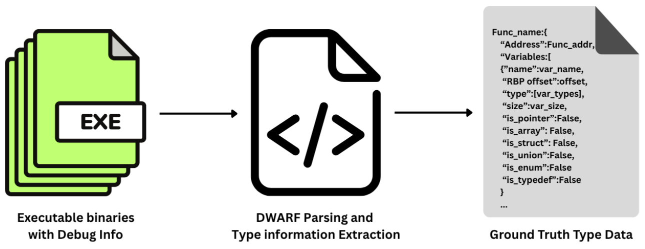
We offer the following motivating examples in support of these decisions:

(1) Saved registers are not useful for a reverse engineer to view in order to comprehend the intended semantics of a function, and they will not be given a DWARF entry.
(2) Variables which are assigned but not used will only be given a DWARF entry if the corresponding memory store actually makes it into the binary.
(3) Two variables which re-use the same compiler stack location will be given two DWARF entries. However, our methodology will collapse these variables into one, on both the ground-truth and decompiled sides.

In accordance with this philosophy, we generate our ground truth database by parsing DWARF debugging information within ELF binaries. We extract the type information by traversing DIEs, identifying function entries, and marshalling associated variable data into a JSON file (Figure 3). Each variable is described by a variable entry in the ground truth database. A variable entry contains variable name, stack frame offset, type information, and size, as well as several Boolean values indicating whether the variable is a pointer, array, struct, union, enum or typedef. We then conduct a type normalization routine on all extracted type information to enable cross-decompiler comparison.

The extraction script also implements handling for few complex types:

- **Arrays** have three additional fields for its length, element type, and element size (indicating base type size).
- **Typedef declarations** have an array containing the complete typedef chain, to allow quick comparison between semantically equivalent types with different representations.

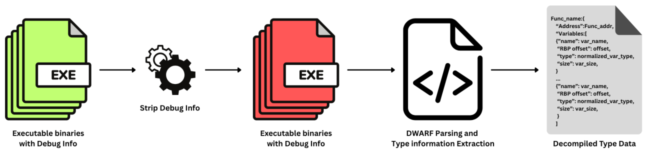

EXE
Executable binaries with Debug Info
DWARF Parsing and Type information Extraction
Ground Truth Type Data

```
Func_name:{
   "Address":Func_addr,
   "Variables:[
   {"name":var_name,
    "RBP offset":offset,
    "type":[var_types],
    "size":var_size,
    "is_pointer":False,
    "is_array": False,
    "is_struct": False,
    "is_union":False,
    "is_enum":False
    "is_typedef":False
   }
   ...
```

Figure 3: Ground truth type data extraction.

3.3 Data Extraction

We begin the evaluation for each decompiler by decompiling and dumping each of the stripped binaries using decompiler APIs (Figure 4). As there is no standalone tool for Retypd, the inferred types are extracted using the `ghidra-retypd` Ghidra plugin [12].


EXE
Executable binaries with Debug Info
Strip Debug Info
EXE
Executable binaries with Debug Info
DWARF Parsing and Type information Extraction
Decompiled Type Data

```
Func_name:{
   "Address":Func_addr,
   "Variables:[
   {"name": var_name,
    "RBP offset": offset,
    "type": normalized_var_type,
    "size": var_size,
   }
   ...
   {"name": var_name,
    "RBP offset": offset,
    "type": normalized_var_type,
    "size": var_size,
   }
   ]
```

Figure 4: Decompilation and type data extraction.

3.4 Type Normalization

Extracted type annotations from various decompilers and from DWARF-based ground truth often differ in naming conventions, qualifiers, and syntax. To allow for a fair comparison, we apply the following normalization procedure to transform all variants of a type into a standard form:

- **Remove C qualifiers** such as const, struct, volatile, signed, and unsigned.
- **Map synonyms**, or common decompiler-specific types, to standard C types. Table 1 and Table 2 show motivating examples.

4 Evaluation Metrics

We establish a set of metrics to evaluate the performance of decompilers in binary type inference. Because variable information is elided in binaries, decompilers must first reason about the locations of variables before inferring their types. Decompilers cannot infer the type of a variable if they fail to recognize it in the first

Table 1: 64-bit integer type normalization.

Decompiler	Type		
Hex-Rays	`_QWORD`	$\Longrightarrow$	`Long Long`
Ghidra & Retypd	`undefined8`		
Binary Ninja	`uint64_t`		

Table 2: 32-bit integer type normalization.

Decompiler	Type		
Hex-Rays	`_DWORD`	$\Longrightarrow$	`Int`
Ghidra & Retypd	`undefined4`		
Binary Ninja	`int32_t`		

place. As such, we must measure the performance of both variable recognition (Section 4.1) and variable type inference (Section 4.2).

4.1 Variables

First, we measure the presence and absence of variables with respect to the ground truth. We establish four metrics: Accuracy, Precision, False Positive Rate (FPR), and Coverage.

Coverage assesses the comprehensiveness of the decompiler by measuring how many variables from the ground truth it successfully identifies, regardless of type correctness. It measures the completeness of the type inference process.

$$\text{Coverage} = \frac{|\,\text{Var}_{\text{gt}} \cap \text{Var}_{\text{dec}}\,|}{|\,\text{Var}_{\text{gt}}\,|}$$

where Var_{gt} is the set of variables in the ground truth and Var_{dec} is the set of variables identified by a decompiler.

Accuracy measures the proportion of variables correctly identified and typed by the decompiler relative to the total number of variables present in the ground truth. It provides insight into the overall correctness of type inference.

$$\text{Accuracy} = \frac{|\,\text{Var}_{\text{corr}}\,|}{|\,\text{Var}_{\text{gt}}\,|}$$

where Var_{corr} is the set of variables that are both correctly recognized and correctly typed by a decompiler.

