Type Inference in Stripped Binaries via NLP-Guided Binary Analysis

by

Raisul Arefin Nahid

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 8, 2026

Keywords: Binary Program Analysis, Natural Language Processing, Reverse Engineering, Binary Type Inference

Approved by

Samuel Mulder, Chair, Associate Professor of Computer Science and Software Engineering
Cheryl Seals, Charles W. Barkley Professor of Computer Science and Software Engineering
Farah Kandah, Associate Professor of Computer Science and Software Engineering
Sathyanarayanan Aakur, Assistant Professor of Computer Science and Software Engineering

Abstract

Binary program analysis poses fundamental challenges due to the loss of high-level abstractions during the compilation process. In contrast to source code, binary executables lack explicit representations of variables, types, control structures, and other abstractions essential for program comprehension. The primary objective of this dissertation is to advance the state of type inference in binaries, a core problem in binary reverse engineering.

Accurately reconstructing data type information enables more reliable recovery of source-level representations. At the heart of our approach is the use of Natural Language Processing (NLP) models for type inference. We formulate type prediction as a sequence classification task over instruction-level representations. However, standard NLP architectures and tokenization strategies are not well-suited to the properties of assembly code. We therefore develop and evaluate specialized tokenization methods for assembly, designed to preserve semantic granularity and structural regularity in the input sequences. These methods yield measurable improvements in downstream model performance on type prediction tasks.

A further question concerns which class of models is best suited to type inference itself. Type inference can be performed using a smaller transformer trained specifically for the task or a large, general-purpose large language model (LLM) applied without task-specific supervision. To resolve this trade-off, we conduct a systematic evaluation that compares both paradigms across a spectrum of binary analysis tasks centered on type inference. This study clarifies when model scale can substitute for task-specific supervision and provides concrete guidance for selecting the appropriate modeling approach.

As the primary direction for future work, we outline an NLP-driven approach to reconstructing high-level type abstractions from C++ binaries. C++ binaries are especially difficult to analyze because features such as classes, templates, and multiple inheritance leave only fragmentary traces in raw binary form. We aim to map low-level assembly instructions to high-level abstractions, including Standard Template Library types (e.g., map, list) and their corresponding class and structure definitions. Recovering these object-oriented abstractions would improve decompilation, vulnerability detection, and the analysis of complex C++ programs.

Together, these components—specialized tokenization, transformer-based type inference, and a systematic evaluation of model paradigms—constitute a unified NLP-driven framework for binary program analysis, with improved disassembly reliability identified as a direction for future work.

AI Use Disclosure Statement

The author acknowledges the use of the generative AI tool Claude Opus 4.8 during the preparation of this dissertation. It was used for grammar and language editing, code assistance, and formatting. It was not used to generate any content. Anything processed by the AI was reviewed, verified, and edited by the author, who takes full responsibility for the final content of this dissertation.

Digital Accessibility Disclosure Statement

The author has made a good-faith effort to ensure that this dissertation is digitally accessible in accordance with Auburn University guidelines. Measures taken include the use of an accessible sans-serif font, a logical heading structure, descriptive alternative text for figures, accessible page numbering, and a document reading order compatible with assistive technologies such as screen readers. Despite these efforts, some elements—such as complex figures, tables, or mathematical expressions—may not be fully accessible to all assistive technologies. Readers who encounter accessibility barriers and require an alternative format are encouraged to contact the author.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Samuel Mulder, for his exceptional mentorship. He has been a great mentor and a wonderful person who made my graduate life at Auburn University smooth and worthwhile. He was not only my research advisor but also someone who helped me navigate life beyond academia. I am sincerely grateful for his guidance, patience, and support throughout this journey.

I would also like to thank my committee members for their unwavering support throughout this process. Their insightful comments and thoughtful guidance have made my research stronger and more valuable. I am also grateful to my university reader for taking the time to review this work and for providing valuable feedback that helped improve its quality.

I am thankful to all my colleagues in the PUN Lab—especially Kevan Baker, Dylan Stancil, Ahmed Mostafa, Enoch Yang, and C.T. Chidambaram—for their help and friendship, which made traveling this difficult path much easier.

I extend my gratitude to the Department of Computer Science and Software Engineering at Auburn University for its immense support of international students. I would also like to thank Dr. Xiao Qin for his guidance throughout my time in the department. His advice on navigating the PhD journey and his mentorship during my academic job search have helped me immensely.

Finally, I would like to thank my friends and family. My mother, Kohinoor Akter, and my father, Mosleh Uddin, have been a constant source of support and comfort throughout my life—not only during my PhD. My children, Aisha and Muhammad, have been the

motivation behind all my efforts. Last but not least, my wife, Yasmin Afsana, has sacrificed the most for this journey—perhaps even more than I have.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABI Application Binary Interface

ACM Association for Computing Machinery

API Application Programming Interface

AST Abstract Syntax Tree

BART Bidirectional and Auto-Regressive Transformers

BCS Binary Code Similarity

BERT Bidirectional Encoder Representations from Transformers

BPE Byte Pair Encoding

CFG Control Flow Graph

CNN Convolutional Neural Network

CPU Central Processing Unit

CRF Conditional Random Field

DNN Deep Neural Network

DWARF Debugging With Attributed Record Formats

ELF Executable and Linkable Format

GNN Graph Neural Network

GPT Generative Pre-trained Transformer

GPU Graphics Processing Unit

IDA Interactive Disassembler

IEEE Institute of Electrical and Electronics Engineers

IR Intermediate Representation

ISA Instruction Set Architecture

LLM Large Language Model

LSTM Long Short-Term Memory

ML Machine Learning

MLM Masked Language Modeling

NLP Natural Language Processing

NLM Natural Language Model

OOV Out-of-Vocabulary

OSS Open-Source Software

PDB Program Database

PLM Pre-trained Language Model

RE Reverse Engineering

RNN Recurrent Neural Network

ROUGE Recall-Oriented Understudy for Gisting Evaluation

SSA Static Single Assignment

STL Standard Template Library

SVM Support Vector Machine

VSA Value Set Analysis

Chapter 1

Introduction

1.1 Motivation

Binary program analysis is a foundational task in domains such as reverse engineering, vulnerability discovery, malware analysis, and legacy system maintenance. It enables analysts to recover semantic information from compiled executables in the absence of source code, often under conditions where symbols, debug metadata, and other high-level cues have been stripped. Executable binaries are the output of a multi-stage transformation pipeline that begins with program design and ends with machine-level code generation. While the design phase captures problem-domain semantics for human reasoning, source code serves as an intermediate representation that bridges human intent and machine execution. Compilation, however, aggressively eliminates high-level abstractions—including variable names, data types, function boundaries, and control structures—in favor of low-level, architecture-specific instructions suitable for execution on physical hardware. The resulting binary code lacks clear meaning and structure, making it very difficult to analyze. In this context, robust type inference is critical. It serves as a cornerstone for reconstructing source-level abstractions, enabling more accurate decompilation, and better program understanding.

Achieving precise type inference is dependent on the correctness of earlier stages in the binary analysis pipeline. Disassembly, the process of translating binary data into assembly instructions, is error-prone due to instruction misalignment, interleaved code and data, and the absence of control-flow metadata. Disassembly errors can propagate

into later analyses, reducing confidence in any downstream binary analysis. Improving disassembly reliability therefore remains an important direction, which we discuss as future work. More immediately, NLP models are tailored for natural language, not assembly code. So we adapted the tokenization step for assembly code which improves the overall performance of binary analysis using NLP models.

A further question concerns the kind of model best suited to type inference itself. Type inference can be approached either with a smaller transformer trained specifically for the task or with a general-purpose large language model (LLM) used off the shelf. These two options differ sharply in cost, effort, and accuracy, and it is not obvious which is preferable. We therefore study this trade-off directly, using type inference as the central task, to determine whether investing in a task-trained model is justified or whether a general-purpose LLM is sufficient. This evaluation informs the modeling choices made throughout the rest of the dissertation.

To address these challenges, this dissertation proposes a pipeline centered on type inference, but tightly integrated with supporting components: domain-specific tokenization techniques and a systematic evaluation of which class of models is best suited to the task. Together, these components enable more accurate and scalable recovery of type information from stripped binaries, advancing the state of binary analysis.

1.2 Problem Statement

The translation from source code to binary code is a lossy process, making it generally impossible to reconstruct the original source information, especially with stripped binaries that lack debug information and symbols. Therefore, there is a need for sophisticated and automated tools to overcome these challenges and enable binary-level semantic analysis. Most binary analysis tools use different kinds of heuristics to complete this tedious task.

The heuristic-based approaches often struggle with scalability and adaptability to different architectures and compiler optimizations. For analyzing binaries with newer characteristics, new heuristics need to be developed. Additionally, maintaining and updating heuristic-based tools requires significant manual effort, making them less effective against rapidly evolving software. These limitations lead to inaccuracies in fundamental binary analysis tasks such as disassembly, type recovery, and function identification, which, in turn, affect downstream analysis.

Learning-based methods address many of these limitations, but they introduce a modeling decision of their own. Type inference can be performed with a smaller transformer trained on labeled binary data or with a large, general-purpose LLM applied without task-specific training. Task-specific models are efficient and accurate but expensive to train, whereas general-purpose LLMs are easy to use but costly at inference and not always reliable on fine-grained tasks. Resolving this trade-off is itself part of the problem, since it determines how type inference and related tasks should be built.

In summary, the core problem this dissertation addresses is how to perform accurate type inference on stripped binary code using machine learning techniques. To solve this, it is necessary to:

- Adapt the tokenization process for assembly code to enhance NLP performance.
- Determine whether type inference and related binary analysis tasks are better served by task-trained smaller transformers or by general-purpose LLMs.
- Integrate these components into a unified pipeline that enables robust and scalable type abstractions recovery.

Improving the reliability of disassembly is also an important factor in this pipeline, since disassembly errors propagate into type recovery; we treat it as a direction for future work rather than a completed component.

1.3 Research Objectives

1.3.1 Improve NLP Performance on Assembly Language:

A further objective involves optimizing NLP models specifically for assembly language to enhance their effectiveness and applicability across various binary analysis tasks.

1.3.2 Improving large function analysis performance:

NLP models have a limitation of the input sequence length they can process. We have used program-slicing methods to slice a large function and feed the model the relevant portion specific to the task.

1.3.3 Producing a Large Binary Program Dataset:

The objective is to construct a large-scale dataset by collecting a diverse set of open-source software projects and compiling executables under various configurations. This dataset will encompass a wide range of compiler settings, optimization levels, and architectures, making it suitable for training and evaluating models across multiple binary analysis tasks.

1.3.4 Selecting the Right Model Paradigm for Type Inference:

An objective of this work is to determine, through systematic evaluation, whether type inference and related binary analysis tasks are best served by smaller task-trained transformers or by general-purpose LLMs. By comparing both paradigms across a spectrum of tasks centered on type inference, we identify where model scale helps and where task-specific supervision remains necessary, providing concrete guidance for how the type inference pipeline should be constructed.

1.3.5 Enhanced Binary-Type Prediction:

The primary goal of this work is to enhance type prediction in binaries using NLP models. By leveraging tailored machine learning techniques, we aim to recover variable types and data structures from low-level assembly, improving program understanding and supporting downstream binary analysis tasks, including decompilation, binary rewriting, and vulnerability detection.

1.4 Contributions of this Work

The key contributions of this work are as follows:

- We collected over 100K public C/C++ repositories from GitHub and compiled them into a diverse and extensive binary dataset, consisting of more than 500K binaries with each configuration. This dataset serves as a valuable resource for machine learning and establishing ground truth in data-driven binary analysis experiments.
- We conducted a comprehensive study demonstrating that tokenization and preprocessing of assembly language can significantly enhance the performance of transformers and large language models (LLMs) in binary analysis tasks.
- We performed an in-depth survey on the binary-type prediction task, providing a thorough examination of existing approaches and their limitations.
- We carried out an extensive set of experiments to predict binary types in binaries compiled from C programs using a transformer-based model. Our model outperformed state-of-the-art binary type prediction tools, including StateFormer [1] and Ghidra [2], achieving significantly improved accuracy and effectiveness.
- We conducted a systematic evaluation comparing general-purpose LLMs with smaller task-trained transformers across a spectrum of binary analysis tasks centered on

type inference. This study clarifies when model scale can substitute for task-specific supervision and provides practical guidance for selecting the appropriate model paradigm for type inference and related tasks.

1.5 Ongoing and Future Work

We are currently pursuing the following directions, including work that is in progress and work planned for the future:

- Our main ongoing work focuses on recovering high-level type abstractions from C++ binaries using NLP techniques. C++ binaries are particularly challenging due to complex features like classes, templates, and multiple inheritance, which are difficult to interpret in their raw binary form. Using NLP models, we aim to map low-level assembly instructions to high-level type abstractions, such as C++ Standard Template Library (STL) types (e.g., map, list) and possibly their corresponding class/structure definitions. This work seeks to improve the understanding of complex C++ programs, enabling better decompilation, vulnerability detection, and analysis of program behavior.

- As a more preliminary future direction, we also plan to improve binary disassembly by building on superset and probabilistic disassembly with additional machine-learning and statistical heuristics, aiming to reduce false positives and strengthen the downstream type inference pipeline.

Chapter 2

Background and Related Work

This chapter is dedicated to our discovery process to different binary analysis tools specifically in the area of type information recovery from stripped binaries.

2.1 Introduction

Software development is hard. A major goal of software engineering is to make the development process easier, correct, manageable, and efficient. To achieve this, most software development techniques rely on abstraction and the divide-and-conquer paradigm. These are fundamentally integrated into most modern development workflows. While these make programming more manageable, the resulting extra layers of abstraction often make a program more difficult to reverse engineer.

At the machine level, there are no variables, data structures, or types—only raw 32- or 64-bit values stored in physical locations such as registers or memory. High-level constructs such as integers, structs, or objects are abstractions introduced by programming languages. During compilation, source-level variables—with names, scopes, and defined types—are mapped to physical storage without retaining those abstractions. Reverse engineers must recover this lost information to understand program behavior. For example, identifying two values as integers gives more context than knowing they are just variables; recognizing them as a pair suggests a higher-level structure like a Cartesian point. Inferring higher abstractions yields deeper insight but requires more complex analysis.

Local variables may be stack offsets, or they may be optimized into CPU registers which are faster but few in number. When the CPU has enough registers, the compiler might optimize the program to store all the local variables in them. In other cases, they spill onto the stack. Objects and data structures dynamically allocated during program execution are typically saved into the heap section of the memory. This is because they might not fit into the limited stack memory or they need to persist even after the called function exits, which results in the termination of the function's stack memory. Sometimes the compiler will evaluate a variable as a constant and it will get no real physical storage at all. Furthermore, some variables may be unused and may not be present in the executable. The mapping of variables to physical storage is complex and depends on various decisions made by the compiler.

A lot of work has been done on binary-type information prediction. While general-purpose reverse engineering tools often include type inference features, there are also dedicated tools developed specifically for type prediction. In this paper, we survey 50 tools that have type prediction as a major feature. These use a variety of methods and vary in the types they produce. Major methods used by the tools are dynamic analysis, formal methods, statistical analysis, and machine learning. One insight is that recent tools prefer static analysis and machine learning over dynamic analysis. Some tools, particularly the proprietary reverse engineering tools such as IDAPro [3], and Binary Ninja [4] do not describe their type inference method in detail. To understand their approaches, we relied on secondary sources such as blog posts and books.

We have categorized and compared the methodologies used in detail in this paper. We also examine how different tools operate at varying levels of data abstraction: for instance, some tools infer only low-level primitive types, while others attempt to reconstruct high-level constructs such as C++ class definitions. On the other hand, tools also vary in the representation of the program they work on. Few tools work directly on assembly code, some on raw bytes, and more sophisticated tools on intermediate representations.

Our intent is to convey this information categorically and draw a clear picture of the advancements and challenges in this field.

We also identified several practical and technical limitations in the current state of binary type inference. Although many tools have been developed, there is no standard dataset or evaluation framework that the community consistently uses. This lack of standardization makes it difficult to compare tools fairly. Tools also differ in the type systems they adopt—some focus only on primitive types, others include user-defined types like structs or classes. Even among those that handle primitive types, there is no agreement on how many or which types to predict, with some tools using 20 categories and others using 40 or more. This inconsistency complicates any direct comparison of performance. In addition, many tools do not release source code, and even when they do, they often lack documentation or usable interfaces. These factors limit the adoption of academic tools in real-world reverse engineering workflows. Most general-purpose RE platforms continue to rely on their built-in type inference systems and do not incorporate the advances made by research tools.

This survey is guided by the following research questions:

- **RQ1: How has the landscape of binary type inference tools evolved over the past decade?** We examine 50 tools, tracing the evolution from early dynamic analysis approaches to modern static, learning-based, and hybrid methods.
- **RQ2: What methodologies do existing tools use to recover type information from stripped binaries?** We categorize tools by their underlying techniques and analyze the trade-offs of each approach.
- **RQ3: What types of abstractions can existing tools recover?** We examine the scope of type recovery, ranging from primitive types to higher-level abstractions such as data structures, class hierarchies, and function signatures.

- **RQ4: How do existing tools represent and process binary programs as input?** We analyze input representations and context extraction strategies across tools.
- **RQ5: How are binary type inference tools evaluated, and how reproducible are their results?** We assess evaluation methodologies, benchmarks, metrics, and the state of reproducibility and open-source availability.
- **RQ6: What are the open challenges and future directions in binary type inference?** We identify key limitations and outline promising directions for future research.

2.2 Background

The objective of type inference is to use the information present in the binary file to recover the original data types specified by the developer as closely as possible. This includes recovering basic types (int, float, pointers, etc), function signatures, and recognizing or reconstructing other higher-level abstractions such as classes or objects. During compilation from source-code, type information can be stored as symbols or debug information. The challenge in type inference is that most distributed executables are stripped of symbols and other information that provide clues to the correct types.

The process of recovering types from executables involves several distinct tasks. Initially, the executable must be parsed to identify its various components, followed by the disassembly of its instruction sections. These initial tasks are beyond the scope of our survey, even though they are integral subproblems of type recovery. Instead, our focus is on the later stages of type recovery, specifically variable discovery and typing in assembly, as well as retyping in the decompiled source.

2.2.1 Motivation for a New Survey

Nearly a decade has passed since the publication of the 2016 survey by Caballero and Lin [5], which systematized 38 binary type inference works up to that point. In that time, the field has undergone significant transformation, both in the volume of new contributions and in the nature of the methods being applied. This survey is motivated by the need to capture and systematize these developments in a coherent and updated framework.

Expanded Tool Coverage. Our survey covers 50 tools, of which 37 were not covered in the previous survey. Although the 2016 survey examined 38 tools, we intentionally excluded many of them from our scope, as they did not substantially engage with the type inference problem itself and were more focused on adjacent tasks such as disassembly, protocol reverse engineering, or vulnerability detection. This selective inclusion allows us to maintain a tighter focus on tools where type inference is either the central objective or constitutes a substantial part of the methodology.

New Methodological Landscape. The 2016 survey was largely organized around static and dynamic analysis, which represented the dominant paradigms at the time. Since then, the field has seen the emergence of entirely new methodological categories that were nonexistent or nascent in 2016. These include Graph Neural Networks (GNNs), Transformer and NLP-based approaches, Large Language Model (LLM)-inspired methods, and hybrid systems that combine formal reasoning with machine learning. In particular, hybrid methods — such as TypeSqueezer [6], HyRES [7], and TypeForge [8], which integrate static analysis, dynamic analysis, and LLMs within unified pipelines — represent a category that is entirely absent from the 2016 survey and reflects a fundamentally different approach to type recovery.

New Taxonomy. The binary nature of the 2016 taxonomy — static versus dynamic — is no longer sufficient to describe the diversity of modern approaches. This survey introduces a richer, multi-level taxonomy that organizes tools into four major categories: Dynamic Analysis, Static Analysis (further subdivided into Value Set Analysis, Symbolic Execution, Constraint Solving, Logic-Based Reasoning, and Statistical Methods), Learning-Based Methods (subdivided into Classical ML, CNN, NLP/Transformer Models, and GNNs), and Hybrid Approaches. This taxonomy better reflects the current state of the field and provides a more principled basis for comparing tools across methodological boundaries.

New Cross-Cutting Insights. This survey identifies several major trends that were not visible in 2016, including a shift from dynamic to static methods, a transition from handcrafted rules to ML-based approaches, the emergence of GNNs and Transformers, increasing use of hybrid pipelines, and a growing dependence on decompilers and intermediate representations. We also cover significant post-2016 advances in constraint-based type inference — including Retypd [9], BinSub [10], TRex [11], and Manta [12] — which are absent from the prior survey.

Reproducibility, Benchmarks, and Practical Usability. The 2016 survey noted a lack of standardized evaluation metrics but did not examine reproducibility as a systematic concern. Our survey surfaces this as a critical limitation, observing inconsistent type taxonomies, incompatible evaluation metrics, limited source code availability, and poor reproducibility across many tools. Beyond cataloguing methods, we also evaluate real-world usability, tool accessibility, and integration into reverse engineering workflows, offering a practitioner-focused perspective that is absent from the prior work.

2.2.2 Scope

Type inference for executable binaries has been studied for more than two decades and spans a diverse set of methodologies, objectives, and recovered abstractions. To maintain a focused and manageable scope, this survey includes tools in which type inference

is either the primary objective or a substantial component of the overall methodology. We consider approaches that recover primitive types, pointers, function signatures, data structures, object-oriented abstractions, and other high-level program constructs from compiled binaries.

We intentionally exclude works that focus primarily on adjacent reverse-engineering tasks, such as disassembly, malware analysis, binary similarity, or vulnerability discovery, where type inference plays only a minor role. Our survey covers both academic research systems and widely used industrial reverse-engineering tools. Applying the selection criteria described later in this section, we reviewed a total of 50 tools.

Literature Sources We searched major academic databases and publication venues, including Google Scholar, ACM Digital Library, IEEE Xplore, USENIX Security Symposium proceedings, and NDSS Symposium proceedings. In addition, we reviewed the papers included in the previous survey by Caballero and Lin [5] and recursively explored both backward and forward citation networks to identify additional relevant works.

The following search strings were used:

- `"binary type inference"`
- `"type recovery" AND "binary"`
- `"type recovery" AND "executable"`
- `"type inference" AND "executable"`
- `"stripped binary" AND "type"`
- `"reverse engineering" AND "type inference"`
- `"data structure recovery" AND "binary"`
- `"variable type recovery" AND "binary"`

The literature search and tool collection process were conducted between August 2024 and March 2026. The search covered publications from **1999 to 2026**, beginning with the seminal work of Mycroft [13], one of the earliest studies on type inference from executable binaries. Newly published works that appeared during the survey process, including recent studies from 2026, were also considered for inclusion when they satisfied the survey criteria.

Because many reverse engineering tools are distributed as software projects rather than described in dedicated research publications, we also consulted researchers and practitioners in the reverse engineering community to identify widely used tools that may not be well represented in the academic literature. For commercial and closed-source tools whose type inference methodologies were not fully described in peer-reviewed publications, we relied on official documentation, project websites, technical manuals, books, and developer blog posts to characterize their capabilities and reported functionality.

Selection Criteria To maintain a focused scope, we established explicit inclusion and exclusion criteria for selecting tools and research works.

Inclusion Criteria. A tool or work was included in this survey if it satisfied all of the following criteria:

- Type inference was either the *primary objective* or constituted a *substantial component* of the methodology.
- The work targeted *compiled binary executables* or *binary-level intermediate representations*.
- The work was published in a *peer-reviewed venue* or represented a widely adopted tool with sufficient publicly available technical documentation.

- The work proposed, implemented, or evaluated a *concrete type recovery technique* rather than merely discussing type inference as a motivation or downstream application.

Exclusion Criteria. A tool or work was excluded if any of the following conditions applied:

- The work focused primarily on adjacent reverse-engineering tasks such as disassembly, function boundary detection, protocol reverse engineering, malware analysis, or binary similarity, where type inference was only a minor component.

Screening Process The initial search identified 96 candidate research papers. After removing duplicates and entries with insufficient publicly available information, 87 works remained. Title, abstract, and documentation screening excluded 21 entries that did not substantially address binary type inference. The remaining 66 works were reviewed in detail using the selection criteria described above. Following full review, 22 works were excluded, resulting in 44 tools identified from peer-reviewed publications. To improve coverage of widely used industrial and reverse-engineering tools, we additionally included six tools identified through expert consultation and publicly available technical documentation. The final survey includes 50 tools spanning academic research prototypes, open-source software projects, and commercial reverse-engineering platforms. Table 2.1 summarizes the screening process.

2.2.3 Application

Binary type inference is one of the fundamental tasks of binary program understanding. It is crucial to identify the types of data the instructions manipulate to understand the semantics of the program. One of the most important applications of type inference is to assist [2, 3, 14] or improve [15–18] the decompilation of binaries. Decompiled code is

Table 2.1: Summary of the tool selection and screening process.

Screening Stage	Count
Initial candidates identified	96
After removing duplicates and insufficient entries	87
After title, abstract, and documentation screening	66
After full-text review against selection criteria	44
Additionally included via expert consultation	+6
Final tools included in survey	**50**

an important way to understand the functionality of a binary and quality type information can improve the readability of the decompiled code. Type information can also assist in malware and software vulnerability detection. Executables containing function signatures, and data structures known to be used by known malware can be more accurately identified [19]. Type information is also essential for binary rewriting [20], useful for vulnerability prevention, and binary code reuse. Type information is needed for determining the signatures of functions, and variables related to inputs and outputs so new code can be interfaced with existing code. Types have also proven to be useful for improving Control-Flow Integrity [20]. For example, TypeSqueezer [6] utilized type information for improving their Control-Flow Integrity algorithm to prevent forward-edge attacks [21].

2.2.4 Debug Information: Ground Truth for Type Inference

Debug information is supplementary data that is included in a compiled executable to assist in understanding the behavior of the program during execution. As the name suggests, the intention behind debugging information is debugging a program, i.e., setting a breakpoint at the source code level and examining the variable values when the program stops there. To achieve this, a lot of crucial information about the original source is preserved through the compilation process. This information helps to reverse the compiler's transformations, converting the program's data and state back into terms that the programmer originally used in the program's source [22]. Source information such as function names,

function parameters with types, variable names with types and locations, and information about Class and Structure can be recovered from debug information. To protect intellectual property, most closed-source programs are delivered without debugging information. Two popular debug formats are DWARF[1], used in Linux, and PDB[2], used in Windows programs. Debug information is crucial for creating the ground truth for type prediction task.

2.2.5 Intermediate Representations(IR): Architecture-Independent Analysis for Binaries

Originally, Intermediate Representations were representations of the source code used by compilers to enable code optimization. Compilers convert source code into an IR and then optimize for performance. In this way, compilers can use the same transformations on source code from different languages. In the context of reverse engineering, IRs serve a similar purpose. Executables from different architectures are lifted to IRs and RE tools can apply the same analysis regardless of the architecture. However, like any other reverse engineering task, lifting a binary to an intermediate representation is difficult and error-prone. A few examples of popular IRs are P-Code used by Ghidra [2], VEX used by Angr [23], and Microcode used by IDA Pro [3].

2.2.6 Binary Variability and Its Implications for Type Inference

Binary executables vary in several aspects that significantly impact binary analysis. A tool that performs well on one variant may completely fail on another.. While Intermediate Representations can help mitigate some of these challenges by abstracting away details, a considerable number of tools still operate directly on disassembled machine code. This approach has both advantages, such as fidelity to the original binary, and drawbacks, such

[1] dwarfstd.org
[2] llvm.org/docs/PDB/index.html

as reduced portability and flexibility. Below, we discuss some of the most critical variations in binaries that can influence the effectiveness of analysis:

Optimization Levels (e.g., -O0, -O1, -O2, -O3) Optimization levels control how compilers modify code for performance or size. For example, higher optimization levels (-O2, -O3) often result in restructured loops, inlined functions, combining or removing variables entirely, or the removal of unused/unneeded code. While these changes improve runtime efficiency, they may also increase complexity and remove or change abstractions made in the original source, making the recovery of the original program structure and type information significantly harder.

Target Architecture (e.g., x86, ARM, RISC-V) Each architecture defines its own instruction set architecture, endianness, calling conventions, and other properties. The complexity of binary analysis is typically not due to the architecture itself but rather to the level of toolchain coverage, lack of public documentation, and uncertainty related to the architecture's design principles. Architectures that are less widely adopted may be harder to analyze primarily because reverse engineering tools are less frequently tested or optimized for them. Conversely, mainstream architectures like x86, despite broad tool support, can be challenging because of characteristics such as variable-length instructions.

Compiler (e.g., GCC, Clang, MSVC) Compilers use different idioms to translate source code into machine code, often producing distinct assembly patterns, optimization artifacts, or debug metadata. Understanding the behavior of specific compilers can provide valuable insights during reverse engineering and type recovery. However, this variability also introduces an additional layer of uncertainty, as the same source code can result in different instructions depending on the compiler.

2.3 A Taxonomy of Type Inference Methods

Binary type inference tools use a wide range of methodologies to recover variable type abstractions. These methodologies can be broadly categorized into **Dynamic Analysis** and **Static Analysis**, based on whether the binary is executed during analysis. Additionally, hybrid techniques and learning-based methods often blend elements from both categories.

2.3.1 Dynamic Analysis

Dynamic analysis techniques observe the binary during execution to infer type information. These methods benefit from precise runtime values and control-flow but suffer from limited code coverage and susceptibility to evasion by obfuscated or malicious code. Dynamic techniques can be further divided into the following subcategories:

- **Value-Based Analysis**: Infers types from runtime values in memory/registers (e.g., *Laika* [19]).
- **Flow-Based Analysis**: Tracks data-flow during execution to propagate type information (e.g., *Rewards [24]*, *DynCompB [25]*).
- **Memory Access Pattern Analysis**: Uses patterns in dereferencing and memory layout to infer structures (e.g., *Howard* [26], *DSIBin [27]*).
- **Memory Graph Analysis**: Constructs dynamic graphs from memory allocations and links to recover structural types (e.g., *DDT* [28], *MemPick* [29], *ARTISTE* [30]).

2.3.2 Static Analysis

Static methods analyze the binary without execution. They operate on disassembled code, intermediate representations, or abstract models. These techniques scale better and can

analyze full binaries but may suffer from ambiguity and require complex reasoning. Subcategories include:

- **Value Set Analysis (VSA)**: Over-approximates the set of possible values for abstract locations (e.g., *Ghidra* [2], *Binary Ninja* [4], *angr* [23], *Retypd* [9]).
- **Symbolic Execution**: Simulates program paths with symbolic inputs to infer constraints and behavior (e.g., *angr* [23], *OBJDIGGER* [31]).
- **Forward Reasoning / Logic Inference**: Applies inference rules to extracted facts using logical reasoning (e.g., *OOAnalyzer* [32]).
- **Constraint Solving**: Extracts type constraints from code and solves them using SMT or custom solvers (e.g., *TIE* [33], *Retypd* [9], *TRex* [11], *BinSub* [10], *Manta* [12]).
- **Statistical Models**: Uses probabilistic reasoning to infer types from heuristics and data patterns (e.g., *OSPREY* [34], *Stride* [16]).
- **Type Propagation**: Propagates known type information through call graphs or data flows (e.g., *NTFUZZ* [35]).

2.3.3 Learning-Based Methods

Learning-based approaches apply machine learning models to infer types from patterns in code, control-flow, data-flow, or embeddings. These methods often generalize well and can capture statistical relationships missed by traditional analysis, but they require large training datasets and may struggle with out-of-distribution binaries. Subcategories include:

- **Classical Machine Learning Models**: Use engineered features and classifiers such as random forests or SVMs (e.g., *TypeMiner* [36], *DEBIN* [15], *BITY* [37], *Escalada et al.* [18]).

- **Convolutional Neural Networks (CNNs)**: Treat binary representations or instruction traces as spatial sequences (e.g., *Cati* [38]).
- **Natural Language Processing (NLP) Models**: Leverage tokenization, embeddings, and sequence models to learn semantic and structural cues from instructions (e.g., *EKLAVYA* [39], *DIRTY* [17], *StateFormer* [1], *SnowWhite* [40], *ReSym* [41], *IDIOMS* [42]).
- **Graph Neural Networks (GNNs)**: Apply message passing over control-flow graphs, data-flow graphs, or interprocedural graphs to infer types from structural relations (e.g., *TIARA* [43], *TYGR* [44], *DRAGON* [45], *ByteTR* [46]).

2.3.4 Hybrid Methods

Hybrid tools are systems in which multiple inference techniques play comparably central roles in the overall analysis pipeline, rather than one technique merely augmenting another. Hybrid tools integrate multiple major inference techniques whose complementary contributions are individually indispensable to the analysis, distinguishing them from systems where secondary methods only augment a primary approach.

Representative hybrid systems include TypeSqueezer [57], HyRES [81], and TypeForge [96].

2.3.5 Dynamic Analysis Techniques for Binary Type Inference

Dynamic program analysis refers to analyzing a binary's behavior while it executes, either on native hardware or in an instrumented emulated environment. Unlike static analysis, which must reason about all possible execution paths, dynamic analysis observes concrete runtime behavior, allowing it to resolve uncertainties about memory addresses, operand values, calling contexts, and control-flow targets. This makes dynamic approaches appealing for type inference, where accurate observation of how memory locations are

read, written, and passed across function boundaries can directly reveal type-related semantics. However, dynamic analysis has several structural limitations in the context of binary type recovery. The most significant is coverage: only instructions that execute under available inputs are analyzed, leaving potentially large regions of the program untyped. Achieving sufficient behavioral coverage requires diverse inputs, constructing test cases, and configuring the execution environment in a way that mirrors the program's real-world usage. Software-testing ideas apply here as well: choosing good inputs, running the program in a realistic environment, and tracking which code paths are exercised. Tools often use small test drivers, auto-generated inputs, or fuzzing to increase coverage, but full coverage is almost never achievable for large or interactive binaries.

Dynamic approaches must also deal with adversarial behavior: many malware binaries detect sandboxed or instrumented environments and deliberately suppress meaningful execution, preventing critical type-related behaviors from appearing. Furthermore, runtime observations capture only a subset of the program's semantic possibilities, complicating root-cause analysis of inferred types and making dynamic analysis inherently incomplete. Despite these limitations, dynamic type inference remains valuable because many use cases require only partial typing—for example, inferring the layout of a specific data structure, identifying heap object boundaries, or recovering types for functions of forensic interest. The ability to precisely monitor memory accesses and object lifetimes at runtime allows dynamic systems to reveal complex behaviors that static methods may over-approximate or miss entirely. Historically, many of the earliest type-recovery systems relied heavily on dynamic analysis, including LEGO [47], Mempick [29], ARTISTE [30], Howard [26], TIE [33], Rewards [24], DDT [28], Dc [48], Laika [19], and DynCompB [25]. These tools can be further categorized by their primary methodologies—for example, runtime memory-access tracing, object lifetime profiling, dynamic points-to analysis, and differential behavioral comparison across executions.

Value-Based Type Inference: Inferring Types from Runtime Values

Value-Based Type Inference deduces the types of variables by examining the actual values stored in registers and memory during program execution. This approach does not require analyzing the program's code but focuses instead on the content of memory and registers. It relies on heuristics to determine if certain patterns in the data correspond to specific data types, such as pointers or strings. For example, if 4 consecutive bytes form an address in the live memory, they are likely a pointer.

Laika [19] is a dynamic analysis tool that uses memory images and probabilistic methods to predict object types. It uses dynamic analysis to generate the memory map of the program. Laika identifies potential pointers in the memory image, based on whether the contents of 4-byte words look like a valid pointer, e.g., a value that points into the heap. Then it uses the pointers to identify object positions and sizes. Laika classifies every memory block into four categories: address, zero, string, and data (everything else). Then it converts the object's bytes in the memory from raw bytes to sequences of block types. It uses a Bayesian unsupervised algorithm to detect similar objects with similar sequences of block types. It also detects similar objects and clusters them to detect lists and other abstract data types. This technique can be used to find similarities between two programs by looking at their memory image, it has been used as a malware detection tool.