Precision evaluates how reliable the decompiler's output is by measuring the fraction of correctly identified and typed variables against all variables reported by the decompiler. A higher precision indicates fewer incorrect predictions.

$$\text{Precision} = \frac{|\,\text{Var}_{\text{corr}}\,|}{|\,\text{Var}_{\text{dec}}\,|}$$

False Positive Rate (FPR) quantifies the rate at which a decompiler incorrectly identifies variables that do not exist in the ground truth. It measures the tendency of the tool to produce erroneous outputs.

$$\text{FPR} = \frac{|\,\text{Var}_{\text{dec}} - \text{Var}_{\text{gt}}\,|}{|\,\text{Var}_{\text{dec}}\,|}$$

where $\text{Var}_{\text{dec}} - \text{Var}_{\text{gt}}$ is the set of variables that are identified by the decompiler but do not exist in the ground truth.

Together, these metrics provide a balanced evaluation framework, highlighting reliability (Accuracy and Precision) while also addressing completeness (Coverage) and propensity for errors (False Positive Rate).

4.2 Variable Types

For variables whose storage locations match between the ground truth and the decompiler output, we conduct a detailed comparison between the corresponding types. We break the notion of binary type inference performance into several distinct metrics, different for each variety of type:

Primitive Types. We define two metrics: type inference accuracy, which measures the correct identification of the specific type, and size inference accuracy, evaluating the correctness of the identified size.

Pointers. We calculate the accuracy of pointer identification and correctness of the identified pointer targets. Additionally, a size inference accuracy metric is computed, similar to primitive types.

Structures. We calculate a single metric which indicates whether each struct was identified correctly. This focuses on how effectively the decompiler recognizes structure variables from variable uses across different functions.

Arrays. We establish two metrics for arrays. One is accuracy of complete identification, where both the base type and array length are correctly inferred. The other is accuracy of identifying arrays with the correct base element type but wrong length.

This comprehensive type-level evaluation provides deeper insights into specific type handling capabilities of the decompilers, highlighting their effectiveness across diverse variable types.

5 Evaluation

We conduct two experiments. The first one is a general analysis of binary type inference, where we evaluate the quality of binary type inference in decompilers. The second experiment is type-specific analysis, where we use the previously described metrics to score each decompiler on its treatment of individual types.

As noted in Section 3.1, some tools timed out when processing larger or more complex binaries. Although we imposed a size restriction to mitigate this issue, a few binaries still caused timeouts. Consequently, our final evaluation includes only the binaries successfully processed by all tools, resulting in 94 binaries for `-O0` and 106 binaries for `-O2`.

5.1 Evaluated Decompilers

Because of the constraints of our large-scale experiment, we chose the decompilers to evaluate for their scriptable APIs. We identified five decompilers which we could harness to produce variable recovery and type inference information. This collection represents a realistic sample of the state of the art of decompilation.

(1) Hex-Rays Decompiler is a commercial product, part of the IDA Pro interactive disassembler. We used version 9.0.
(2) Binary Ninja is a commercial product developed by Vector 35. We used version 5.1.

Table 3: Variable location evaluation for all variables.

Decompilers	Var_{dec}	$\lvert Var_{dec} - Var_{gt} \rvert$	$\lvert Var_{gt} - Var_{dec} \rvert$
`-O0` (Var_{gt} = 28,564)			
angr	23,032	7,590	13,122
Binary Ninja	14,139	5,208	19,633
Ghidra	15,159	6,345	19,750
Hex-Rays	18,926	3,978	13,616
Retypd	28,461	18,638	18,741
`-O2` (Var_{gt} = 3,492)			
angr	11,371	8,691	812
Binary Ninja	8,036	5,892	1,348
Ghidra	8,728	6,644	1,408
Hex-Rays	7,948	4,875	419
Retypd	57,944	56,261	1,809

(3) Ghidra is an open-source reverse engineering suite, primarily developed by the United States National Security Agency. We used version 11.3.
(4) angr is an open-source binary analysis research platform primarily developed by various research institutions in the United States. We used version 9.2.169.
(5) Retypd is a binary type recovery algorithm primarily developed by GrammaTech [20]. GrammaTech released an open-source implementation of the algorithm [11] as well as a Ghidra plugin [12] on GitHub. We used the most recent commit of both repositories as of January 2025.

5.2 General Evaluation

Table 3 and Table 4 show the evaluation result against all variables. angr and Hex-Rays achieve the highest variable coverage with only 13,122 and 13,616 missed variables out of 28,564 ground-truth variables for `-O0` binaries and 812 and 419 missed variables out of 3,492 ground-truth variables for `-O2` binaries, respectively. Hex-Rays also leads in type identification accuracy, correctly identifying the types of 10,194 variables, for `-O0` binaries and 1,737 variables for `-O2` binaries.

Due to the intrinsic difficulty with inferring complex types (e.g., arrays and structs) for stack variables, some decompilers struggle with them and may break them into their constituent elements. This can lead to high FPRs. To tackle this bias, we divide the functions into two sets, $Func_P$ (functions with only primitive-typed variables) and $Func_C$ (functions with complex-typed variables). Results in Table 5 and Table 6 show that only a few decompilers see a drastic decrease in the number of false positives.

5.2.1 Variable Coverage. Variable coverage reflects how well each decompiler identifies the *presence* of variables from functions, and the results are shown in Figure 5 and Figure 6. angr and Hex-Rays stands out by a large margin, achieving >50% variable recovery for `-O0` binaries and >75% variable recovery for `-O2` binaries. Surprisingly, variable coverage across all decompilers increases when the dataset is restricted to only functions with complex types. This suggests that these complex variables are in some sense easier for decompilers to assess existence.

5.2.2 Type Inference Accuracy. Figure 7 and Figure 8 show how accurately different decompilers infer variable *types*. Out of all the decompilers evaluated, Hex-Rays performs the best across all categories of variables, achieving an overall accuracy of 35.69% for `-O0` binaries and 49.74% for `-O2` binaries.

Closely following Hex-Rays is angr, achieving an overall accuracy of 29.41% for `-O0` binaries and 28.52% for `-O2` binaries.

5.2.3 Type Inference Precision. Figure 9 and Figure 10 present the precision—the proportion of correctly inferred variable types relative to all variables they identify—of each decompiler across three function subsets. As expected, precision rises markedly for $Func_P$ compared to $Func_C$, highlighting the relative ease of inferring simple data types versus complex structures.

Out of all decompilers, Hex-Rays achieves the highest precision across all function sets and optimizations, with 54.92% overall for `-O0` binaries and 21.85% overall for `-O2` binaries. Ghidra follows,

Table 4: Variable type evaluation for all variables.