Discussion: Value-based type inference leverages runtime value regularities—especially pointer structure and repeated heap templates—to recover high-level data-structure types, enabling robust detection even under code polymorphism. Its effectiveness, however, is limited by the noisy and ambiguous nature of real heap memory, which often contains padding, freed regions, and misleading values. In addition, object boundaries are difficult to determine reliably, making the overall inference more error-prone.

This method assigns types to variables based on how the program uses them during execution. The core concept is to model the discovery of data types and semantics as an information flow problem, where type attributes ”flow” between registers and memory locations based on program execution.

Rewards [24] uses type propagation in a dynamic setting. During execution, it looks for operations such as system calls, standard library calls, and other type-revealing instructions. As the parameters and return types of the system calls and standard library calls are known, those types are propagated to the variables directly passed and returned in the binary. Type-revealing instructions are instructions that reveal type information about its operands. For example, floating-point instructions such as `FADD` operate with floating-point numbers. Similarly, with a “`MOV [mem] , eax`” instruction that has an indirect memory access operand, the source has to be a pointer. Rewards also use type propagation to propagate the discovered type information to other variables that interact with the variables with known types through dataflow analysis.

DynCompB [25] infers abstract type similarities. It identifies a set of variables that serve a similar kind of purpose. For example, two integer variables which are used as weekly salary and monthly salary are of the same abstract type. DynCompB uses flow-based analysis to find interactions between variables to determine if they are of the same abstract type. For instance, in an assignment `x=y`, when the value stored in y is copied into x, the abstract type of y will be assigned as the abstract type of x.

Discussion: A key strength of flow-based type inference is its precision: because type propagation is grounded in concrete executions, these methods can often recover more accurate type relationships than purely statistical or heuristic approaches. Dynamic flow-based techniques can also infer higher-level abstract types by unifying values that interact during execution, which provides useful semantic groupings beyond primitive typing. However, their effectiveness is fundamentally constrained by runtime coverage. Since

propagation is limited to observed execution paths, unexecuted code regions and rarely triggered behaviors remain untyped, resulting in incomplete type recovery. In this respect, value-based inference can sometimes be more robust, as it relies less on long propagation chains. Flow-based methods are also sensitive to memory reuse and variable lifetime issues, requiring additional mechanisms such as timestamps or offline resolution procedures to prevent conflating distinct variables that occupy the same memory location at different points in execution.

Inference from Memory Access Patterns

Memory Access Pattern analysis infers type information by observing how memory is accessed during program execution. The way that a program dereferences addresses—such as accessing specific offsets from a base pointer—can reveal structural layouts, array indexing behavior, or variable locations in stack frames.

Howard [26] identifies data structures by looking at the pattern of memory access during runtime, which provides clues about the layout of data in memory. For example, if X is an address to a structure, a dereference of ∗(X + 4) likely accesses a field of that structure. Similarly, if X is the address of an integer array, ∗(X + 4) is its second element. If X is a function frame pointer, then ∗(X + 4) likely refers to a parameter and ∗(X - 8) is likely a local variable. Howard analyzes both the stack and heap for data structures. It tries to identify the individual fields of a structure, but members never accessed during the dynamic run cannot be recovered.

DSIBin [27] is a binary extension of the former work DSI [49] which operates on C source code. It analyzes binaries by integrating with Howard. Initially, it uses Howard to produce low-level types. Then it uses a type graph to propagate existing type information. In a type graph types are vertices and pointers are edges. It refines the type information obtained by leveraging the core algorithm of DSI.

Discussion: The effectiveness of inference from memory access patterns lies in its ability to recover data structures that are internal to the program and not a part of the standard library or system call interfaces. Unlike methods that rely on data propagation from known ”type sinks,” this approach tracks the life cycle of pointers within a private, instrumented environment, allowing it to identify structures that are otherwise invisible to external observers. Furthermore, it achieves a superior level of granularity by moving beyond simple aggregate identification; by logging specific offsets and access size, the method tries to distinguish the internal composition of a structure, such as differentiating between a 4-byte int and a single-byte char. This bottom-up reconstruction of the data layout enables complex and custom aggregates to be mapped.

Type Inference via Runtime Memory Graph Analysis

Memory Graph Analysis is a technique used in dynamic analysis to structurally represent how data is organized, stored, and interconnected within a program’s memory at runtime. In a memory graph, nodes represent allocated memory regions (like heap allocations or global variables) and edges illustrate the references or pointers between these regions.

DDT [28] dynamically monitors the organization of data in the memory and keeps track of how data is stored and loaded from memory. It records memory allocations and memory stores to create an evolving memory graph. It identifies the interface functions that interact with a data structure in the memory graph by analyzing the memory loads by the functions. It also tracks the memory state before and after a function call happens, which is used to determine the invariants of the function calls [50]. Function invariants are program properties that are unchanged throughout the execution of a function and can be used as a signature. Memory graph, interface functions, and invariants are then matched against a library of known data structures to identify the data structure under examination.

MemPick [29] also builds a dynamic memory graph from heap allocations and pointer linkages but avoids reliance on explicit interface functions. Instead, it groups similar heap

objects, isolates candidate structures, and classifies them using shape-based properties through a hand-crafted decision tree. For example, if the memory graph contains a tree that has 5 nodes, then there should be 5 similar objects in the graph. Following this scheme, MemPick detects and isolates different data structures. To identify the type of the data-structures, it uses a hand-crafted decision tree which looks at the property of the data structure and classifies that to a graph, linked list, binary-tree, n-aray tree etc.

ARTISTE [30] recovers composite data structures along with the primitive types constituting the data structure. It also performs shape analysis to detect recursive data structures. It observes instruction and function type sinks to detect primitive types. Instruction sinks are those instructions that reveal type information about their operands. For example, both operands of an `add` in 32-bit x86 code should be of type `num32`. Function type sinks are functions whose signature and return type are publicly known, such as standard library calls. ARTISTE performs analysis on the memory graph and with the recovered primitive types, tries to recover the data structures containing them. Compared to DDT's reliance on interface-function invariants and MemPick's purely shape-based classification, ARTISTE provides a more comprehensive and automated reconstruction of complex heap structures in stripped binaries. This work is unique from other dynamic analysis-based type tools in that it tries to generate both high-level data structures and also primitive constituent types.

Discussion. Runtime memory graph analysis infers structural type information by modeling points-to relationships among dynamically allocated memory regions, where nodes represent concrete regions and edges capture pointer references between them. This representation enables recovery of recursive structures such as lists and trees and can evolve across execution as regions are allocated or freed. static methods generally cannot generate these graphs, because concrete region lifetimes and multiplicities depend on runtime inputs. Collectively, memory-graph–based inference provides strong structural insight into heap organization compared to the other dynamic methods discussed.

Other Methods

LEGO [47] uses dynamic analysis to recover the class hierarchies and composition relationships of the program. LEGO recovers class structures from binaries in two phases. First, it dynamically instruments a stripped binary during execution with test inputs to monitor object allocations, lifetimes, and method invocations. This produces object-traces, where each trace records method calls and returns for a specific object, along with the methods directly invoked. In the second phase, LEGO analyzes the destructor sequence of each object-trace, which encodes inheritance depth and ancestor cleanup order. LEGO uses this information and method-call information to infer class hierarchies, identify member functions, and detect composition relationships.

2.3.6 Static Analysis Techniques for Type Inference

Static analysis methods try to understand how a program behaves without running it [51]. These methods work by looking at the code or binary and applying different techniques to make safe guesses about values, memory usage, or types. One common approach is abstract interpretation [52, 53], which builds a simpler version of the program that still keeps important information. Abstract interpretation approaches approximate the program's actual behavior using an abstract model, which helps analyze it safely [35]. Tools can find variable types, memory use patterns, or function prototypes. In the following subsections, we describe several static techniques used for type inference in binaries.

Value Set Analysis

Value-set analysis (VSA) was first introduced by Balakrishnan and Reps [54, 55] and is used for analyzing the contents of the memory. It uses non-overlapping regions in memory and a-locs, which are abstract locations which could be in memory or registers. The a-locs represent variables. VSA is a static analysis technique that uses abstract interpretation

to over-approximate the possible numeric values and memory addresses that a-locs may hold at specific program points.

VSA provides a sound approximation of memory usage and variable behavior. It is a powerful method for variable-like entity recovery. The information recovered from VSA can be utilized further to discover the aggregate structures. VSA is used by tools like Ghidra [2], Binary Ninja [4], angr [23], TIE [23], BITY [37], and Retypd [9].

Symbolic Execution-Based Methods

Symbolic execution analyzes a program by simulating its execution with symbolic inputs rather than concrete values, producing, for each explored path, a set of constraints expressed over those symbolic variables. At each guarded branch the symbolic state forks: one successor accumulates the branch condition as a constraint and the other its negation. Solving the accumulated path constraints—typically with an SMT solver [56]—reveals the input conditions under which a given program point is reachable or an assertion is violated. The principal cost of this approach is *state-space explosion*: because the state forks at every branch, the number of paths can grow exponentially, and loops with symbolic exit conditions can in principle fork without bound.

For type inference, the value of symbolic execution lies less in the technique itself than in *how surveyed tools embed it* within their analysis pipelines. Two contrasting usages are illustrative.

Symbolic execution as a primary recovery driver (OBJDIGGER). OBJDIGGER [31] uses symbolic execution as the central mechanism for recovering object-oriented structure from C++ binaries compiled with MSVC. Its analysis is organized around the `ThisPtr`, the pointer to an object's start in memory that the compiler passes as a hidden argument to member methods. By symbolically tracking how each `ThisPtr` is propagated and used—combined with static inter-procedural dataflow analysis [57]—OBJDIGGER deduces relationships between methods and object layout, identifying constructors, methods, and data

members from the memory access patterns indexed off the pointer. It further follows how virtual function tables are installed and used to recover virtual methods and their positions, and by observing how `ThisPtr`s are passed and modified across constructor calls, it infers inheritance relationships between classes. Here symbolic execution is the typing engine itself: the recovered abstractions are a direct product of symbolically reasoning about pointer propagation.

Symbolic execution as supporting infrastructure (angr). angr [23] occupies the opposite end of the spectrum, treating symbolic execution as general binary-analysis infrastructure rather than as the typing mechanism proper. angr is built on VSA and symbolic execution [58–60], and its variable recovery originally mirrored that of TIE [33]. Its current variable-recovery approach[3] takes a function, performs data-flow analysis in SSA form [61], and runs concrete execution over each statement to track every register and memory access that touches a variable.[4] Crucially, the *typing* step is handed off to a separate constraint-solving component[5] that largely implements the Retypd [9] constraint system. Symbolic execution thus underpins exploration and variable discovery, but the actual type recovery is delegated to constraint solving rather than performed symbolically.

Discussion: The OBJDIGGER–angr contrast captures the broader role of symbolic execution in binary type inference. As OBJDIGGER shows, symbolic execution can serve as a primary typing driver, and it is especially valuable because it reasons about program behavior without executing the code, exposing structure in binaries where source is unavailable and where dynamic execution might miss rarely triggered paths. Yet as angr illustrates, mature pipelines more often relegate symbolic execution to exploration and fact-gathering, layering a dedicated constraint solver on top to perform inference. This pattern reflects the technique's core trade-off: symbolic execution provides a strong

[3] angr project. 2025. Issue #1965: Variable recovery redesign. GitHub issue; accessed 2025-08-26.

[4] angr project. 2025. *Variable Recovery API (docs for v9.2.56)*. Project documentation; accessed 2025-08-26.

[5] angr project. 2025. *Typehoon type recovery implementation (typehoon.py)*. GitHub; commit 20d3dd1; accessed 2025-08-26.

semantic foundation and precise path conditions, but state-space explosion and the difficulty of mapping path constraints directly to source-level types mean that complete and accurate recovery typically requires integrating it with complementary methods such as constraint solving or dataflow analysis.

Forward Reasoning (or Forward Chaining)

Forward reasoning, also known as forward chaining, is an inference method that begins with a set of initial facts and repeatedly applies inference rules—each consisting of a set of **preconditions** and a **conclusion**—to derive new facts. Whenever a rule's preconditions are satisfied by the current fact set, its conclusion is added, and the process iterates until no further facts can be derived. In binary type inference, this lets an analysis build high-level abstractions such as classes, methods, and inheritance relationships out of small, low-level observations about a program.

OOAnalyzer [32] is the canonical example in this category, recovering C++ classes and methods from stripped executables using logic programming. It first uses a lightweight symbolic analysis to generate an initial set of low-level facts—its fact-base—and then performs forward reasoning by matching a built-in set of rules against those facts. When forward reasoning stalls before important properties are resolved, OOAnalyzer makes an educated guess about an ambiguous property, called hypothetical reasoning, and, whenever it detects an inconsistency in the fact base, backtracks and systematically revisits earlier guesses, starting with the most recent one. Consistency is enforced by a special set of rules that detect contradictions rather than assert new facts, and the final output is produced only once all facts are consistent.

Limitations of rule-based reasoning. The intuitive appeal of forward reasoning—combining simple observations into complex structural conclusions—is offset by its dependence on handcrafted rules, which is its central weakness. Because the rules encode

specific compiler idioms (for OOAnalyzer, how a given compiler emits vtables, constructors, destructors, and this-pointer conventions), they are tightly coupled to the compilation settings under which they were authored. A rule set tuned for one compiler, target architecture, or optimization level can silently fail on another: higher optimization levels inline or elide the very constructor and destructor calls that the rules key on, different compilers adopt different object-layout and dispatch idioms, and less mainstream architectures may not match the assumed instruction patterns at all. Extending coverage therefore tends to mean accumulating ever more special-case rules rather than generalizing, and a single unsound rule can corrupt the entire downstream deduction.

EmCee [62] makes this fragility concrete. It exists specifically to harden OOAnalyzer, whose many inference rules for reconstructing classes, inheritance, constructors/destructors, and vtable structure often fail in rare compiler corner cases. EmCee acts as a model of the compiler that automatically generates diverse test programs together with reliable ground truth about which OO abstractions should exist. By compiling each generated program into structured OO-related facts—spanning both executable-level observations (e.g., this-pointer propagation, vftable installations, call targets, and offsets) and source-level entity facts (e.g., class membership, inheritance relationships, and destructor types)—and systematically testing OOAnalyzer's reasoning against them, EmCee identifies unsound rules and missing scenarios so developers can patch and refine them. That a dedicated tool is needed to discover where OOAnalyzer's rules break down is itself direct evidence that rule coverage is brittle across compilers and compilation settings.

Discussion: Forward reasoning is well suited to type inference because it derives higher-level abstractions from small, low-level hints through transparent, explainable logical steps. In practice, however, the quality of its output is bounded by the strength and coverage of its rules. Since those rules are handcrafted and tailored to particular compiler behaviors and binary patterns, they typically require substantial adaptation to remain

sound across different binaries, compilers, architectures, and optimization levels—an effort that automated rule-auditing approaches such as EmCee aim to reduce but do not eliminate.

Type Inference via Constraint Generation and Solving

Constraint-based type inference determines unknown types by translating observed program behavior into logical constraints and then solving those constraints to obtain consistent type assignments. In binary analysis, these constraints are derived directly from how variables are used during execution. Representative examples include:

- Passing a value to a well-known function such as `strlen` constrains the argument to be a pointer to characters and the return value to be an unsigned integer, allowing these types to propagate to related variables through assignments.
- Using a variable in arithmetic operations (e.g., subtraction with an unsigned integer) constrains the result to be compatible with a numeric type, enabling inference through subtype relationships.
- Storing a value at a memory address and later loading from the same address implies both operations share the same underlying type, creating consistency constraints over memory.

These usage-based constraints are collected into a unified constraint system and solved to compute conservative yet precise upper and lower bounds for each variable's type. By merging information across control-flow paths and propagating subtype relations, the solver can recover meaningful high-level abstractions such as pointers, integers, and structures.

TIE [33] by Lee et al. is a principled type inference tool that integrates both static and dynamic analysis. It is built on the BAP framework [63], which lifts binaries into the

Binary Intermediate Language (BIL). BIL provides foundational low-level type information, such as whether a value originates from memory or a specific register, and the bit-width of the registers. TIE enhances this by using Value Set Analysis and memory access pattern analysis to recover high-level variables. It then generates type constraints based on variable usage—for example, if a variable is used in a signed division, a constraint is added to enforce that it must be a signed type. TIE's philosophy is to make the most informed inference possible without guessing. If the analysis concludes that a variable is an integer but cannot determine its signedness, TIE outputs a custom *number* type, representing either a signed or unsigned integer. Thus, the inferred types may be ranges or sets of possibilities, requiring the reverse engineer to finalize the interpretation.

Retypd [9], developed by Noonan et al., applies a static, constraint-based type inference approach capable of handling structure and polymorphic types [64]. It operates on an intermediate representation (IR) produced by GrammaTech's CodeSurfer [65]. Retypd produces the number and location of inputs and outputs to each procedure, as well as the program's call graph and per-procedure control-flow graphs utilizing the IR. Afterward, Retypd generates type constraints from a TSL-based abstract interpreter [66] and includes the pre-computed type information obtained from externally linked functions. The constraints are resolved using a simplification algorithm based on [67], after which the inferred type information is translated into C-like types.

BinSub [10] builds on Retypd by replacing its subtyping mechanism with algebraic subtyping [68], improving both theoretical clarity and practical efficiency. Retypd supports polymorphic inference but suffers from a worst-case cubic-time constraint solving process. BinSub addresses this by reformulating the constraints in the algebraic subtyping framework, which scales better. The pipeline remains similar: disassembly is lifted to IR, type constraints are generated and transformed into subtyping constraints, which are then solved to yield inferred types. BinSub is implemented using the `angr` framework, and

the authors compare its performance and accuracy to Retypd using the Average Type Distance metric.

TRex's [11] methodology is grounded in constraint-based type inference. TRex distinguishes itself from earlier tools by focusing on reconstructing types that are compatible with the observed behavior rather than the exact source type. The goal is to construct the "nearest" source-level types that capture observable behaviors, using deductive techniques. The analysis begins by converting code to SSA form, enabling precise tracking of each variable's behavior. Afterwards, TRex extracts fine-grained behavioral constraints (such as reading, writing, arithmetic, or dereferencing) for each location. This set of behaviours is the base for Trex's behavior-driven abstraction that describes a variable based on the operations it participates in, rather than relying on names or declared types. Next, colocation analysis groups fields that are accessed together via fixed offsets, suggesting aggregate structures such as structs or arrays. These are further refined through aggregate analysis, which interprets the memory layout and access patterns to construct logical data structures. Type rounding then maps the inferred structural behaviors to C-like types by approximating their operational semantics with known primitive types. Finally, TRex assigns symbolic names and generates C-style type declarations, including recursive and nested structures.

Manta [12], by Ye et al. attempts to address the fact that existing type inference techniques either produce overapproximated types—where tools have broader coverage but low precision due to merging conflicting type hints—or suffer from underapproximation, where high-precision analyses like flow-sensitive or context-sensitive methods avoid overmerging but fail to infer types for many variables due to the lack of usable hints. To tackle this, they proposed a hybrid approach to benefit from both of these techniques. Initially, they perform a less strict, control-flow insensitive type inference pass where they aggressively unify type hints to maximize coverage and identify as many candidate types

as possible. Then, for variables with ambiguous (overapproximated) types, they progressively apply context-sensitive and flow-sensitive refinement passes to prune conflicting type hints and resolve more precise, site-specific types. This staged refinement is supposed to provide high recall without sacrificing precision.

Discussion: Constraint-based types are inferred from how values are used in operations, memory accesses, and function calls. Constraint solving supports *conservative yet precise* reasoning by producing upper and lower type bounds instead of forcing a single potentially incorrect type, which allows uncertainty to be represented explicitly. However, constraint generation depends on informative usage patterns; variables that are weakly used or only copied between registers may remain imprecisely typed. This sensitivity is compounded by binary variability: because constraints are read off concrete usage patterns, compiler choices, target architecture, and optimization level can all reshape the evidence available to the solver. Higher optimization levels, for instance, may inline functions, promote variables to registers, or eliminate the very memory accesses and library calls that anchor type constraints, while different compilers and architectures emit different idioms for the same source-level operation. Constraint generation also typically operates on a lifted intermediate representation, so any unsoundness in disassembly or IR lifting propagates directly into the constraint system and can silently corrupt the inferred bounds. Scalability can be a further challenge, since solving complex subtyping constraint systems may have high computational cost in the worst case, which has motivated later work such as algebraic subtyping to improve efficiency. In addition, conservative inference may produce type ranges or abstract numeric categories rather than concrete source-level types, requiring manual interpretation by reverse engineers. These trade-offs place constraint-based type inference between highly precise formal analyses and coarse heuristic methods.

Type inference is challenging due to the uncertainties introduced during compilation. Compilation is inherently lossy, causing different source-level variables and data structure fields to appear indistinguishable in assembly code. In addition, behavioral patterns used as recovery hints may arise coincidentally rather than reflecting true program structure, which further complicates inference. These uncertainties make deterministic recovery unreliable and motivate probabilistic reasoning to combine multiple uncertain hints and infer the most likely variable and structure types.

OSPREY [34] is a probabilistic variable and data structure recovery technique. Compilation is a lossy process and formulating general rules to recover variables is very difficult. OSPREY tries to address the uncertainty of variable information recovery from binaries using random variables and probabilistic constraints. It uses different hints to predict the variable information, such as data-flow hints that take data-dependency into consideration. It also defined hints for memory access patterns that use the fact that fields of the same data structures would be accessed in a unified manner. Another kind of hint it uses is the points-to hint which relates two variables using pointer analysis. Afterwards, it constructs probabilistic constraints where the predicates describe the structural and type properties of memory chunks, which are denoted by random variables. These constraints are solved to predict the variable information.

Stride [16] is another statistical approach to predict type information from decompiled binaries. Instead of analyzing entire functions, Stride focuses on extracting the *usage signature* of individual variables. It adopts the concept of n-grams [69], where an n-gram is a contiguous sequence of n tokens in decompiled source code. For each variable, Stride captures the n-grams that appear immediately before and after its use. Prior to extraction, the decompiled source code is normalized to improve generalization. Stride then

constructs a large database of n-grams mapped to known variable types. The intuition is that if the usage signature of a new variable resembles those in the database, its type is likely to be similar. At inference time, Stride matches the observed n-grams of an unseen variable against this database and predicts its type based on the frequency and similarity of matches.

Discussion: Statistical methods treat type inference as a problem with uncertainty rather than a fully deterministic analysis. Compilation removes important source-level information, and static or dynamic analysis may produce incomplete or conflicting hints about variable and structure types. Instead of relying on fixed rules, statistical approaches combine multiple weak hints and estimate the likelihood of possible types. This helps reduce the effect of incorrect evidence and allows uncertainty to be represented clearly. However, when useful behavioral information is limited or contradictory, the inferred results may remain ambiguous. Overall, statistical inference offers a practical way to balance precision and coverage and is best used together with deterministic analysis rather than as a complete replacement.

Type Propagation (Type Sources and Sinks)

Type propagation is the process of spreading known type information through the data flow of a program so that unknown types can be inferred. When a value with a known type moves through assignments, memory stores and loads, or function calls, static analysis tracks these flows across registers, memory, and inter-procedural call chains to transfer the type information to related variables and arguments. This is especially useful in binary analysis, where documented API functions provide initial type hints that can be propagated through internal functions to infer the types of undocumented system calls. Achieving this requires tracking register and memory states and summarizing function behavior so that

type information can move across function boundaries efficiently, enabling scalable and practical type inference.

NTFUZZ [35] by Choi et al. infers system call signatures of Windows operating system. NTFUZZ is a type-aware kernel fuzzing [70] framework that facilitates Windows kernel testing. To generate meaningful test cases, system call type information is crucial. Unlike Linux, windows system calls are widely unknown and undocumented which led NETFUZZ to develop a type inference algorithm specifically for the system calls. Windows provides documentation for API functions, which are used by the User Applications, System Processes, and Services. The API functions make calls to the undocumented system calls and the signature of the API functions are known. NTFUZZ's type-inferrer algorithm uses the call graph to propagate the known type information(from documented APIs) to the entire program in a bottom-up style to infer the types of the system calls.

Discussion: Type propagation enables scalable type inference by reusing known type information and spreading it through data-flow relationships across registers, memory, and function calls. This provides an easy and precise way to infer types of more variables from known variables. However, its accuracy depends on precise data-flow tracking; imprecise analysis, unresolved indirect calls, or missing execution paths can propagate incorrect or incomplete types. In addition, propagation alone cannot infer types when no initial type hints are available, limiting coverage for weakly used values. As a result, type propagation is powerful for improving scalability and coverage, but it is typically combined with other analysis techniques to achieve reliable type recovery.

2.3.7 Type Inference in Different RE Tools

Reverse engineering (RE) tools increasingly incorporate type inference mechanisms to aid analysts in recovering high-level program semantics from stripped binaries. These tools employ a wide variety of methods, including value set analysis, signature matching, control and data-flow tracking, pattern recognition, and symbolic execution. However, because

each tool often combines several of these techniques in proprietary or non-standardized ways, categorizing them strictly according to a single taxonomy would be complex and potentially misleading. For this reason, we treat RE tools as a separate category and summarize their type inference capabilities in a unified section. Our discussion includes both open-source and commercial tools such as Ghidra [2], IDA Pro [3], RetDec [14], Radare2 [71], Binary Ninja [4], and CMU BAP [63].

Ghidra [2] is one of the most widely used reverse engineering tools, offering features such as disassembly, control flow graph generation, decompilation, and type recovery. It performs Value Set Analysis (VSA) for variable recovery[6] on its intermediate representation called Pcode. Ghidra uses multiple strategies for recovering type information. For instance, it stores common library function signatures and matches them to calls in the binary to infer prototypes [2]. This recovered information is then propagated to other variables. Additionally, Ghidra employs rule-based heuristics that analyze how memory is accessed to predict data types [2, 34]. For example, the pattern for accessing a global integer array is different than accessing a heap-allocated integer array.

IDA Pro [3] is another widely adopted proprietary reverse engineering platform. It is closed source, but its type recovery mechanisms are well-documented. Within each function, IDA performs a detailed analysis of the stack pointer to infer the function's stack frame and identify stack-based variables [3]. It applies pattern-matching rules to detect data structure fields—e.g., by recognizing register-based base-plus-offset field accesses [34]. For type recovery, IDA combines static analysis with a signature matching mechanism (FLIRT) to identify known library functions[7] . Once identified, the type information of these functions is propagated to other variables in the program [3].

RetDec[8] [14], an open-sourced reverse engineering toolchain developed by Avast is a retargetable machine-code decompiler based on LLVM [72]. Given a binary, RetDec

[6]Ghidra. *Ghidra GitHub Repository — `varnode.hh` (Decompiler component)*. GitHub; accessed 2025-08-26.

[7]Hex Rays. *Hex Rays Documentation*. Hex Rays website; accessed 2025-08-26.

[8]*Avast RetDec GitHub Repository*: `https://github.com/avast/retdec`

lifts the executable to LLVM IR. In this step, each machine instruction of the executable is converted into a sequence of IR code which should emulate the functionality of the instructions, including CPU register-level computations, memory updates, and other side effects [73]. Afterward, the obtained IR is refined and optimized in multiple passes, and variables are identified with their types. The final IR is converted to high-level C/C++ code including variables and types.

Radare2[9] has variable detection and type prediction features. However, the predicted types are not the same as the C standard. For example, `int8_t` or `uint64_t` are predicted instead of char or int. Radare2's type inference feature relies heavily on matching preloaded function prototypes and calling conventions [71]. It also supports basic Run-Time Type Information (RTTI) recovery based on virtual table parsing.

Binary Ninja [4] initially transforms the instructions to Binary Ninja Intermediate Language (BNIL). Then it performs value set analysis within BNIL for variable [10] . Binary Ninja is closed source, so it is unclear what kind of static analysis it performs for type inference. However, similar to other reverse engineering tools, it also has a function signature database system, which it uses to recognize the parameter types of matched functions. Similarly, it also infers type information from library function calls.

The Carnegie Mellon University Binary Analysis Platform **(CMU BAP)** [63] is another popular suite of utilities and libraries that enables binary program analysis. However, BAP does not directly support type inference[11] . **Ddisasm** has a type recovery implementation based on Retypd [9].

[9]https://github.com/radareorg/radare2

[10]*Brian Potchik, "Architecture Agnostic Function Detection in Binaries," Binary Ninja Blog*: `https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html`

[11]CMU BAP. *BAP GitHub Repository*. GitHub issue #1056; accessed 2025-08-26.

2.3.8 Learning-Based Type Inference

Many of the latest type predicting tools have used various kinds of machine learning methods, such as Random Forest Classifiers [74], Support Vector Machine (SVM) [75], Convolutional Neural Network (CNN) [76], Recurrent Neural Network (RNN) [77, 78], Long Short-Term Memory (LSTM) [79], Graph Neural Networks [80] and Transformers [81]. Machine learning models learn or train on a large number of examples and then make predictions on unseen examples based on their learning. This method of type prediction has a number of benefits and limitations.

One of the reasons why machine learning tools are a good fit for binary analysis tasks is that they can easily generalize, given enough samples. For example, many non-machine learning tools have to cover variations in binaries caused by different optimization levels, compilers, architectures, etc. Machine learning models can learn to analyze any kind of variations given that there are enough samples to train on. On the other hand, this is also a limitation, because without a large amount of samples the models will not work well. There are architectures for which collecting a large amount of source code is not easy. Collecting a large number of binaries and performing analysis on them is also inherently resource-intensive in any case.

Another limitation of machine learning models is they might not work well with outliers, because they learn from what they see during training and try to use that knowledge for future predictions. For example, if someone generates a binary with handwritten instructions or uses an custom compiler, the machine learning models will perform poorly,because the model never got a chance to train on similar samples.

Traditional Classifier-Based Models for Type Inference

Traditional classification models are widely used in early machine learning approaches for type prediction in binaries. These models assign data points to predefined type categories based on patterns learned from training examples. They rely on handcrafted features—such as instruction mnemonics, operand types, or memory access patterns—and apply classifiers like Support Vector Machines (SVM) [75], Random Forests [74], Decision Trees [82], and Naive Bayes to make predictions. The following tools apply such classifiers for type inference, often in multi-stage pipelines or with additional heuristics to enhance accuracy.

BITY [37, 83] uses classification to recover types, starting with variable discovery using a value-set analysis–inspired method [55]. This is one of the first tool to use machine learning method for recovering type information. It then collects instructions related to each discovered variable by analyzing data dependencies. This set of instructions represents how a variable is stored, interpreted, and manipulated, which provides important information for type recovery. These instruction sets are processed into feature vectors, with Term Frequency-Inverse Document Frequency (TF-IDF) [84] used to retain relevant features. The extracted features are then passed to a classification model combining SVM and Random Forest to infer variable types.

TypeMiner [36] uses a combination of Random Forest classifiers [85] and Linear Support Vector Machine [86] for the prediction task. First, it performs trivial static analysis on the binary and generates the disassembly and corresponding data flow graph. Then they normalize the instructions in a custom way. For example, they remove the information from the instructions not required for type prediction, such as addresses and specific register names. Finally, their normalized instruction consists of the instruction's mnemonic and the normalized operands. The normalized operands consists of two parts: the type of the operand and the width of the operand's value. Data Dependency Graphs are used to

find related instructions from the program. Those instructions are then converted into embeddings and fed to the model. The TypeMiner's classification is done in multiple steps. For example, the first step predicts if it's a pointer or not. Then another step decides non-pointer types. Then a final classifier determines if the variable is signed or not.

DEBIN [15] is a machine learning-based system for recovering debug information—such as symbol names, types, and variable locations—from stripped binaries. It targets ELF binaries across x86, x64, and ARM architectures and aims to enhance binary comprehension by predicting DWARF-style information. DEBIN first lifts binaries into an intermediate representation using BAP-IR [63] and classifies program elements (e.g., registers, memory offsets) using an Extremely Randomized Trees [87] classifier to distinguish variable-related elements from incidental ones. This step separates elements that were used to represent a variable from those that were used for other purposes, such as temporary usage for optimization. Next, DEBIN generates a dependency graph from the BAP-IR which represents different units of the programs and their dependency relationships. This dependency graph works as a version of Conditional Random Field (CRF) [88], which makes predictions based on the relationship of a node with its neighbors, in this case, context. Then a probabilistic prediction is made on the variable and type information, based on the dependency graph, which is later encoded as DWARF information.

Escalada et al. [18] focus on predicting function return types using statistical and machine learning classifiers. During dataset creation, they modify C source functions to insert instrumentation that helps link return statements to function calls. This instrumentation breaks down a function into multiple chunks where each chunk is associated with each function invocation and return statement. Since feeding full chunks into models might exceed input length limits, they apply feature selection to extract the most relevant instructions. For classification, the authors experimented with 14 models, including AdaBoost, Naive Bayes, Decision Trees, Perceptron, Logistic Regression, and multiple SVM variants.

Discussion: Traditional classifier-based methods provide an early data-driven approach to type inference by learning patterns from handcrafted features extracted from binaries. These models are relatively simple, interpretable, and efficient to train, and they can achieve reasonable accuracy when informative features and labeled data are available. However, their performance depends heavily on feature engineering and dataset quality, which limits generalization across architectures, compilers, or optimization levels. In addition, fixed type categories and shallow feature representations make it difficult for these models to capture complex program semantics or structural relationships. As a result, classifier-based approaches are useful for initial type prediction but are often complemented by more expressive probabilistic, constraint-based, or deep learning methods for improved accuracy and robustness.

CNN-Based Models for Type Prediction

Convolutional Neural Networks (CNNs) [76] are widely used for tasks involving image, audio, signal, and sequential data. While CNNs are not the most common choice for processing textual sequences, they can still be effective for text classification, particularly when paired with pre-trained embeddings like Word2Vec.

Cati [38], developed by Chen et al., leverages CNN-based architectures. The tool applies several preprocessing steps to the input instructions, such as replacing addresses, function names, and numeric values with special tokens. For each target instruction, Cati considers a context window of 10 instructions before and after the target. This design is based on a statistical analysis showing that spatially local instructions tend to be more informative. The preprocessed instruction sequence is encoded using Word2Vec [89] and fed into a multi-stage CNN classifier. This model predicts type information in several stages: first determining whether a variable is a pointer, then classifying pointer types (e.g., struct, void), or, if not a pointer, inferring the basic data type. One key insight of Cati is its handling of *orphan variables*—those that lack sufficient local context due to sparse usage.

The authors suggest that instructions close to one another usually operate on variables of similar types and propagating type information as a cluster would be beneficial. **Cati++** [90] is an extension of Cati that uses a different embedding algorithm.

Sequence Models for Type Inference

Natural Language Processing (NLP) models are a strong match for type prediction tasks because they are designed to process sequences. Executable programs, like natural languages, can be seen as sequences—specifically, sequences of assembly instructions. Although assembly is not a natural language, it has structure, syntax, and semantics. This makes it suitable for models that are built to understand linguistic patterns.

Transformer based Large Language Models (LLMs) are designed to capture contextual relationships between tokens in a sequence. When applied to assembly instructions, understanding the surrounding program context becomes essential for reasoning about instruction behavior and variable usage. NLP models can effectively learn these contextual and semantic patterns. Recent LLMs such as ChatGPT and LLaMA have shown strong performance in general binary analysis tasks by leveraging their contextual awareness and transfer learning capabilities. Although their use in type prediction is still unexplored, it presents a promising direction for future research.

EKLAVYA [39], by Chua et al., is one of the first tools to use a deep learning NLP model for recovering type information. They use a Recurrent Neural Network [91] to predict function signatures. Unlike most tools that use disassembled instructions or decompiled code, EKLAVYA takes raw bytes of functions as input. The Skip-gram negative sampling [89] method is used to train a word embedding model to generate embeddings from the raw bytes. Skip-gram is an unsupervised learning method where the task is to find the most related words for a given word. In the case of EKLAVYA, the task is to find the most related instructions for a given instruction. This method generates embeddings for

instructions that retain the semantic relations between them. The output of the model is the count of function arguments and the corresponding types of those arguments.