Decompilers	Var_{corr}	Var_{incorr}	$Var_{corr} + Var_{incorr}$
`-O0` (Var_{gt} = 28,564)			
angr	8,402	7,040	15,442
Binary Ninja	5,337	3,594	8,931
Ghidra	6,161	2,653	8,814
Hex-Rays	10,194	4,754	14,948
Retypd	5,789	4,034	9,823
`-O2` (Var_{gt} = 3,492)			
angr	996	1,684	2,680
Binary Ninja	941	1,203	2,144
Ghidra	1,042	1,042	2,084
Hex-Rays	1,737	1,336	3,073
Retypd	601	1,082	1,683

Table 5: Variable location evaluation for function sets.

Decompilers	Var_{dec}	$\lvert Var_{dec} - Var_{gt} \rvert$	$\lvert Var_{gt} - Var_{dec} \rvert$
`-O0`			
$Func_P$ (Var_{gt} = 19,898)			
angr	11,672	2,310	10,536
Binary Ninja	6,011	1,095	14,982
Ghidra	5,839	1,373	15,432
Hex-Rays	10,034	1,411	11,275
Retypd	17,580	11,361	13,679
$Func_C$ (Var_{gt} = 8,666)			
angr	11,360	5,280	2,586
Binary Ninja	8,128	4,113	4,651
Ghidra	9,320	4,972	4,318
Hex-Rays	8,892	2,567	2,341
Retypd	10,881	7,277	5,062
`-O2`			
$Func_P$ (Var_{gt} = 1,333)			
angr	4,957	4,086	462
Binary Ninja	3,068	2,579	844
Ghidra	3,627	3,176	882
Hex-Rays	3,515	2,415	233
Retypd	40,141	39,691	883
$Func_C$ (Var_{gt} = 2,159)			
angr	6,414	4,605	350
Binary Ninja	4,968	3,313	504
Ghidra	5,101	3,468	526
Hex-Rays	4,433	2,460	186
Retypd	17,803	16,570	926

Table 6: Variable type evaluation for function sets.

Decompilers	Var_{corr}	Var_{incorr}	Var_{corr} + Var_{incorr}
-O0			
$Func_P$ (Var_{gt} = 19,898)			
angr	5,480	3,882	9,362
Binary Ninja	3,245	1,671	4,916
Ghidra	3,462	1,004	4,466
Hex-Rays	6,066	2,557	8,623
Retypd	3,957	2,262	6,219
$Func_C$ (Var_{gt} = 8,666)			
angr	2,922	3,158	6,080
Binary Ninja	2,092	1,923	4,015
Ghidra	2,699	1,649	4,348
Hex-Rays	4,128	2,197	6,325
Retypd	1,832	1,772	3,604
-O2			
$Func_P$ (Var_{gt} = 1,333)			
angr	444	427	871
Binary Ninja	317	172	489
Ghidra	292	159	451
Hex-Rays	717	383	1,100
Retypd	207	243	450
$Func_C$ (Var_{gt} = 2,159)			
angr	552	1,257	1,809
Binary Ninja	624	1,031	1,655
Ghidra	750	883	1,633
Hex-Rays	1,020	953	1,973
Retypd	394	839	1,233

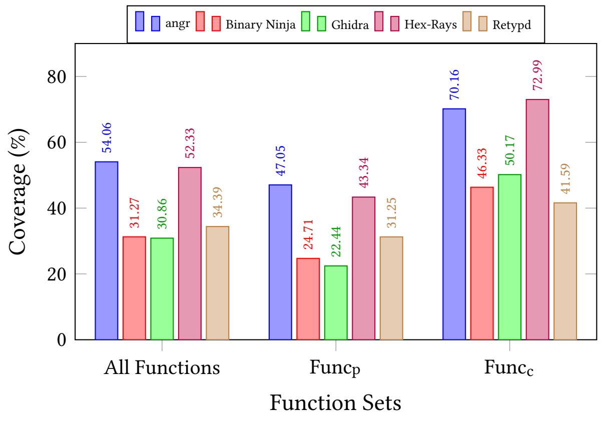


Figure 5: Coverage comparison of type inference across function sets for -O0 optimization.

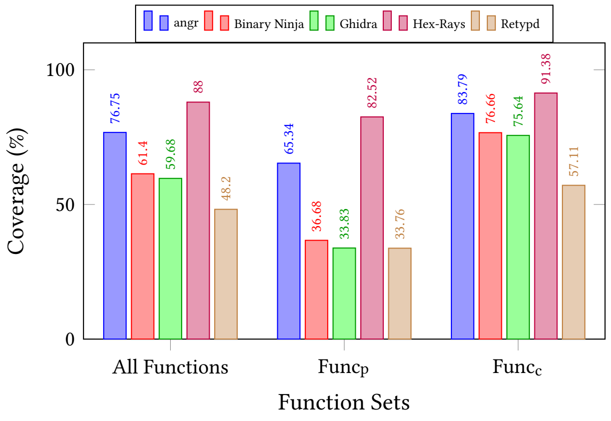


Figure 6: Coverage comparison of type inference across function sets for -O2 optimization.

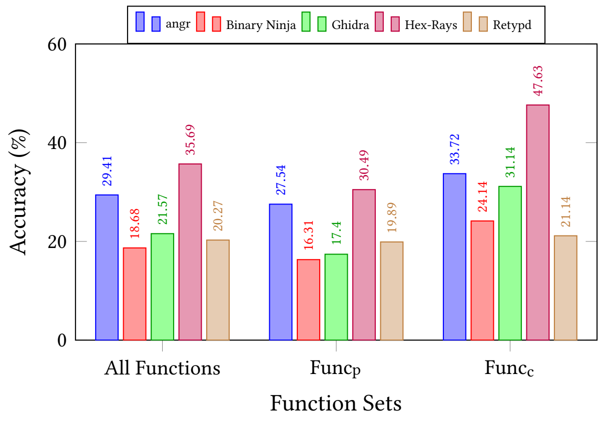


Figure 7: Accuracy comparison of type inference across function sets for -O0 optimization.

attaining 40.64% overall for -O0 binaries and 11.94% overall for -O2 binaries. angr and Binary Ninja exhibit moderate performance overall, with overall lower scores than Hex-Rays or Ghidra, while Retypd trails the other tools by a wide margin.

5.2.4 Type Inference False Positive Rate. Figure 11 and Figure 12 show false positive rates in type inference by each decompiler. As expected, all decompilers saw a drop in their false positive rates in $Func_P$ for -O0, highlighting the universal struggle of identifying complex data types and breaking them into their constituent elements. Surprisingly, the same trend is not observed for -O2, where false positive rates decrease for all decompilers for $Func_C$ as compared to $Func_P$.

5.3 Type-Specific Analysis

In this subsection we report binary type inference performance for individual type in each decompiler.

5.3.1 Primitive Types and Pointers. Primitive types are the most basic types (e.g., char, int, long long, float, etc.) that the C language provides as built-ins. Decompilers are typically accurate at identifying the type and size of these variables (Table 7). However, when accounting for pointers, the picture is less clear. Unlike scalar types such as int and char, which directly store values, a pointer holds a memory address referencing another data object, which is indistinguishable from a pointer-sized integer in machine code. Moreover, correctly inferring a pointer's target adds an extra layer of complexity, as the decompiler must analyze dereference operations and pointer arithmetic to reconstruct the intended target type. Table 8 shows that, as a result, decompilers often misidentify

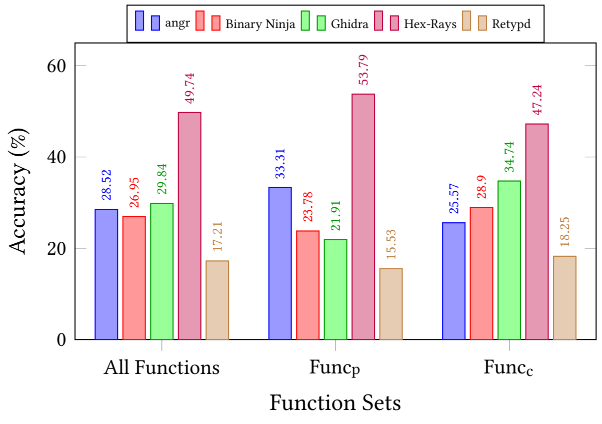


Figure 8: Accuracy comparison of type inference across function sets for `-O2` optimization.

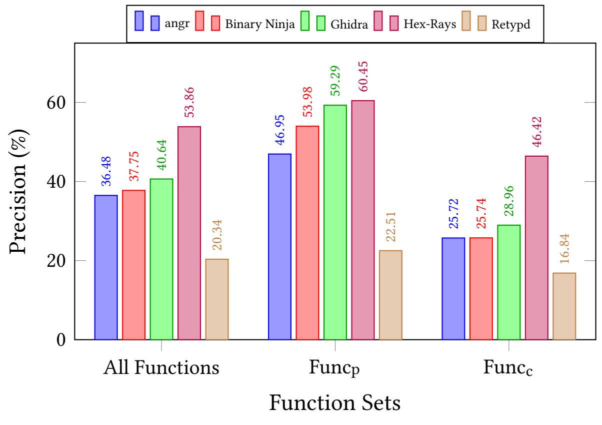


Figure 9: Precision comparison of type inference across function sets for `-O0` optimization.

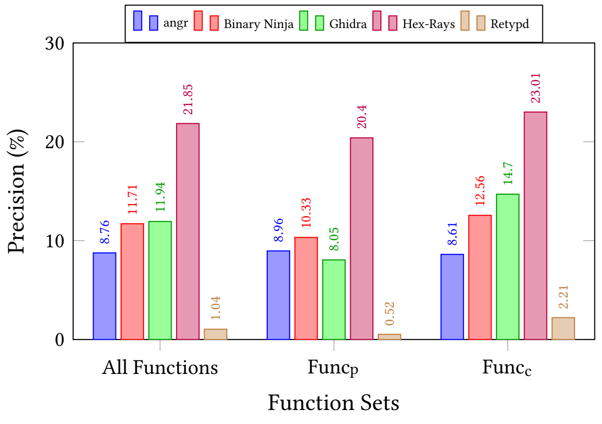


Figure 10: Precision comparison of type inference across function sets for `-O2` optimization.

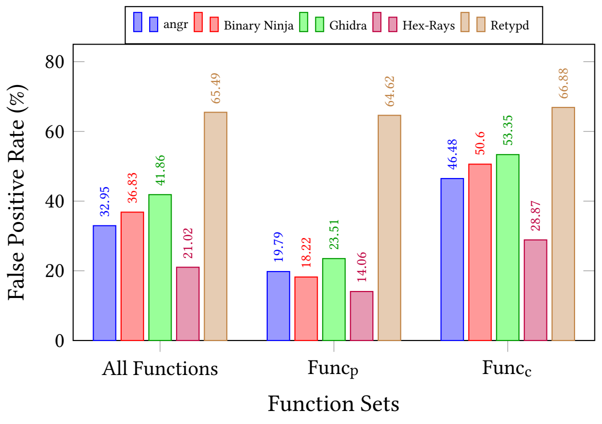


Figure 11: False positive rate comparison of type inference across function sets for `-O0` optimization.

pointers—either assigning the wrong target or mistaking them for non-pointer types.

5.3.2 *Complex Types.*

We term structs and arrays as complex types, which are types defined by aggregating multiple primitive types. Because structs can contain many different types, the decompiler must infer a type for each field of a struct. In comparison, arrays comprise a single primitive type, so once the type of a single element and the array size are inferred, the full array type is known. Despite their differences, arrays and structs share a commonality: To refer to different elements of either, the program must use a base pointer and an offset. Additionally, compilers can add padding to the type structures to make sure that memory accesses are aligned to architectural standards.

As a result of these factors, decompilers often struggle to accurately reconstruct array and struct layouts, as Table 9 and Table 10 show.