DIRTY [17], created by Chen et al., is a Transformer-based model that jointly performs type prediction and variable name prediction, based on the insight that these tasks can reinforce each other. For example, programmers often use different naming patterns for integers versus characters, which can aid in type inference. DIRTY takes as input decompiled source code produced by IDA Pro, rather than raw assembly instructions. It also incorporates interim analysis results from IDA, such as variable size, storage location (e.g., register or stack offset), nested type structure (e.g., structs), and member offsets (e.g., in arrays or structs). This data-layout information is used to learn a soft mask that guides the Transformer decoder. If the initial prediction is incompatible with the known data layout, the soft mask lowers the probability of that prediction. While DIRTY is demonstrated using IDA Pro, it can also work with any decompiler that provides both decompiled code and rich data-layout information.

Building on DIRTY, **Cao and Leach** [92] investigate whether DIRTY's performance can be replicated using Ghidra. They retrain the DIRTY model on Ghidra-decompiled binaries and adapt the training process to handle Ghidra-specific challenges stated by DIRTY, such as missing DWARF links and variable aliasing. Their goal is to evaluate whether Ghidra's outputs are sufficient for deep learning-based type inference, and their results show that, with minimal changes, DIRTY achieves comparable accuracy using Ghidra.

SnowWhite, proposed by Lehmann et al. [40, 93], is an NLP-based approach designed to recover source-level function type signatures from WebAssembly binaries. WebAssemblies allow execution of code written in languages other than JavaScript on the browser at a near-native speed [94]. Code written in languages such as C/CPP, C#, GO, or Rust can be compiled into WebAssemblies. Another advantage of WebAssemblies is that it allows developers to use the existing ecosystems of languages like C. WebAssemblies

are crucial because these allow a browser-based app to perform computation-intensive tasks. A key property of SnowWhite is that it predicts type information in a format similar to the DWARF debugging format. This is because code from different languages can be compiled as WebAssemblies and DWARF is widely supported by several compilers for different source languages. By targeting DWARF-like output, SnowWhite avoids tailoring its predictions to any specific source language, instead learning a generalizable representation. The model architecture consists of a bidirectional LSTM with global attention [95], and uses two separate models to predict return types and parameter types of functions, respectively.

StateFormer [1] uses RoBERTa [96], a transformer model originally developed for natural language processing, to predict variable types from disassembled instructions. Since transformer models are not generally trained on assembly code, the authors first pretrain the model on a related task to help it understand low-level instruction behavior. This pretraining task is called Generative State Modeling (GSM), which teaches the model to simulate the effects of instructions on register states. For example, GSM encourages the model to learn that `add ecx, 3` increases the value of `ecx` by 3, or that `inc ecx` increments it by 1. Once the model acquires this instruction-level semantics, it is fine-tuned on the downstream task of type inference. This two-stage approach enables StateFormer to better capture the semantics of binary code before applying type prediction.

SeeType [97] is another transformer-based tool for type prediction. It pre-trains a BERT [98] model using masked language modeling and a data-dependency prediction objective to better adapt the model to assembly code. The pretraining is supposed to familiarize the BERT model with assembly language, which was originally pre-trained on natural languages. To formulate the input for the model for the type prediction task, it tries to capture the context of a target instruction. Sometimes a function can be large, and the whole function might not fit into the model as input. To solve that problem, they used program slicing [99, 100] methods. It utilized both forward and backward slicing to

formulate the input rich in context for the target instruction. For training SeeType, they also curated a large dataset of open-source programs. They also provided a method for generating ground truth by using both debug-information and the source code. They also reflect on the importance of tokenization and the importance of numerical values in machine code, and propose a method to handle numerical values in a better way.

YALI [101] is another transformer-based tool that uses Ghidra's high-level P-Code intermediate representation. YALI focuses specifically on recovering function parameter types and callsite argument types. YALI fine-tunes a pre-trained DistilBERT [102] model on P-Code argument program slices, with ground truth labels extracted from DWARF debugging symbols.

ReSym [41] combines LLM-based predictions with Prolog-based reasoning to recover variable names, types, and user-defined structures. It employs two specialized LLMs: VarDecoder for recovering local variable names and types, and FieldDecoder for identifying structure field accesses. These models are applied to decompiled code, and their predictions are processed as logical facts, such as variable declarations, field accesses, callsite relationships, and data flows. ReSym then uses a logic-based inference system to group variables by type, propagate type information across functions, and resolve inconsistencies through reasoning rules (e.g., intra/inter-procedural flow, type-agnostic argument detection).

IDIOMS [42] is a neural decompiler that jointly predicts code and user-defined types using a large language model. It leverages neighboring functions from the call graph to provide a richer context for type prediction. The model is finetuned with decompiled code and call graph context as input, with original source code with full type definitions as the output. Ablation studies show that removing neighboring context significantly reduces type prediction accuracy. However, due to the token limit of LLMs, including multiple functions in the input is not always feasible.

Discussion: Sequence-based models extend traditional classifier approaches by learning contextual patterns directly from instruction sequences instead of relying on handcrafted features. This improves generalization and allows the models to capture semantic relationships across instructions and functions. Compared to graph-based methods such as GNNs, sequence models are simpler to train and scale well with large datasets, but they may miss long-range structural dependencies that are naturally represented in graphs, such as data-flow or call relationships. In practice, sequence models complement both classifier-based and GNN-based approaches: classifiers provide lightweight predictions from explicit features, sequence models capture contextual semantics, and graph models encode structural program relationships. Combining these perspectives can lead to more accurate and robust type inference in binaries.

Handling Numerical Values by NLP tools Addresses and other numerical values are useful because they carry information about control flow and the storage locations of source-level variables [54]. They indicate relationships between instructions and are necessary for reconstructing the control flow of the program. NLP models have limitations in processing and understanding numerical values [103, 104]. Different tools have tried various methods for addressing this issue, as numerical values are a significant part of machine code. ByteTR replaces all the constants with a special token. Cati removes all the address-related numeric values but keeps small numbers like offsets. Escalada [18] also performs normalization on the numeric values, replacing numeric values with special tokens. Stride [16] also replaces numbers over 100 into several tokens depending on the magnitude. TypeMiner gets rid of any numeric values during normalization. StateFormer [1] used a different method to handle numeric values, using the Neural Arithmetic Unit (NAU) [105], which creates embeddings meant to represent the meaning of numbers used in arithmetic operations.

Graph-Based Neural Models for Type Inference

Graph Convolutional Networks (GCNs) [106,107] are a class of deep learning models specialized for processing graph-structured data. Unlike traditional neural networks that operate on images, audio, or text, GCNs take graphs—comprising nodes (entities) and edges (relationships)—as their primary input. In binary analysis, programs are often represented as graphs, such as control-flow graphs (CFGs) or data-flow graphs (DFGs). GCNs offer a powerful way to analyze these structures automatically and infer type information from them.

TIARA [43] targets the recovery of STL container classes [108] in C++ binaries using a combination of principled and learning-based techniques. It employs inter-procedural forward slicing [99,109,110] to extract instructions relevant to a target variable. The slicing captures instructions influenced by the variable (forward slice) but does not use backward slicing, unlike other tools such as Cati. TIARA constructs a control-flow graph from the sliced instructions and uses a GCN to predict the specific STL container type.

TYGR [44] is a Graph Neural Network [80]-based tool designed to infer both basic and struct types in stripped binaries. It is architecture-agnostic and operates on VEX IR [111], which provides a uniform intermediate representation across different platforms. TYGR begins by constructing a control-flow graph (CFG) for each function and simulates execution along various non-cyclic paths to produce path-specific data-flow graphs for each variable. These per-path graphs are then aggregated into a function-level data-flow graph, which is encoded into an embedding and used by a classifier to predict variable types. The authors also introduce a new dataset, TYDA, and demonstrate that TYGR outperforms several baseline methods in accuracy.

DRAGON [45] is another GNN based approach for predicting primitive data types from decompiled binaries. It takes Ghidra's decompiled Abstract Syntax Trees (ASTs) as the starting point. Each node in the graph is encoded with its AST kind, associated operation (e.g., +, ==), and initial type information. The edges represent syntactic roles

such as operand order or control structure hierarchy. DRAGON identifies all references to a variable in the AST, collects their k-hop neighborhoods, and merges them into a single ”variable graph” that captures the variable’s usage context. A GNN then uses this input graph to predict the variable’s type. Recognizing that machine learning models often make incorrect predictions without a clear explanation, DRAGON augments each prediction with a confidence estimate—providing a measure of reliability for downstream use. *Discussion:* Graph-based neural models capture structural relationships in programs by learning from control-flow, data-flow, and syntactic graphs rather than only local instruction context. This allows them to model long-range dependencies, variable interactions, and aggregate structures more naturally than classifiers or sequence models. As a result, GNN-based approaches often achieve higher accuracy for complex type recovery tasks such as struct inference or container identification. However, they require precise graph construction, which depends on the quality of disassembly, IR lifting, and data-flow analysis, and this preprocessing can be computationally expensive and error-prone. In addition, graph models are typically harder to scale and train compared to sequence models.

2.3.9 Hybrid Methods

Hybrid methods integrate multiple analysis paradigms—such as static analysis, dynamic analysis, machine learning, and heuristic reasoning—within a unified type recovery pipeline. Instead of relying on a single source of evidence, these approaches combine complementary signals to improve both precision and coverage. This design helps address the limitations of individual techniques, such as incomplete execution coverage in dynamic analysis, imprecision in static reasoning, or ambiguity in learning-based predictions. As a result, hybrid systems aim to produce more reliable and semantically meaningful type information for complex binaries and diverse compilation settings.

TypeSqueezer [6] by Lin et al. combines static and dynamic analysis for recovering function signatures. During analysis, it inspects the registers and the memory locations

involved in argument passing at runtime, inspecting the contents stored in the argument loaded registers to infer their types. For instance, if the content of the register points to the living address space of the process, it considers that value as a reference, otherwise simply a value. This method can be repeated one more time to detect pointers of pointers. TypeSqueezer uses more such heuristics and other static analysis methods for predicting the types of the function parameters and return. This type recovery method is called value-based type inference. It does not distinguish fine grained types but focuses on values, references, references to references, etc. TypeSqueezer uses the recovered type information to devise a method to improve Control-Flow Integrity [20, 112] to prevent forward-edege attacks [21]. Basically it prevents any call to functions whose signature is not considered safe. Though incorporating static analysis and dynamic analysis for type prediction is interesting, they did not evaluate their function signature recovery performance directly. TypeArmor [113] is another similar tool that infers function signatures for Control-Flow Integrity.

WasmHint [114] targets the recovery of function type signatures from Wasm contracts compiled from Rust, addressing limitations of prior work [40] that focused mainly C/C++-oriented type systems. WasmHint first constructs a Wasm Control Flow Graph (wCFG) and applies an operation-centric code slicing technique that dynamically simulates Wasm instruction execution to extract refined instruction slices associated with parameter usage and return-related operations. WasmHint's input embedding combines three sources: opcode instruction slices, control-flow graph structure, and constant values. Opcodes are tokenized and transformed into embeddings. The WCFG is converted into a feature vector, and the constants are also represented through learned embeddings with attention. These representations are combined using an attention-based semantic aggregation mechanism to generate the final model input. Type inference is performed in two stages: WasmHint first predicts basic types using a classification model, and then

generates composite types using a recurrent decoding model to handle recursively defined type structures, such as pointers, struct, trait, etc.

ByteTR [46] begins by performing SSA, def-Use analysis, intra-procedural analysis, and inter-procedural analysis which results in Variable Propagation Graphs. The propagation graph presents the variable's access pattern characteristics and program semantics. This graph is then further processed and normalized for constants and fed to a trained Gated Graph Neural Network (GGNN) for type prediction.

TypeForge [8] introduces a novel method for type prediction with a two-stage pipeline. The first stage synthesizes multiple candidate composite type declarations using a novel graph abstraction. Then the best-fit type declaration is selected by evaluating how well each improves the readability of decompiled code. TypeForge begins by taking decompiled code as the input and produces a type graph from the decompilation. Then, it applies an Adaptive Sliding Window algorithm to synthesize multiple candidate composite type declarations per variable, handling ambiguities like struct flattening and pointer aliasing. The result is a pool of candidate type declarations. The second stage selects the best-fit composite type from the candidate pool using an LLM-assisted double-elimination process to perform pairwise comparisons of generated code variants to determine which version is more readable.

HyRES [7] combines static analysis, large language models (LLMs), and heuristic reasoning in a hybrid framework to recover structure variables and their layouts from stripped binaries.

- Static Analysis with Datalog - HyRES lifts binary code into LLVM IR and applies a Datalog-based reasoning framework to perform static analysis. It identifies structure pointers and infers field layouts by analyzing memory access patterns and propagating data flow across and within functions.

- Semantic Recovery via LLM - HyRES normalizes decompiled code from IDA Pro [3] to resemble source-level C, then queries a general-purpose LLM to infer meaningful field names and types, enhancing the semantic understanding of structure fields.
- Heuristic Structure Aggregation - Finally, HyRES merges different instances of the same structure by comparing their layout and semantics. It uses text embeddings and carefully designed heuristic rules to identify them and then aggregate partial structures into a more complete and accurate form.

2.4 Comparison of Tool Characteristics and Capabilities

This section presents a comparative analysis of existing type inference tools across several key dimensions, including their input representations, use of contextual information, supported programming languages, and the scope of type abstractions they recover. By examining these properties side by side, we highlight both shared design patterns and distinctive capabilities that influence each tool's applicability and effectiveness in different analysis scenarios.

2.4.1 Cross-Cutting Comparison of Method Categories

The per-category discussions in Sections 2.3.5–2.3.9 examined the strengths and limitations of each method family in isolation. This subsection instead compares the families directly along the five dimensions that most affect practical deployment: the granularity of types they recover, their coverage and scalability, the program inputs they require, their dependence on the quality of intermediate-representation (IR) lifting and decompilation, and their practical usability. These dimensions form the columns of Table 2.3, with each method family occupying a row; the following discussion draws out the cross-cutting trade-offs that the table summarizes.

Methods Used			
Tool	**Formal Methods**	**Statistical Analysis**	**Machine Learning**
IDA Pro [3]	Code Pattern Identification	✗	✗
Ghidra [2]	VSA, Code Pattern Identification	✗	✗
RetDec [14]	SSA	✗	✗
BAP [63]	✗	✗	✗
Binary Ninja [4]	VSA	✗	✗
angr [23]	VSA, Symbolic Execution	✗	✗
TIE [33]	VSA	✗	✗
TypeMiner [36]	✗	✗	Random Forest classifier
OSPREY [34]	✓	probabilistic constraints	✗
EKLAVYA [39]	✗	✗	Skip-gram, RNN
WasmHint [114]	✗	✗	Classifier, RNN
BITY [37]	VSA	✓	SVM
Escalada et al. [18]	✗	✓	Classifier Models
DEBIN [15]	✓	✓	Randomized Tree Classification
DIRTY [17]	✗	✗	Transformer
Cao and Leach, 2023 [92]	✗	✗	Transformer
ReSym [7]	Constraint Solving	✗	LLM
IDOMS [7]	Call graph	✗	LLM
SeeType [97]	Program Slicing	✗	BERT
Yali [101]	P-Code Slicing	✗	DistilBERT
StateFormer [1]	✗	✗	Transformer
SnowWhite [40]	✗	✗	Bidirectional LSTM
TypeForge [8]	variable propagation	✗	LLM
HyRES [7]	Static Analysis, Constraint solving	✗	LLM, Transformer
CATI, CATI++ [38, 90]	✗	✓	CNN
TYGR [44]	✗	✗	Graph Neural Network
TIARA [43]	Program Slicing	✗	Graph Convolutional Network
DRAGON [45]	✗	✗	Graph Neural Network
ByteTR [46]	SSA, cross-function variable propagation	✗	Gated Graph Neural Network
TypeSqueezer [6]	Dynamic Analysis and Heuristics	✗	✗
Retypd [9]	VSA, Symbolic Variables		✗
TRex [11]	SSA, Constraint Solving		✗
NTFUZZ [35]	Data Flow Analysis	✗	✗
STRIDE [16]	✗	N-gram	✓
OOAnalyzer [32]	Symbolic Analysis, Forward Reasoning	✗	✗
OBJDIGGER [31]	Symbolic Execution	✗	✗
Manta [12]	SSA, Constraint Solving	✗	✗
BinSub [10]	VSA, Symbolic Variables	✗	✗

Table 2.2: Overview of Tool Classification by Methodology: This table categorizes tools based on their major methodologies.

Table 2.3: Cross-cutting comparison of major type inference method categories. Granularity: P = primitive, S = structure/-composite, C = class/OO, F = function signature. In the Strengths & Weaknesses column, + denotes strengths and – denotes weaknesses. Representative tools: Dynamic—Howard [26], REWARDS [24], MemPick [29]; Formal—TIE [33], Retypd [9], TRex [11], OOAnalyzer [32]; Statistical—OSPREY [34], Stride [16]; Classical ML—BITY [37], TypeMiner [36], DEBIN [15]; Sequence/NLP—EKLAVYA [39], StateFormer [1], DIRTY [17], ReSym [41]; GNN—TYGR [44], TIARA [43], DRAGON [45], ByteTR [46]; Hybrid with LLM—HyRES [7], TypeForge [8], ReSym [41].