5.3.3 *Primitive Types.*

This section evaluates each decompiler's capability in recovering the aforementioned properties for primitive type variables by breaking the analysis into two parts: (1) *Correct Type Inference* and (2) *Correct Size Inference*. The first part analyzes whether a decompiler assigns the correct type (e.g., `int`) to a variable, while the second part investigates whether the decompiler infers the correct size (byte-width) of the variable. Pointers add an extra layer of complexity in the decompilation process, since a decompiler needs to identify a variable as a pointer and also determine the target type for the pointer. To represent this challenge, we measure how many variables identified by the decompiler were identified as pointers and how many were identified with correct target type.

char, int, and long long. Figure 13 compares each decompiler's ability to infer correct types for variables of common primitive types (`char`, `int`, `long long`), while Figure 14 compares their accuracy in inferring the corresponding variable sizes across `-O0` and `-O2` optimizations. angr and Hex-Rays outperform other decompilers in type inference accuracy. All decompilers achieve high accuracy in size inference, with the only exception being Retypd, which struggles with `long long` types.

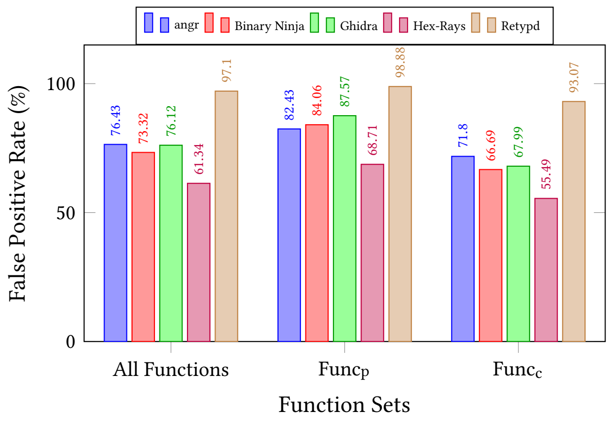


Figure 12: False Positive Rate comparison of type inference across function sets for -O2 optimization.

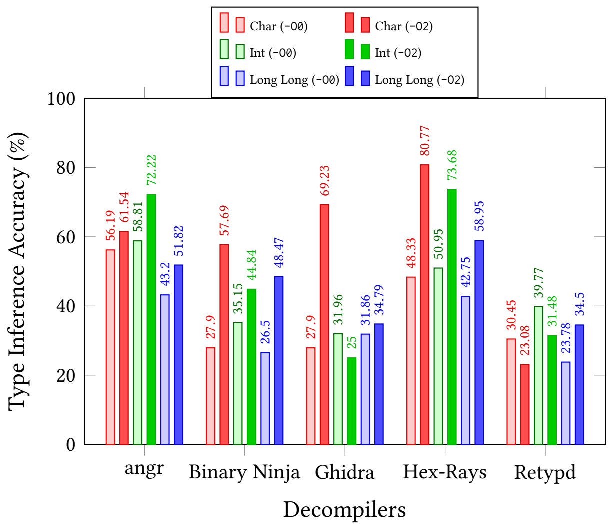


Figure 13: Type inference comparison across char, int, and long long for -O0 and -O2 optimizations.

Pointers. Figure 15 compares pointer identification and correct-target resolution, while Figure 16 compares pointer-size inference, for binaries compiled at -O0 and -O2. Hex-Rays outperforms other tools in pointer identification, successfully identifying pointers for 27.57% of variables, and accurately resolving correct pointer targets for 19.24% of variables for -O0 binaries, surpassing its closest competitor, angr, by a small margin. For -O2 binaries, Hex-Rays retains the lead, identifying 49.17% of pointers and correctly resolving 40.59% of targets; Ghidra follows with 31.27% identified and 25.91% resolved, outperforming angr (26.65% identified; 12.21% resolved). However, the relatively low identification rates across all decompilers highlight a key aspect for improvement in binary type inference. For size inference, performance is generally strong across all decompilers.

Table 7: Type-wise evaluation results.

Decompilers	Var_{dec}	Var_{corr}	Var_{incorr}	$\mid Var_{gt} - Var_{dec} \mid$
-O0				
char (Var_{gt} = 509)				
angr	286	286	0	223
Binary Ninja	143	142	1	366
Ghidra	144	142	2	365
Hex-Rays	251	246	5	258
Retypd	162	155	7	347
int (Var_{gt} = 7,739)				
angr	4,613	4,551	62	3,126
Binary Ninja	2,790	2,720	70	4,949
Ghidra	2,532	2,473	59	5,207
Hex-Rays	4,119	3,943	176	3,620
Retypd	3,096	3,078	18	4,643
long long (Var_{gt} = 5,336)				
angr	3,008	2,305	703	2,328
Binary Ninja	1,633	1,414	219	3,703
Ghidra	1,947	1,700	247	3,389
Hex-Rays	3,051	2,281	770	2,285
Retypd	1,891	1,269	622	3,445
-O2				
char (Var_{gt} = 26)				
angr	18	16	2	8
Binary Ninja	17	15	2	9
Ghidra	22	18	4	4
Hex-Rays	26	21	5	0
Retypd	17	6	11	9
int (Var_{gt} = 756)				
angr	596	546	50	160
Binary Ninja	416	339	77	340
Ghidra	334	189	145	422
Hex-Rays	693	557	136	63
Retypd	317	238	79	439
long long (Var_{gt} = 687)				
angr	540	356	184	147
Binary Ninja	426	333	93	261
Ghidra	358	239	119	329
Hex-Rays	625	405	220	62
Retypd	357	237	120	330

Table 8: Pointer evaluation results.

Decompilers	Var_{dec}	Var_{corr}	Var_{incorr_target}	Var_{not_ptr}	$\mid Var_{gt} - Var_{dec} \mid$
-O0 (Var_{gt} = 12,600)					
angr	5,416	1,239	1,567	2,610	7,184
Binary Ninja	2,500	890	928	682	10,100
Ghidra	2,873	1,456	691	726	9,727
Hex-Rays	5,462	2,424	1,050	1,988	7,138
Retypd	3,736	1,115	756	1,865	8,864
-O2 (Var_{gt} = 1,212)					
angr	742	148	175	419	470
Binary Ninja	466	184	115	167	746
Ghidra	606	314	65	227	606
Hex-Rays	1,011	492	104	415	201
Retypd	480	104	70	306	732

5.3.4 Complex Types. This section focuses on evaluating each decompilers type inference capabilities for complex types: arrays and structs. Structs are usually identified by decompilers based on how a variable is used across functions and matching function argument

Table 9: Array evaluation results. $\text{Var}_{\text{corr\_len}}$**: arrays with correctly inferred length;** $\text{Var}_{\text{incorr\_len}}$**: arrays with correct base but incorrect length;** $\text{Var}_{\text{incorr}}$**: arrays with incorrect type inference.**

Decompilers	Var_{dec}	$\text{Var}_{\text{corr\_len}}$	$\text{Var}_{\text{incorr\_len}}$	$\text{Var}_{\text{incorr}}$	$\lvert \text{Var}_{\text{gt}} - \text{Var}_{\text{dec}} \rvert$
		-O0 (Var_{gt} = 535)			
angr	530	11	91	428	5
Binary Ninja	506	19	104	383	29
Ghidra	498	42	189	267	37
Hex-Rays	510	46	184	280	25
Retypd	139	6	30	103	396
		-O2 (Var_{gt} = 453)			
angr	451	34	17	400	2
Binary Ninja	448	40	54	354	5
Ghidra	357	53	94	210	96
Hex-Rays	370	66	99	205	83
Retypd	203	1	12	190	250

Table 10: Struct evaluation results.

Decompilers	Var_{dec}	Var_{corr}	$\text{Var}_{\text{incorr}}$	$\lvert \text{Var}_{\text{gt}} - \text{Var}_{\text{dec}} \rvert$
		-O0 (Var_{gt} = 800)		
angr	770	20	750	30
Binary Ninja	756	63	693	44
Ghidra	756	150	606	44
Hex-Rays	747	162	585	53
Retypd	302	36	266	498
		-O2 (Var_{gt} = 562)		
angr	546	15	531	16
Binary Ninja	549	83	466	13
Ghidra	476	84	392	86
Hex-Rays	481	96	385	81
Retypd	353	21	332	209

signatures. Although angr and Retypd are able to recover struct layouts based on code patterns, evaluation of these reconstructed structs is out of scope of this paper. We focus on the named structs directly identified by the decompilers (eg. `FILE, stat` etc.) to evaluate the struct identification accuracy. Evaluating arrays is more complicated, as decompilers need to infer not only the base type of the array but also the length. Correspondingly, we report 2 metrics, array variables that were identified correctly (same element type and length) and array variables that were identified as arrays, had the same element type but had the wrong length.

Structs. Figure 17 shows the percentage of structs correctly identified by different decompilers across -O0 and -O2 optimizations. At -O0, Hex-Rays leads, correctly identifying 20.25% of structs. In contrast, angr and Retypd struggle considerably, with low accuracies of only 2.50% and 4.50%, respectively. For -O2 binaries, the pattern persists but the gap narrows: Hex-Rays achieves 17.08%, Ghidra 14.95%, and Binary Ninja 14.77%, while angr and Retypd remain low at 2.67% and 3.74%. The results highlight that struct inference remains a challenging task for most decompilers.

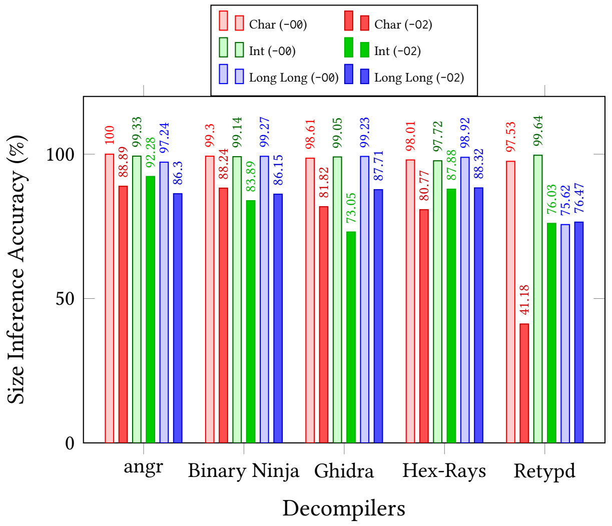


Figure 14: Size inference comparison across char, int, and long long for -O0 and -O2 optimizations.

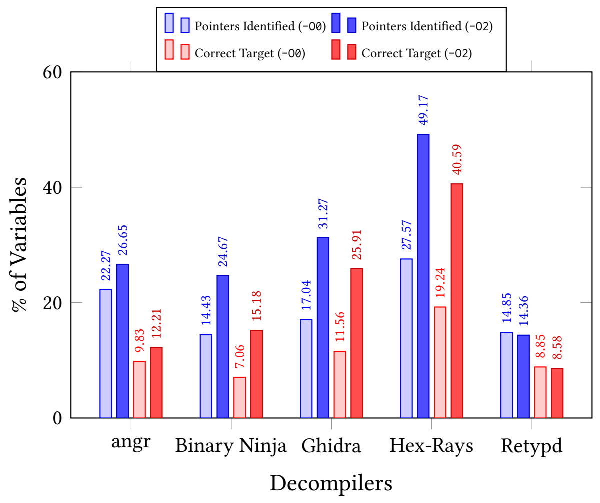


Figure 15: Pointer identification and target resolution accuracy for -O0 and -O2 optimizations.

Arrays. Figure 18 illustrates the accuracy of array identification across different decompilers and optimizations (-O0 and -O2), highlighting arrays correctly identified and those identified with correct base addresses but incorrect lengths. At -O0, Hex-Rays and Ghidra identify the highest total number of arrays (42.99% and 43.18% respectively), although the majority have incorrect lengths despite correct base addresses. For -O2 binaries, the trend is similar, with Hex-Rays and Ghidra again identifying the most arrays (36.42% and 32.45% respectively). It is worth noting that most decompilers struggle to accurately infer array lengths.