Category	Type Granularity	Required Input	IR/Decompiler Dependence	Coverage & Scalability	Strengths & Weaknesses	Usability & Maturity
Dynamic Analysis	S, C; heap layouts, recursive structures; some P (ARTISTE)	Executable + test inputs; instrumented runtime environment	Low: observes concrete machine state	Limited to executed paths; instrumentation overhead; evadable by anti-analysis code	+ Precise runtime layouts; finds heap structures statics miss. – Covers executed paths only; setup-heavy; evadable.	Needs runnable binary and environment setup; older tools unmaintained
Static: Formal Reasoning	P, S, F; polymorphic types (Retypd); C via logic rules (OOAnalyzer)	Disassembly or IR; extracted facts and constraints	High: unsound lifting corrupts constraints and rules	Whole binary; cubic worst-case solving (Retypd), eased by algebraic subtyping (BinSub [10]); state explosion in symbolic exec.	+ Sound, explainable, robust; no training needed. – Conservative output; costly solving; fragile handcrafted rules.	Output needs analyst interpretation; rules need per-compiler tuning; adopted in Ghidra/angr
Static: Statistical	P, S; recovery under uncertainty	Static-analysis hints or normalized decompiled code (n-grams)	Moderate–high: hint quality bounds results	Whole binary; scalable in general	+ Handles noisy/conflicting hints; explainable. – Ambiguous output when hints weak; depends on other methods for hints.	Represents uncertainty explicitly; complements deterministic analysis; research prototypes
Learning: Classical ML	Mostly P; return types [18]; DWARF-style info (DEBIN)	Handcrafted features from disassembly or IR	Moderate	Fast inference; generalization limited by feature engineering	+ Simple, interpretable, cheap. – Capped by handcrafted features; poor generalization.	Fixed type sets; superseded in accuracy by deep models
Learning: Sequence / NLP	P, F; names+types jointly (DIRTY); user-defined S	Raw bytes, disassembly, or decompiled code	Low for raw bytes; high for decompiled-code models	Whole binary; token limits force slicing/windows; quadratic attention; GPU inference	+ Learns context; joint name+type prediction; recovers higher-level abstractions. – Data/-compute heavy; token limits; weak on unseen compilers/archs.	Strong in-distribution accuracy; few usable interfaces
Learning: Graph-Based (GNN)	P, S; STL containers (TIARA); cross-function variables (ByteTR)	CFG/DFG/AST or propagation graphs from IR or decompiler	High: graphs need accurate lifting and data-flow	Whole binary; captures long-range structure; graph construction and training expensive	+ Models how variables relate program-wide; robust for composite types. – Expensive preprocessing; hardest to train/deploy.	Top accuracy on structural tasks; confidence estimates (DRAGON)
Hybrid with LLM	S layouts + semantics (HyRES); composite types (TypeForge).	Multiple: binary, IR, decompiled code, extracted features, LLM queries	Varies; typically high	Combines signals for precision and coverage; cost is sum of components; LLM stages add latency	+ Semantic recovery via LLM, grounded by static analysis. – Costly LLM queries; hallucination risk.	Best precision/coverage balance; complex pipelines; very recent

Type granularity. The categories occupy distinct positions on a spectrum from primitive types to high-level abstractions, and no single family dominates the whole range. Dynamic methods are strongest at the structural end: by observing concrete memory allocations, pointer linkages, and access patterns at runtime, they reconstruct heap data structures, recursive types, and—through object lifetime and destructor analysis—class hierarchies that are difficult to recover any other way. They typically recover primitive types only as a byproduct, with ARTISTE [30] being a notable exception that recovers both composite structures and their constituent primitives. Static formal methods span the widest range within a single paradigm, recovering primitives, structures, function signatures, and even polymorphic types through constraint solving, while object-oriented abstractions are reached only by the logic-based branch such as OOAnalyzer [32]. Learning-based methods show a clear historical progression in granularity: early classical-ML and sequence models were largely confined to primitive types and function signatures, and only the more recent GNN, LLM, and hybrid approaches push learned recovery upward to composite and user-defined structures. This pattern suggests that recovering high-level abstractions remains the hardest granularity tier—accessible to formal reasoning, runtime observation, or, most recently, large language models, but not yet routine for any category.

Coverage and scalability. A fundamental divide separates dynamic from static and learning-based methods. Because dynamic analysis types only the code that actually executes, its coverage is bounded by input quality and is inherently incomplete; achieving adequate coverage requires test drivers, fuzzing, or careful environment configuration, and adversarial binaries can suppress the very behaviors of interest. Static and learning-based methods, by contrast, reason over the entire binary and do not depend on execution, but each family pays a different scalability cost. For formal methods the bottleneck is constraint-solving complexity—Retypd's worst-case cubic solving [9], later mitigated by the algebraic-subtyping reformulation in BinSub [10], and the state-space explosion that

limits symbolic execution. For sequence models the bottleneck is the quadratic cost of attention together with fixed token limits, which forces context-reduction strategies such as program slicing or fixed instruction windows . For GNNs the dominant cost lies in constructing precise data-flow or AST graphs before inference can begin. Thus scalability is a challenge across the board, but its root cause differs by category, which matters when selecting a method for large or complex targets.

Required inputs. The categories also differ in what they consume, which directly affects where they can be applied. Dynamic tools require a runnable executable together with representative inputs and an instrumented or emulated environment—a substantial setup burden that is infeasible for binaries that cannot be safely or completely executed. Static formal methods operate on disassembly or IR and the facts and constraints derived from them, requiring no execution but presupposing reliable disassembly. Learning-based methods are the most heterogeneous in their inputs: some consume raw function bytes, others disassembled instructions, and many of the recent tools consume decompiled source code. This last choice has an important consequence, because it couples the tool to a specific decompiler and inherits its idiosyncrasies. Hybrid systems combine several of these inputs—binary, IR, decompiled code, and LLM queries—within a single pipeline, which broadens the evidence available but also widens the surface on which the analysis can fail.

Dependence on IR and decompiler quality. A property that the per-category discussions tend to treat implicitly becomes visible when the families are compared side by side: their robustness is determined by which upstream artifact they trust. Dynamic methods have the lowest dependence because they observe the concrete machine state directly, sidestepping lifting errors entirely. Formal methods sit at the opposite extreme, since unsound IR lifting silently corrupts the constraint systems and inference rules they rely on, and such errors propagate through the entire deduction. Learning-based methods split

according to their input: models that consume raw bytes or assembly are relatively insensitive to lifting quality, whereas models built on decompiled code or constructed graphs are highly sensitive to decompiler accuracy, slicing precision, and data-flow analysis. This dependence is rarely evaluated explicitly, yet it is a major and underexamined source of error, since a tool that performs well with one decompiler or lifter may degrade substantially with another.

Practical usability and maturity. Finally, comparing the families exposes a persistent tension between accuracy and usability. The methods embedded in mature, widely used reverse-engineering platforms—signature matching, value-set analysis, and heuristic type propagation in IDA Pro [3], Ghidra [2], and angr [23]—are the most usable and the best documented, but methodologically the most conservative, and they have largely not absorbed the advances made by research tools. The most accurate recent systems, by contrast, are the hardest to deploy: GNN, LLM-based, and hybrid pipelines require trained models, GPU resources, and multi-stage preprocessing, are frequently released without usable interfaces, and in several cases are not released at all. A few research tools improve usability in targeted ways—DRAGON [45], for instance, augments each prediction with a confidence estimate that helps analysts triage which results to trust—but these remain exceptions. As discussed in Section 2.6, narrowing this gap, for example by packaging research tools as plugins for established platforms, is one of the more pressing practical challenges in the field.

Summary. Across these five dimensions the categories are complementary rather than competing: each trades coverage, granularity, input cost, IR dependence, and usability differently, with no family dominating all five. This is precisely what motivates the recent shift toward hybrid systems, suggesting that future progress will come less from any single paradigm than from principled combinations that let each method offset the others' weaknesses.

2.4.2 Input Representations Used in Type Inference Tools

This subsection discusses the input representations used by various tools to perform type inference from binary programs. The binary executable is the common starting point for all tools. However, the specific form in which the program is represented to the tool—such as disassembly, intermediate representation (IR), or graphs—varies depending on the analysis method used.

ML Tool Inputs. Machine learning-based tools use a variety of input representations depending on the nature of the models they employ. Tools that use NLP-inspired models typically rely on *sequential representations* of the binary, where the order of instructions or tokens carries semantic meaning. These include:

- **Disassembled instructions:** (e.g., StateFormer [1], CATI [38], SeeType [97], WasmHint [114], SnowWhite [40], Escalada et al. [18])
- **Decompiled source code:** (e.g., DIRTY [17], Cao and Leach [92], STRIDE [16], DRAGON [45], RESYM [41], IDIOMS [42], TypeForge [8])
- **Intermediate representations (IR):** (e.g., DEBIN [15])

These inputs are suited for models like transformers, RNNs, and LSTMs, which are designed to capture context and dependencies across sequences. An exception is EKLAVYA [39], which uses raw function bytes as input for an RNN model.

Other models use non-sequential representations. For example, TYGR [44] constructs a data-flow graph for each function and uses a Graph Neural Network (GNN) to infer types. TIARA [43], DRAGON [45] ,WasmHint [114] also use graphical representations of the program as their input. TypeMiner [36], on the other hand, uses a random forest classifier and processes a feature set derived from an unordered collection of related instructions.

Use of Intermediate Representations Many reverse engineering tools rely on Intermediate Representations (IRs) as a foundation for their analysis. IRs allow these tools to abstract away architecture-specific details and work with a unified, simplified version of the program. This enables the same analysis logic to be applied across binaries compiled for different architectures, as long as the binary can be translated into IR. By separating architecture handling from the core analysis, developers can focus more on improving the analysis itself.

IRs also operate at a higher level of abstraction than raw machine code, which can make tasks like type inference easier and more effective. Some well-known IRs used in reverse engineering tools include:

- Ghidra [2]: P-code
- IDA Pro [3]: Microcode
- Angr [23]: VEX
- BAP [63]: BIL
- Binary Ninja [4]: Low-Level IL (LLIL), Mid-Level IL (MLIL), and High-Level IL (HLIL)
- DEBIN [15] and TIE [33]: BAP's [63] BIL
- Retypd [68]: CodeSurfer IR [65]

However, using IR also has limitations. The translation process from machine code to IR is not always perfect and may introduce errors. Additionally, certain low-level architecture-specific behaviors may be lost or oversimplified in IR, which can hinder accurate program reconstruction or analysis in some cases.

Input for other Principled Methods The other type of tools that take a more principled approach to type recovery use a variety of kinds of input depending on the type

of analysis they perform, ranging from facts about the program (OOAnalyzer [32], OSPREY [34]) and program constraints (Retypd [68], BinSub [10], OSPREY [34]) to inferred types from another tool (DSIbin [27]).

Dynamic Tools. Dynamic analysis tools primarily use the executable binary as their input since the program must be executed to observe its behavior. In addition to executing the program, some tools analyze runtime memory images to extract structural information. For instance, several tools generate memory graphs during execution to aid type recovery:

- DSIBin [27]
- ARTISTE [30]
- MemPick [29]
- DDT [28]

Other dynamic tools rely on known function signatures to infer types. For example:

- REWARDS [24] propagates type information based on calls to known library functions.

2.4.3 Use of Custom Context for Type Recovery

In programming, *program context* refers to the surrounding information, data, and execution environment that influences how a program behaves. In binary analysis, context is crucial for type inference, as it helps understand how a variable or instruction interacts with others in its vicinity. However, many binary functions are large, making it computationally expensive to process the entire function. This poses challenges for machine learning-based type inference tools, particularly those using NLP models.

NLP models face inherent limitations with long sequences. Traditional RNNs suffer from vanishing gradients [115], making it difficult to capture long-range dependencies.

While transformers handle longer contexts better, their attention mechanism has quadratic time and space complexity with respect to input length [81]. As a result, several tools use custom methods to extract compact, meaningful context from functions.

Program Window

Some tools limit the context window around the target instruction to avoid long input sequences:

- **CATI** [38] and **SnowWhite** [40] both define the context of a variable-related instruction by selecting a fixed-size window of 10 instructions before and 10 after the target, resulting in a total context window of 21 instructions.
- **Stride** [16] bases its type prediction on contextual similarity. It assumes that variables with similar surrounding code (i.e., n-grams of tokens) are likely to be of the same type. By comparing n-gram patterns, it infers types based on previously seen, similar contexts.

Program Slicing

Program slicing is a static analysis technique used to extract only the parts of code relevant to a particular variable or instruction by analyzing data and control flow dependencies [99, 100]. This helps reduce the input size while retaining the essential context. **TIARA** [43] uses *forward slicing* to include all instructions influenced by the target instruction. SeeType [97] performs their custom program slicing on each target instruction to find the relevant context and uses that to handle longer functions. NLP models have a maximum token limit, and often large functions can exceed the limit. To handle the larger functions, this technique is useful, and it helps the model to focus on only the interesting part of the program for the target task. WasmHint [114] also uses program slicing to reduce its input sequence to only the instructions related to the parameters of the functions.

Table 2.4: Languages Supported by Type Inference Tools

Tool	C	C++	WebAssembly
BinSub [10]	✓		
Escalada et al. [18]	✓		
BITY [37]	✓		
CATI [38]	✓		
StateFormer [1]	✓		
DIRTY [17]	✓		
DEBIN [15]	✓		
EKLAVYA [39]	✓		
OSPREY [34]	✓		
TIE [33]	✓		
Howard [26]	✓		
DRAGON [45]	✓		
TRex [11]	✓		
ByteTR [46]	✓		
IDIOMS [42]	✓		
SeeType [97]	✓		
DSIBin [27]	✓	✓	
Stride [16]	✓	✓	
Retypd [9]	✓	✓	
TypeSqueezer [6]	✓	✓	
RESYM [41]	✓	✓	
Manta [12]	✓	✓	
TIARA [43]		✓	
ObjDigger [31]		✓	
OOAnalyzer [32]		✓	
SnowWhite [40]			✓
WasmHint [114]			✓

Variable-Centered Graphs with k-hop Context

DRAGON [45] takes a graph-based approach to context extraction. Instead of a fixed instruction window or slicing-based method, it builds a custom variable-specific graph from the decompiled abstract syntax tree (AST). For each variable, it identifies all references within the AST and collects their *k-hop neighborhoods*—that is, all nodes within *k* edges of each reference node. These local subgraphs are then merged into a single *variable graph* representing the variable's usage context. WasmHint [114]

2.4.4 Languages Supported by the Tools

Most type recovery tools primarily target executables compiled from C or C++ programs, with a few supporting other languages like Rust or Go via WebAssembly. Presenting this information in a table allows a concise view of language coverage across tools.

As shown in Table 2.4, the majority of tools focus on C, with several extending support to C++. Tools like TIARA [43], ObjDigger [31], and OOAnalyzer [32] are specifically tailored for C++ binaries, while SnowWhite [40] and WasmHint [114] targets WebAssembly binaries that may originate from C, C++ or Rust. Among them SnowWhite focuses on C and C++ WebAssemblies while WasmHint focuses on Rust WebAssemblies.

Table 2.5: Categorization of Type Analysis Tools by Functionality

Category	Tools
Primitive Type Prediction	Binsub [10], TYGR [44], Escalada et al. [18], Bity [37], Cati [38], Stateformer [1], DEBIN [15], Eklavya [39], TypeMiner [36], DRAGON [45], DIRTY [17], RESYM [41], ByteTR [46], SeeType [97], WasmHint [114], Manta [12]
Structure Identification	Tiara [43], OOAnalyzer [32], DSIBIN [27], Mempick [29], DDT [28], DIRTY [17]
Structure/Class Reconstruction	Retypd [9], Osprey [34], Rewards [24], ARTISTE [30], TIE [33], OOAnalyzer [32], Objdigger [31], Howard [26], TRex [11], RESYM [41]
Class Hierarchy Recovery	Lego [47] (dynamic), Objdigger [31] (static)
Variable Role Similarity	DynCompB [25]
Object Similarity	Laika [19]
Function Signature Recovery	Binsub [10], TypeSqueezer [6]

2.4.5 Type Coverage: From Primitive Types to Reconstructed Abstractions

Binary type analysis tools serve a range of purposes, covering diverse aspects of type inference. This subsection categorizes these tools based on their primary functionalities, such as analyzing primitive types, identifying object-oriented class hierarchies, and recognizing and reconstructing complex data structures. The goal is to provide a clear overview of how each tool contributes to the field of binary analysis.

Primitive Type Prediction: Primitive, or atomic, types represent the most basic data types in a program, including integers (e.g., signed, unsigned, short, long), floating-point

numbers, characters, booleans, pointers, and arrays. Accurately predicting these types in binary analysis is essential for understanding the low-level structure and operational semantics of a program. By identifying primitive types from binary code, analysts can infer data layouts and gain deeper insights into program behavior. Furthermore, primitive type inference forms the foundation for reconstructing higher-level abstractions, as these types serve as the building blocks of more complex data structures. Tools such as Binsub [10], TYGR [44], Escalada et al. [18], Bity [37], Cati [38], Stateformer [1], DEBIN [15], Eklavya [39], DRAGON [45], Manta [12] and TypeMiner [36] primarily focus on recovering primitive types.

Structure and Class Identification: Some tools aim to identify known data structures and classes present within binary executables. For example, Tiara [43] focuses on detecting common C++ classes, including those from the Standard Template Library (STL). Similarly, OOAnalyzer [32] identifies C++ classes and their associated methods. DSIBIN [27] specializes in recognizing frequently used data structures such as doubly and singly linked lists, skip lists, and nested structures. Mempick [29] is capable of detecting a variety of dynamic structures, including linked lists, binary trees, and n-ary trees. Additionally, both DDT [28] and DSIBIN [27] contribute to structure identification.

Structure and Class Reconstruction: Unlike identification, which matches observed data patterns to a predefined set of known structures, structure reconstruction focuses on recovering the internal layout and member variables of a structure—regardless of whether they have been previously defined. This form of analysis aims to infer the composition of novel or custom data structures by analyzing how memory is accessed and manipulated. Even in the absence of prior knowledge about a structure's type, reconstruction techniques can provide valuable insights into its organization and usage. Tools such as Retypd [9], Osprey [34], Rewards [24], ARTISTE [30], TIE [33], OOAnalyzer [32], Objdigger [31], TRex [11], RESYM [41] and Howard [26] fall into this category.

Others: Some tools provide complementary type-related information beyond primitive types and structure recovery. Lego performs dynamic analysis to recover class hierarchies, while Objdigger [31] recovers similar hierarchies through static analysis. DynCompB [25] predicts whether two variables serve the same high-level purpose based on their usage patterns. Laika [19] attempts to identify similar objects by comparing their behavioral and memory characteristics. Additionally, tools like Binsub [10] and TypeSqueezer [6] focus on recovering function signatures, which can aid in understanding calling conventions and interface reconstruction.

Table 2.5 summarizes the categorization of these tools based on the type inference functionalities described above.

2.4.6 Publication Landscape and Evolution of the Field

Table 2.6 lists all 50 tools chronologically with their publication year, venue, and open-source status. Early tools appeared at systems and architecture venues (OSDI, MICRO, CAV); from 2017 onward, security venues (CCS, USENIX Security, IEEE S&P) dominate, with PL and software-engineering venues (PLDI, ESEC/FSE, ICPC) increasing after 2020 as learning-based methods were adopted. Open-source availability is inconsistent across all periods and is discussed further in Section 2.6.