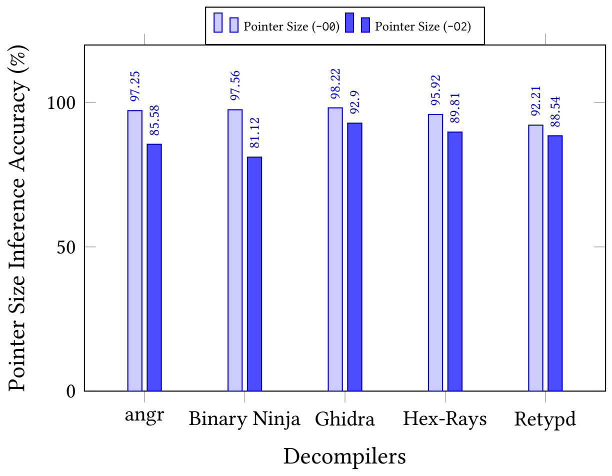


Figure 16: Pointer size inference accuracy for `-O0` and `-O2` optimizations.

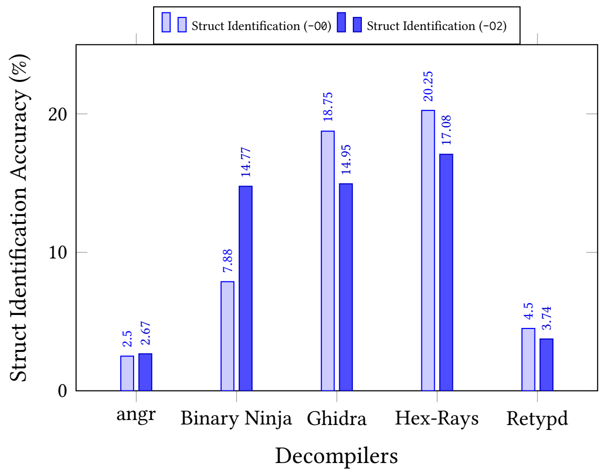


Figure 17: Struct identification accuracy for `-O0` and `-O2` optimizations.

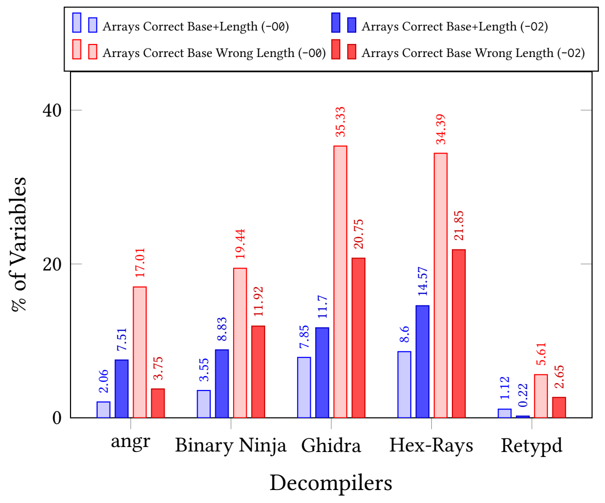


Figure 18: Array identification and length inference accuracy for `-O0` and `-O2` optimizations.

5.4 Case Study: Binary Ninja's Low Coverage

Binary Ninja seems to completely discard some stack variables even though they are in fact written. for example, in the `[` (synonym for `test` in shell scripting conditional expressions) binary, at address `0x403a59`, the result of `strcoll` is written to the stack at `[rbp - 0x1b0]`, and then immediately read out again at `0x403a68`. This results in the creation of a variable in Binary Ninja's low- and mid-level IRs, but in the high level IR and pseudo-C, there is no sign of a stack write–only that the value is stored to a register variable and then discarded after being passed on. This omission is a prescient contributor to the low coverage score that Binary Ninja receives in our evaluation, and furthermore that it does in fact represent a deficiency in the decompiler, since an analyst working on a binary exploitation task will want to know at least the memory regions touched by a piece of pseudo-C from looking at it. This would be less of an issue if the pseudo-C was agnostic to storage locations, but it does in fact show a variable with register storage being written to and not a variable with stack storage being written to.

6 Discussion

In this section, we discuss the limitations of our study and potential avenues for future work.

6.1 Limitations

Our study has several limitations that should be considered when interpreting the results. First off, in order to take into account the subpar performance of angr and Ghidra-Retypd, we only chose binaries smaller than 800 KB, which might have biased the dataset against bigger and more complex real-world binaries. Second, our experiments are limited to `x86-64` Linux binaries, so the findings may not generalize to other architectures (e.g., ARM and AArch64) or platforms (e.g., Windows). We also note that released versions of code (e.g., the Retypd Ghidra plugin [12]) may not completely or accurately reflect the originally published research (e.g., the Retypd paper [20]) due to the potential existence of software bugs. Future work that improves upon this study may address these limitations.

6.2 Future Work

Successors to this research could perform similar studies that address the limitation of this one, including larger binaries, evaluating across more optimization levels, and architectures beyond `x86-64`. Additionally, based on the fact that the two research prototypes evaluated in this survey, angr and Retypd, were the only decompilers able to generate struct layouts from scratch instead of only identifying usages of structs from popular libraries, it seems that this capability is an emerging technology which has not yet made its way into commercial offerings. As this technique matures, it should be possible to perform a similar comparative analysis on decompilers' ability to perform arbitrary struct recovery. Hopefully, this work provides a solid foundation on which we can improve in decompiler performance in a principled manner.

7 Related Work

Over the past decade, researchers have conducted several surveys, benchmarks, and comparative studies for decompilers, each focusing on different aspects.

The work *Type Inference on Executables* [4] presents a comprehensive survey of binary type-recovery approaches, examining static, dynamic, and hybrid analysis techniques. This survey highlights many the challenges inherit to recovering complex data types from stripped binaries. The work also highlights the need for standardized benchmarks and quantitative metrics to fairly compare type-inference algorithms.

There have been other survey-based studies such as Fawareh et. al's 2024 survey [10], which provides a broad survey of modern decompilation techniques, comparing decompilers and highlighting recurring error patterns in their outputs. Dramko et. al's 2024 survey [8] conducts an in-depth evaluation of C decompiler output across multiple tools using open-coding analysis, resulting in a detailed taxonomy of decompilation issues that classifies structural inaccuracies.

Liu et. al's 2020 evaluation [17] tested C decompilers by recompiling their outputs and checking behavioral consistency with the original binaries. Cao et. al's 2024 evaluation [5] assessed mainstream decompilers based on their semantic consistency and code readability. It also reports that despite improvements over the years, decompiled C code often still resembles compiler-generated patterns rather than human-written code.

Kline et. al [14] developed a framework for quantitatively assessing a decompiler's ability to recover source-level constructs (functions, variables, data types) with ground-truth comparisons. Tan et. al [26] introduced DecompileBench, a large-scale open benchmark suite for decompiler evaluation. DecompileBench compared six commercial decompilers and six learning-based code generation models under consistent conditions, providing insights into both functional correctness and code understandability.

8 Conclusion

In this paper, we conducted a benchmarking study on the type inference accuracy of five decompilers and type inference solutions: Hex-Rays, Binary Ninja, Ghidra, angr, and Retypd (implemented as Ghidra plugin). Using a dataset of binaries compiled from Nixpkgs at different optimization levels, we extracted type information from debug symbols and compared it with the types inferred by the decompilers. Our results revealed significant variation among the tools. Hex-Rays demonstrated the best overall performance, achieving the highest type inference accuracy and precision, particularly excelling in primitive type inference. angr closely followed Hex-Rays, exhibiting slightly better coverage and accuracy for primitive types but struggling with complex types. Ghidra showed strength in handling complex data structures such as structs and arrays, while Binary Ninja maintained consistent performance across various scenarios. Retypd displayed notable limitations across all metrics. Overall, our findings highlight the distinct strengths and weaknesses of current state-of-the-art decompilers.

Acknowledgments

We would like to thank our anonymous reviewers and shepherd, Dr. Edward J. Schwartz, for their valuable and constructive feedback, which significantly improved the quality of this paper. This material is based upon work supported by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, and the National Science Foundation (NSF) under Awards No. 2146568 and 2232915. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ARPA-H or NSF.

References

[1] Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation*, Byron Cook and Andreas Podelski (Eds.). Springer, Berlin, Heidelberg, 1–28. https://doi.org/10.1007/978-3-540-69738-1_1

[2] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 361–378.

[3] Jay Bosamiya, Maverick Woo, Bryan Parno, and Reviewing Model. 2025. TRex: Practical type reconstruction for binary code. In *USENIX Security Symposium.*

[4] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *Comput. Surveys* 48, 4 (May 2016), 1–35. https://doi.org/10.1145/2896499

[5] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. 2024. Evaluating the Effectiveness of Decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 491–502. https://doi.org/10.1145/3650212.3652144

[6] Ligeng Chen, Zhongling He, and Bing Mao. 2020. CATI: Context-Assisted Type Inference from Stripped Binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 88–98. https://doi.org/10.1109/DSN48063.2020.00028

[7] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[8] Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A Taxonomy of C Decompiler Fidelity Issues. 379–396. https://www.usenix.org/conference/usenixsecurity24/presentation/dramko

[9] DWARF Debugging Information Format Committee. 2017. *DWARF Debugging Information Format, Version 5.* Technical Report. DWARF Debugging Information Format Committee. https://dwarfstd.org/doc/DWARF5.pdf

[10] Hamed Fawareh, Yazan Al-Smadi, Mohammad Hassan, Faid AlNoor Fawareh, and Ali Elrashidi. 2024. Software Code De-Compilation Techniques and Approaches: A Comparative Study. In *2024 25th International Arab Conference on Information Technology (ACIT)*. 1–4. https://doi.org/10.1109/ACIT62805.2024.10877024

[11] GrammaTech. 2021. GrammaTech/retypd. https://github.com/GrammaTech/retypd

[12] GrammaTech. 2025. GrammaTech/retypd-ghidra-plugin. https://github.com/GrammaTech/retypd-ghidra-plugin

[13] Hex-Rays. 2025. IDA Pro. https://hex-rays.com/ida-pro

[14] Jace Kline and Prasad Kulkarni. 2023. A Framework for Assessing Decompiler Inference Accuracy of Source-Level Program Constructs:. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy.* SCITEPRESS - Science and Technology Publications, Lisbon, Portugal, 228–239. https://doi.org/10.5220/0011872600003405

[15] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011.* The Internet Society. https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs

[16] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium (CERIAS '10)*. CERIAS - Purdue University, West Lafayette, IN, 1.

[17] Zhibo Liu and Shuai Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 475–487. https:

//doi.org/10.1145/3395363.3397370

[18] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. ACM, Nagasaki Japan, 602–615. https://doi.org/10.1145/3488932.3497764

[19] National Security Agency. 2025. Ghidra. https://github.com/NationalSecurityAgency/ghidra

[20] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Santa Barbara, CA, USA, 27–41. https://doi.org/10.1145/2908080.2908119

[21] Thomas Rupprecht, Xi Chen, David H. White, Jan H. Boockmann, Gerald Luttgen, and Herbert Bos. 2017. DSIbin: Identifying dynamic data structures in C/C++ binaries. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 331–341. https://doi.org/10.1109/ASE.2017.8115646

[22] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[23] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: a Dynamic Excavator for Reverse Engineering Data Structures. (2011).

[24] Ian Smith. 2024. BinSub: The Simple Essence of Polymorphic Type Inference for Machine Code. https://doi.org/10.48550/arXiv.2409.01841

[25] Zirui Song, YuTong Zhou, Shuaike Dong, Ke Zhang, and Kehuan Zhang. 2024. TypeFSL: Type Prediction from Binaries via Inter-procedural Data-flow Analysis and Few-shot Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1269–1281. https://doi.org/10.1145/3691620.3695502

[26] Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. 2025. Decompile-Bench: Million-Scale Binary-Source Function Pairs for Real-World Binary Decompilation. https://doi.org/10.48550/arXiv.2505.12668

[27] Vector 35. 2025. Binary Ninja. https://binary.ninja/

[28] David H. White, Thomas Rupprecht, and Gerald Lüttgen. 2016. DSI: an evidence-based approach to identify dynamic data structures in C programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2931037.2931071

[29] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. ACM, Salt Lake City, UT, USA, 4554–4568. https://doi.org/10.1145/3658644.3670340

[30] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 813–832. https://doi.org/10.1109/SP40001.2021.00051

[31] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. 2024. TYGR: Type Inference on Stripped Binaries using Graph Neural Networks. 4283–4300.