Figure 2.1 plots the tools on a timeline colour-coded by method category. Three phases are discernible. **Phase 1 (1999–2013):** dynamic analysis dominates; runtime observation was the primary approach prior to mature IR lifters and large binary corpora. **Phase 2 (2011–2018):** static and formal methods emerge—constraint solvers (TIE [33], Retypd [9]) and logic-based systems (OOAnalyzer [32]) overlap with late dynamic work and early classical ML. **Phase 3 (2017–present):** learning-based methods predominate, progressing from classical ML to NLP/sequence models, GNNs, and LLM-assisted hybrid pipelines.

Table 2.6: Publication year, venue, and open-source availability of the 50 surveyed tools. Tools are sorted chronologically. “Open Source” values: ✓= publicly available source; × = closed / not released; – = not applicable (RE platform or software tool distributed as binary).

Tool	Year	Venue	Open Source
Dynamic Analysis			
DynCompB [25]	2006	ISSTA	✓
Laika [19]	2008	OSDI	×
Dc [48]	2008	WCRE	×
DDT [28]	2009	MICRO	×
REWARDS [24]	2010	Annual Info. Sec. Symp.	×
Howard [26]	2011	NDSS	×
ARTISTE [30]	2012	Tech. Report (IMDEA)	×
MemPick [29]	2013	WCRE	×
LEGO [47]	2014	CC	×
DSIBin [27]	2017	ASE	✓
TypeSqueezer [6]	2023	CCS	✓
Static / Formal Methods			
Mycroft [13]	1999	ESOP	×
TIE [33]	2011	NDSS	×
BAP [63]	2011	CAV	✓
OBJDIGGER [31]	2014	PPREW	✓
angr [23]	2016	IEEE S&P	✓
Retypd [9]	2016	PLDI	✓
OOAnalyzer [32]	2018	CCS	✓

Continued on next page…

Table 2.6 (continued)

Tool	Year	Venue	Open Source
NTFUZZ [35]	2021	IEEE S&P	✓
OSPREY [34]	2021	IEEE S&P	×
BinSub [10]	2024	SAS	✓
Manta [12]	2024	ASPLOS	×
TRex [11]	2025	USENIX Security	✓
EmCee [62]	2025	SURE	✓
Reverse Engineering Platforms			
IDA Pro [3]	2011	Book (No Starch Press)	×
Radare2 [71]	2017	Book / Open Source	✓
RetDec [14]	2017	Botconf	✓
Ghidra [2]	2020	Book (No Starch Press)	✓
Binary Ninja [4]	2025	Commercial Tool	×
Statistical / Classical Machine Learning			
DEBIN [15]	2018	CCS	✓
BITY [37]	2019	IEEE Trans. Reliability	×
TypeMiner [36]	2019	DIMVA	×
Escalada et al. [18]	2021	arXiv	✓
Stride [16]	2024	arXiv	✓
NLP / Sequence Models			
EKLAVYA [39]	2017	USENIX Security	✓
CATI [38]	2020	DSN	×
StateFormer [1]	2021	ESEC/FSE	✓

Continued on next page…

Table 2.6 (continued)

Tool	Year	Venue	Open Source
SnowWhite [40]	2022	PLDI	✓
DIRTY [17]	2022	USENIX Security	✓
Cao & Leach [92]	2023	ICPC	–
SeeType [97]	2025	IEEE Access	×
CATI++ [90]	2025	The Computer Journal	×
YALI [101]	2026	ICISSP	×
Graph Neural Networks			
TIARA [43]	2022	CGO	✓
TYGR [44]	2024	USENIX Security	✓
DRAGON [45]	2025	BAR	✓
ByteTR [46]	2025	arXiv	✓
Hybrid / LLM-Assisted			
ReSym [41]	2024	CCS	✓
IDIOMS [42]	2025	arXiv	✓
TypeForge [8]	2025	IEEE S&P	✓
HyRES [7]	2025	ACM TOSEM	✓
WasmHint [114]	2025	FSE	✓

2.5 Evaluation Methodologies in Binary Type Inference Tools

Different type analysis tools produce distinct forms of type information, each requiring tailored evaluation strategies. For instance, metrics commonly used for assessing primitive-type prediction—such as accuracy or precision—are not directly applicable to tools that

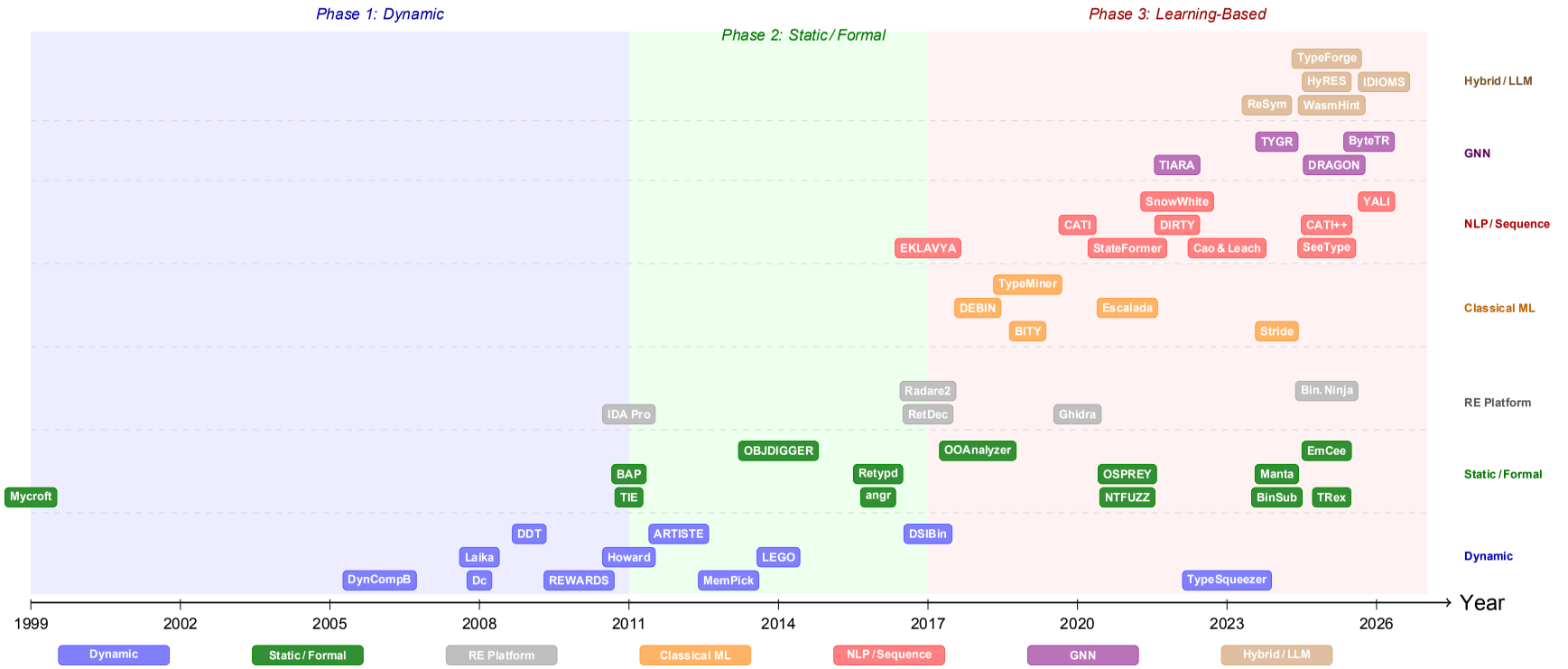


Figure 2.1: Evolution of binary type inference tools from 1999 to 2026. Each node represents one tool at its publication year, colour-coded by method category (right-side labels; see legend). Background shading marks the three phases: **Phase 1** (blue, 1999–2013) dynamic analysis; **Phase 2** (green, 2011–2018) static and formal methods; **Phase 3** (red, 2017–present) learning-based methods.

recover complex structures or class definitions. In this section, we first highlight empirical findings from ByteTR [46]. Later we briefly discuss the various evaluation methodologies used by type inference tools, highlighting how they align with the specific nature of the type abstraction being recovered.

2.5.1 Empirical Findings of ByteTR's Type Recovery Study

Li et al. [46] performed an empirical study on different properties of type usage over a large set of binaries. They utilized the TYDA [44] dataset, which includes 163,643 binary programs across four architectures and four compiler optimization options. They studied the unique patterns that characterize types and variables in binary code yielding five key insights.

- **Finding 1:** Primitive types are the core of the type system, appearing with significantly higher frequency compared to user-defined types.
- **Finding 2:** Only a few types are very frequent, while the rest occur very rarely, showing an imbalance in type usage frequency.
- **Finding 3:** Only a few key members are frequently accessed during a structure variable's life cycle, while others are relatively unused.
- **Finding 4:** Variable propagation across function boundaries is very common. Nearly half of the variables are propagated as arguments via function calls, and close to 60% of functions contain such propagated variables.
- **Finding 5:** Compiler optimization transforms the storage pattern of variables from stack-based to register-based.

2.5.2 Perfect Match Accuracy:

Perfect match accuracy evaluates a tool's predictions based on exact alignment with the ground truth. Under this metric, a prediction is considered correct only if it matches the expected type or label precisely, without partial credit. This strict criterion highlights the importance of exactness in type recovery, particularly for tasks where even minor deviations can lead to significant misinterpretations of program behavior. Many type inference tools adopt variants of this metric—such as accuracy, precision, recall, and F1-score—to evaluate their performance. Tools including TYGR [44], StateFormer [1], SeeType [97],WasmHint [114],Tiara [43], Rewards [24], Cati [38], DIRTY [17], DEBIN [15], EKLAVYA [39], TypeMiner [36], and OSPREY [34] have reported results using this approach.

2.5.3 Fine-grained Accuracy:

Fine-grained accuracy measures the degree of similarity between predicted types and the ground truth, allowing partial credit for near-correct predictions. Unlike perfect match accuracy, which requires exact matching, this approach allows minor discrepancies and captures the semantic closeness of types. For example, if a tool predicts a `short int` as an `int`, perfect match accuracy would consider it entirely incorrect, whereas a fine-grained metric would acknowledge the partial correctness, reflecting that the general type category was correctly inferred. These metrics are often customized for specific problem settings and can provide more nuanced insights into a tool's predictive capabilities, especially in complex or ambiguous typing scenarios. Several fine-grained accuracy metrics that have been employed by recent type inference tools include the following.

Type Distance Metric: The Type Distance Metric evaluates the accuracy of type inference tools by measuring the distance between predicted types and ground-truth types within a predefined type lattice. In this framework, types are organized hierarchically or

semantically, and the distance quantifies how far the inferred type deviates from the correct one. A smaller distance indicates a closer, and thus more acceptable, prediction. This approach reflects the intuition that while exact matches are ideal, near-correct predictions still carry meaningful semantic information. Several tools—including TIE [33], BinSub [10], Retypd [9], BITY [37], and ARTISTE [30]—employ variants of this metric to assess their performance. Figure 2.2 illustrates an example type lattice used by TIE [33], highlighting how types relate to each other in a type lattice. This method provides a more graded evaluation by recognizing that proximity to the correct type retains analytical value, even when exact recovery is not achieved.

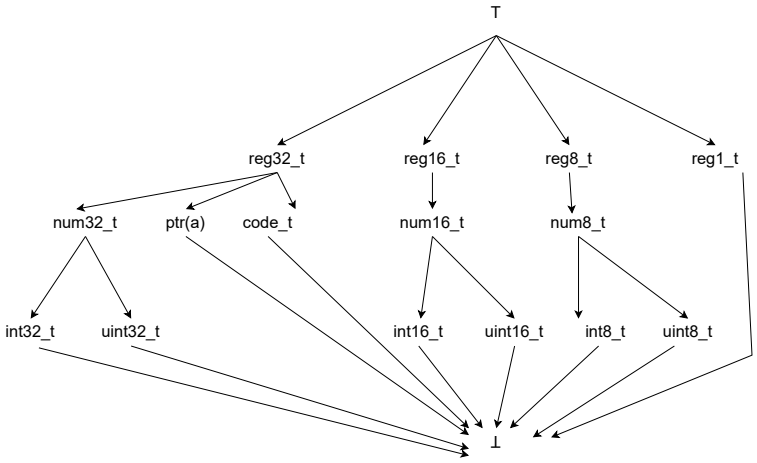


Figure 2.2: Type Distance Metric in Type Lattice: This figure illustrates how types are represented in the type lattice of TIE [33], showing the accuracy of type inference systems by measuring the distance between inferred and ground-truth types. The closer the predicted type is to the actual type, the more accurate the inference.

Type Prefix Score: The Type Prefix Score measures the number of consecutive type tokens from the beginning of a type sequence that are correctly predicted before the first incorrect token. This score provides a way to measure the precision of type predictions, especially in contexts where knowing the most significant part of a type (e.g., being a pointer type versus an integer) can be more important than getting every detail correct. This metric rewards predictions that are correct at the beginning of the sequence even if they diverge later. SnowWhite [40] uses this metric to measure their accuracy.

Decision Tree: TRex [11] introduces a custom decision tree to evaluate type recovery accuracy. The scoring process assigns incremental credit for each semantically compatible property between the predicted and ground-truth types. For example, the evaluator checks whether the inferred type matches in pointer status and indirection levels, whether it is a struct or primitive, and whether the signedness aligns. Each compatible property contributes to the cumulative score, allowing partial credit for types that are close but not identical (e.g., int64_t instead of uint64_t). This compatibility-driven metric reflects the practical usability of inferred types in reverse engineering workflows.

Edit distance: Edit distance is a metric used to evaluate the structural accuracy of recovered C++ class abstractions. It quantifies the number of modifications required to transform the set of classes inferred by a tool into the ground-truth class layout. These modifications may include moving methods between classes, adding missing methods, removing extraneous ones, or merging and splitting class definitions. This metric provides a meaningful way to assess how closely the reconstructed class hierarchy aligns with the original design. OOAnalyzer [32] adopts this metric to evaluate its effectiveness in recovering C++ class structures.

2.5.4 Per-Tool Evaluation Summary

Table 2.7 maps each tool to four evaluation dimensions: ground-truth construction, benchmark dataset, baselines, and evaluation metric. Three points stand out. First, no shared benchmark exists—most tools strip private corpora of DWARF-annotated open-source binaries, so cross-tool comparison is infeasible. Second, baselines are sparse and inconsistent, leaving published accuracy figures incommensurable. Third, many tools, especially early dynamic systems and several recent hybrid pipelines, report no quantitative type-recovery evaluation at all—a reproducibility gap revisited in Section 2.6.

Table 2.7: Per-tool evaluation summary. “GT” = ground truth source; “Dataset” = named benchmark or corpus used; “Baselines” = tools explicitly compared against; “Metric” = primary evaluation metric reported. Cells marked — indicate information not stated in the surveyed publication.

Tool	GT Source	Dataset	Baselines	Metric
Dynamic Analysis				
Dyn-CompB [25]	Source code	6 benign programs	Lackwit [116]	Quantitative (coarse)
Laika [19]	Source / debug symbols	10 benign + 27 binaries for malware detection	ClamAV [117]	Quantitative (coarse)
Dc [48]	None	-	None	None
DDT [28]	Source code	11 benign programs	None	Quantitative (coarse)
RE-WARDS [24]	Source / debug symbols	Coreutils, 10 benign	None	Quantitative (coarse)
Howard [26]	Source / debug symbols	9 benign programs	None	Quantitative (coarse)
ARTISTE [30]	Source / debug symbols	5 benign programs	RE-WARDS [24]	Fine-grained (distance / conservativeness)
Mem-Pick [29]	Source / debug symbols	26 benign programs	None	Quantitative (coarse)

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
LEGO [47]	Source / debug symbols	10 benign programs	None	Fine-grained (class hierarchy set comparison)
DSIBin [27]	DWARF	30 programs	Howard [26]	Quantitative (coarse)
TypeSqueezer [6]	Source code / debug info	SPEC CPU2006 + real-world applications	TypeArmor [113]	—
Static / Formal Methods				
TIE [33]	Source / debug symbols	Coreutils, 87 binaries	REWARDS [24], HexRays decompiler	Fine-grained (distance + conservativeness)
Retypd [9]	DWARF	coreutils and SPEC2006	TIE [33], REWARDS [24]	Type Distance
OOAnalyzer [32]	PDB	18 benign and 9 malware	ObjDigger [31]	Edit Distance

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
OBJDIG-GER [31]	MSVC class-layout dumps + PDB + Source Code	5 open-source MSVC packages + 1 malware sample (585 functions)	None	Recall, Precision
OS-PREY [34]	DWARF	GNU Coreutils, Howard Benchmark [26]	IDA [3], Howard [26] Ghidra [2]	Accuracy, F1
BinSub [10]	DWARF	ALLSTAR	Retypd [9]	Type Distance
Manta [12]	DWARF	14 open-source projects plus 9 IoT firmware samples.	Dirty [17], Ghidra [2], RetDec [14], Retyped	Precision and Recall
TRex [11]	DWARF	GNU Coreutils 9.3 and SPEC CPU 2006	Ghidra, ReSym [41]	F1, Decision Tree (custom)
EmCee [62]	Facts from EmCee Model Compiler	AFL++ fuzzer-generated test cases for EmCee	OOAnalyzer [32]	Recall
Classical / Statistical Machine Learning				
BITY [37]	DWARF	Coreutils, diffutils, findutils	Ida Pro [3], EKLAVYA [39]	F-1, Type Distance

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
TypeMiner [36]	DWARF	14 open-source C projects	—	Accuracy, F1
DEBIN [15]	DWARF	9000 ELF binaries	—	Accuracy, F1
Escalada et al. [18]	DWARF	—	Ida Pro [3], RetDec [14]	Accuracy, F1
Stride [16]	DWARF	DIRT [17], DIRE [118]	—	—
NLP / Sequence Models				
EKLAVYA [39]	DWARF	8 Linux packages	—	Accuracy, F1
CATI [38]	DWARF	2141 GCC-compiled binaries from open-source projects (coreutils, binutils, nginx, etc.)	DEBIN [15], Ida Pro	Accuracy, F1
StateFormer [1]	DWARF	—	EKLAVYA [39]	Accuracy, F1

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
SnowWhite [40]	Wasm DWARF	Custom Wasm corpus(4,081 Ubuntu C/C++ packages)	—	Accuracy and type prefix score.
DIRTY [17]	DWARF	DIRT [17]	OSPREY [34], Ida Pro [3]	Accuracy, F1
Cao & Leach [92]	DWARF	DIRT [17]	DIRTY [17]	Accuracy
SeeType [97]	DWARF, Source Code	500k binaries	Ghidra [2], State-Former [1]	Accuracy, F1
CATI++ [90]	DWARF	—	CATI [38]	Accuracy, F1
YALI [101]	DWARF	BINKIT (33,840 binaries, 51 GNU projects, 4 architectures)	Ghidra, Retypd, EKLAVYA, Dirty	Accuracy, F1
Graph Neural Networks				
TIARA [43]	DWARF	8 real-world COTS C++ binaries (clang, cmake, bit-coind, etc.)	—	Accuracy, F1

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
TYGR [44]	DWARF	TYDA [44]	OSPREY [34], DIRTY [17], State-Former [1], DEBIN [15]	Accuracy, F1
DRAGON [45]	DWARF	TyDaMIN-D (sampled from TyDa) [44] and ReSym's training set [41]	TYGR [44], ReSym [41], Ghidra	Accuracy, F1
ByteTR [46]	DWARF	TYDA [44]	DIRTY [17], State-Former [1], TYGR [44], Ghidra, Ida Pro	F1
Hybrid / LLM-Assisted				
ReSym [41]	DWARF	7,416 GitHub C/C++ projects	DIRTY, OS-PREY	Accuracy, F1
IDIOMS [42]	DWARF	REALTYPE [42]	llm4decompile [119]	Recursive structural exact-match for UDTs

Continued on next page…

Table 2.7 (continued)

Tool	GT Source	Dataset	Baselines	Metric
TypeForge [8]	Debug information	OSPREY's coreutils and Howard dataset	DIRTY, OSPREY, TYGR, ReSym	Precision/recall/F1 for type identification, layout recovery, relationship recovery
HyRES [7]	Debug information	OSPREY's coreutils, wget, grep, gzip, lighttpd	OSPREY, DIRTY, ReSym	F1, cosine similarity
WasmHint [114]	77,208 Wasm functions from 369 GitHub Rust contract projects	EKLAVYA, SnowWhite [40]	Accuracy	

2.6 Discussion

In this section, we discuss unresolved issues and offer perspectives on potential avenues for future research.

Challenges of Dynamically Typed Languages: Languages with dynamic typing, such as Python and JavaScript, present unique challenges for type inference tools. Their flexibility—enabling on-the-fly type changes, late binding, and extensive polymorphism—makes static analysis significantly more complex. Unlike statically typed languages, these do not

enforce explicit type declarations, which limits the effectiveness of traditional inference techniques. To the best of our knowledge, existing type inference tools are not specifically designed to handle binaries generated from dynamically typed languages. As programming languages continue to evolve and incorporate advanced features such as generics and metaprogramming, future type inference tools will need to adapt accordingly to remain effective.

Scalability: Scalability remains a fundamental challenge across nearly all binary analysis tasks. As the size and complexity of binaries grow, the computational resources and analysis time required tend to increase exponentially. To address this, some tools adopt strategies that focus on analyzing selected regions or functions rather than entire programs. This method is often effective for some binary analysis tasks but scalability continues to significantly limit the practical applicability of type inference tools.

User Experience and Tool Integration: With the exception of general-purpose reverse engineering platforms such as IDA Pro [3], Ghidra [2], and Angr [23], most type inference tools lack a user-friendly interface for practical use. In many cases, these tools are not publicly released, and even when open-sourced, they often suffer from minimal documentation and limited usability. This creates a barrier for adoption by practitioners and researchers alike. One promising direction for improving accessibility is integrating these tools as plugins or extensions within widely used reverse engineering frameworks like IDA Pro or Ghidra. Such integration enables easier deployment and interaction within familiar environments.

Lack of Standardization in Output Formats Complicates Tool Comparison: One major challenge in evaluating and comparing type inference tools is the absence of a standardized output format. Each tool typically produces results tailored to its internal methodology, differing in granularity, representation, and the types of information conveyed. This variability makes it difficult to perform fair, direct comparisons across tools.

2.7 Conclusion

Type inference from stripped binaries is a foundational challenge in reverse engineering, with broad applications in decompilation, vulnerability detection, binary rewriting, and software comprehension. This survey reviewed 50 type inference tools, categorizing them by the types they recover, the methodologies they employ, and the evaluation metrics they use. We observed a clear shift from early dynamic analysis approaches to modern static and learning-based techniques, each offering different strengths and trade-offs. However, several challenges remain unresolved, including limited scalability, lack of support for dynamically typed languages, non-standardized output formats, and poor integration into widely used reverse engineering workflows. Additionally, we note the need for more consistent evaluation frameworks to enable fair comparison among tools. Looking ahead, future work should focus on improving the robustness, usability, and interoperability of type inference systems. It should also explore deeper semantic reasoning and tighter integration with emerging binary analysis techniques.

Chapter 3

From Bytes to Types : Enhancing Type Prediction in C Executables with Transformer-based Binary Analysis Tool SeeType

Problem Statement. Assembly instructions rely on generic registers and memory locations, which lack inherent type indications. This missing information makes it difficult to decipher the meaning of the executable's operations on a semantic level understandable by human experts. For instance, in the case of the x86 architecture, if we are dealing with the register `EAX`, we may infer that the register has some data type that is 32-bits. Yet this would be a naive approach as there are several high-level C types such as `float`, `signed long int`, or `pointers` which share the same size [33]. However, if we look beyond that single instruction towards where the data in `EAX` came from and how it is used, we can achieve a more accurate estimation as to the data type. The primary problem we are trying to solve is source-level type information recovery from binary code. For our experiment, we are considering the prediction of C types from binaries compiled from C programs. We target the primitive (e.g. `int, float`), aggregate (e.g. `struct, array`), and `pointer` types. Our method attempts to predict any type, given sufficient examples of that type in our dataset. Our current dataset includes 52 different C types, and we measure the performance on the most common 44 of these as presented in Table 3.3. The input to our type prediction task is a sequence of instructions and a marker to the specific instruction we are interested in. This is a classification task where the types are the classes that the model needs to assign to a specific instruction in a program.

Challenges and Our Solution. One of the barriers to training a Deep Neural Network (DNN) is that a substantial amount of labeled examples are necessary to make robust predictions. To address this, we gathered a large number of programs from open-source repositories (totaling 100.2k repositories, including 559k executable files). While training the models, we faced multiple problems that the previous type of predicting tools did not address well. For example, instructions have a lot of numbers, and natural language models are not good at processing them [103]. So we conducted experiments to find a better method to tune the model to process numbers. We plan to do an extensive experiment on this problem in our future work as this is not our primary problem.

Instructions are not like human language and do not follow a consecutive order. For example, two neighboring instructions can be unrelated and two distant instructions can be closest to each other in relation. If we process the instructions as they are located originally in the file, it would not be optimal. So, we used program slicing methods to formulate a method to calculate the context of an instruction and use that as the model input. This method helps us to process functions that have a lot of instructions and some instructions need to be rejected from the model input. We also devised a new approach to pretrain our natural language model to adapt to machine language better.

Contributions. The contributions of this work can be summarized as follows:

(1) We developed a novel way of using source code type information to assist in labeling type information of machine instructions.

(2) We created a diverse and large corpus of compiled binaries and labeled them with type information to train a DNN model.

(3) We designed and evaluated a new tokenization technique to handle numerical values more effectively.

(4) Our model outperformed two SOTA tools, StateFormer [1] and Ghidra [2].

(5) We devised and evaluated a technique to analyze functions with an instruction count greater than the capability of the DNN model.

(6) We designed and evaluated a new pre-training technique that increases the performance of the DNN model.

We plan to release the code and datasets used for this work upon acceptance for publication.

3.1 Background

3.1.1 The Complexity of Executables

One of the biggest challenges in reverse engineering is trying to understand the semantics of a program when most of the abstractions created by the original programmer are lost in the compilation process. The most basic starting points for analyses of an executable are code discovery, Control Flow Graph (CFG) construction, and Data Flow Graph (DFG) construction. Each of these rely on and have interdependence with the recovery of a correct disassembly. There has been significant research on this task [120, 121], but the underlying problem of generating a correct and complete disassembly is undecidable [122] and modern tools rely on a variety of heuristics. Our goal is to recover the abstract source-level types from executables. The process will have some inherent noise, which we ignore. For example, we assume that the disassembly and DFG are correct. This is the case for our ground truth set, because of the generation process used, although it may not be the case for binary executables in general.

3.1.2 Classification with BERT

Our experiment involves analyzing a set of sequential instructions and classifying a target instruction into a source-level type. BERT (Bidirectional Encoder Representations from Transformers) [123] is a popular Transformer [81] model for text classification. The following steps describe how a Transformer classifies a sequence.

Input Encoding

The input sequence undergoes tokenization, breaking it down into smaller units like words or subwords, which in the case of machine instructions are the elementary parts of the instructions such as `mov`, `add`, `+`, `[` etc. Each token is subsequently converted into an embedding vector, effectively capturing the semantic meaning of the token. These embeddings encapsulate the contextual information of each word within the text. An advantage of using encoding instead of actual tokens is that tokens with similar meanings should have similar embeddings which helps the model to generalize. We have trained our custom tokenizer so that the embeddings generated for instruction tokens work more effectively than the off-the-shelf BERT tokenizer trained with natural language data.

Positional Encoding

Along with token embeddings, BERT uses positional embeddings and segment embeddings for each token, which incorporate regional information into the input. Positional embeddings play a crucial role in Transformer-based architectures for preserving token order, as their absence would result in a bag-of-words representation [124]. This step is crucial as Transformers do not inherently understand the sequential order of tokens.

Self-attention

The attention mechanism, first introduced by Vaswani et.al. [81], helps Transformers focus on relevant parts of the input when making predictions. It assigns different weights to different parts of the input, indicating their importance. This allows the model to “pay attention” to the most relevant information while ignoring irrelevant details. This is vital for our task as some parts of the program are more important than others when predicting the type of information of a particular instruction.

BERT uses a self-attention mechanism. In simple terms, when BERT is analyzing a sentence, self-attention enables the model processing a word in the sentence to look at other words to know which words contribute to the current word. So, the model evaluates the sentence by looking at all the tokens to figure out how to represent each token.

Transformer Encoder

While the original transformer architecture has two main components, an encoder and a decoder, BERT only uses the encoder. The encoder has several repeated layers of self-attention and feedforward neural networks. The augmented encodings from the tokens are passed through the encoder layer. The output vector from the encoder then can be used for various downstream tasks such as classifying types.

Classification and Learning

The final output from the encoder obtained after processing the input sequence is commonly pooled or concatenated to create a fixed-size representation for the entire input. This aggregated representation is then fed into a classification layer, such as a softmax [125] layer, to predict the desired class label. The predicted labels are then compared with the ground truth labels and loss is calculated, representing how good or bad the prediction was. The loss is then backpropagated to update the model parameters.

3.2 SeeType

We treat our type prediction problem as a classification problem and use BERT, a Transformer machine learning model, as a tool to approach the problem. We first pre-process the disassembled instructions before feeding them to BERT. In the case of large functions with an instruction count higher than the maximum limit of BERT, we find the partial part of the function relevant to the target instruction using data dependency analysis and program

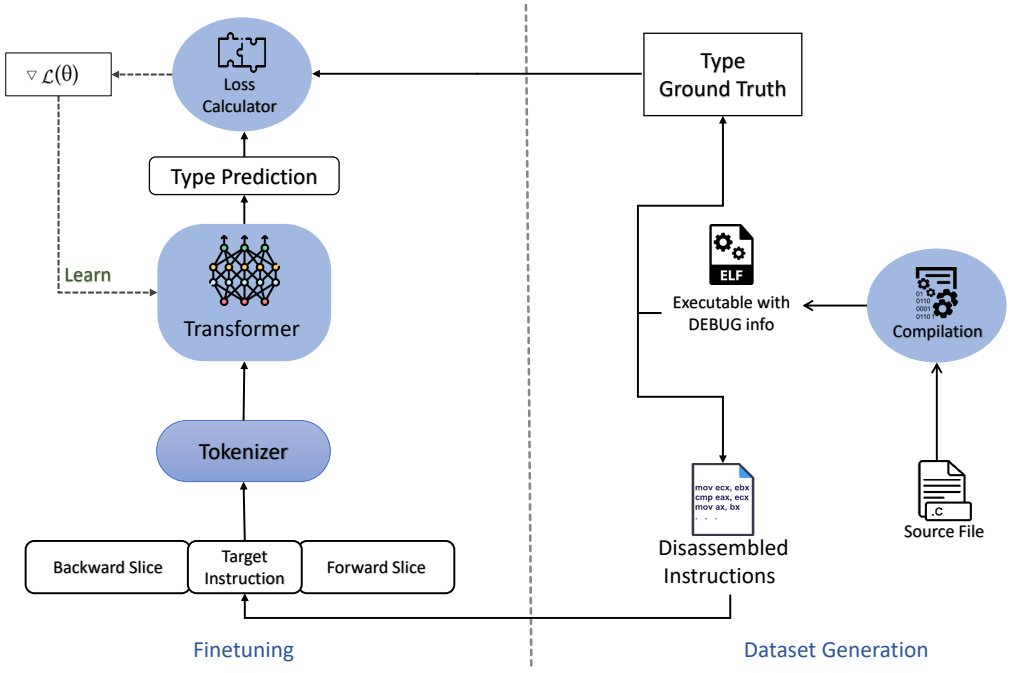


Figure 3.1: System design of SeeType and Dataset pre-processing.

slicing. We explain the program slicing process in detail in Section 3.4.1. We then train a tokenizer suitable for tokenizing machine instructions. Tokenizing instructions differs from tokenizing natural languages and we discuss our approach in Section 3.6. Next, we pre-train the BERT model with Masked Language Modeling and Data Dependency Modeling to make the BERT model more suitable for machine instructions, as described in 3.5. After pre-training, we transfer the learned knowledge during pre-training for our desired task by fine-tuning the BERT model on the type prediction task, discussed in 3.7. The fine-tuning process is shown in Figure 3.1. The right half of the figure shows how an executable is compiled from a source file with debug information. The executable is then disassembled which is used for generating the input of the model. At the same time, Ground truth is also generated with the help of debugging information which is used to train and validate the model. The left side of the figure shows how SeeType works. Initially, the input sequence is produced from the disassembly. Only the relevant instructions

to the target instructions are included in the input in case the original function disassembly is larger than the maximum length allowed for the model. Then the input sequence is tokenized and fed to the transformer model. Analyzing the input, the model makes a prediction about the type which is then compared with the ground truth type. During training, the model parameters are updated based on the correctness of the prediction. After the training is done, the correctness of the model is evaluated on a test dataset.

3.3 Challenge Of Building A Type Data Set

Training a large language Transformer model requires much more labeled data than other traditional approaches. To generate this data, we created a fully automatic process that compiled a multitude of programs and labeled the disassembled instructions of the programs with their respective types. The challenges of making this kind of dataset are discussed below.

3.3.1 Magnitude and Diversity

A large dataset with diverse examples mirrors the various scenarios the model may face in real-world applications. This facilitates improved generalization, enhancing the model's robustness and capacity to handle novel and unseen data. The dataset needs to cover a wide range of program structures, functions, logic, implementation styles, and compiler optimizations to enable the model to learn robust representations and patterns.

We searched GitHub for suitable open-source code repositories that are mostly written in the C programming language. We found 100.2k repositories and cloned them. We then compiled all of the C files to build their respective 559k executables. During the compilation process, we turned on the debug information generation mode to create the ground truth set.

Figure 3.2 shows the frequency of ground truth samples recovered for types in our dataset and Figure 3.1 shows the dataset generation process.

3.3.2 Challenges With Variations of Optimization Level

The selection of an optimization level, spanning from minimal optimization (e.g. 'O0') to aggressive optimization (e.g. 'O3'), significantly shapes the characteristics of the resultant executables. At lower optimization levels, the compiler maintains the original structure of the source code as much as possible, resulting in larger, slower executables. Higher optimization levels employ advanced optimization techniques, such as loop unrolling, function inlining, and constant propagation, yielding more compact and faster code, and critically, using different instruction idioms to produce the final assembly. Recognizing the impact of these differences is crucial for any model analyzing binary programs. In our dataset creation process, each program was compiled with several optimization levels from the same source file.

We could collect fewer samples from higher optimization-level binaries. This is because it gets harder to label types for highly optimized binaries, even with DWARF information, as instructions no longer maintain as close a relationship to the original source language.

3.3.3 Comprehensive Type Coverage

The C programming language supports a wide range of data types, ranging from commonly used ones like `int` and `float` to more specialized types such as `long long int`, `long double`, `pointer` and `structure`. Despite their varying degrees of prevalence, each type holds significance in the semantics and other properties of a program. We have generated and trained our model on 41, 39, 50, and 39 fine-grained types for O0, O1, O2, and O3 respectively. We differentiate array types, which have not been explored by previous research [1, 36]. We have ignored some types such as `Union` and `Enum`. Enums, which

are symbolic names for integer values, are replaced by their corresponding integer values during compilation. Unions allow different data types to share the same memory location, and may not leave a recognizable footprint in the compiled instructions, as the compiler resolves their implementation during the compilation process. Both of these types act more as hints to the compiler, rather than fundamental types that are represented in the compiled binary.

The ground truth generation process also yielded different numbers of samples from different binaries from the same source but different optimization levels. Some type samples were simply absent in data generated for one optimization level but were present in another. So, as we are saying we are classifying more than 40 classes, the set of classes varies for different optimization-level experiments. In Figure 3.2 the Frequency of the samples per type represents the average frequency of the samples of 52 types. SeeType trained on fewer types because some types had insufficient samples for training and testing.

3.3.4 Challenges With Heterogeneity in Binaries

The binary output from the compilation of a specific C program can vary widely based on factors such as different compilers, compilation parameters, and architectures. Each compiler uses its custom optimization process and idioms to produce output assembly, producing very different binaries. Compiling for different instruction set architectures radically changes the binary code, with different underlying types, registers, and operations. For this experiment, we did not explore these differences, but as our dataset generation process is automatic and we have all the source code, we can generate binaries with all these variations for further experiments.

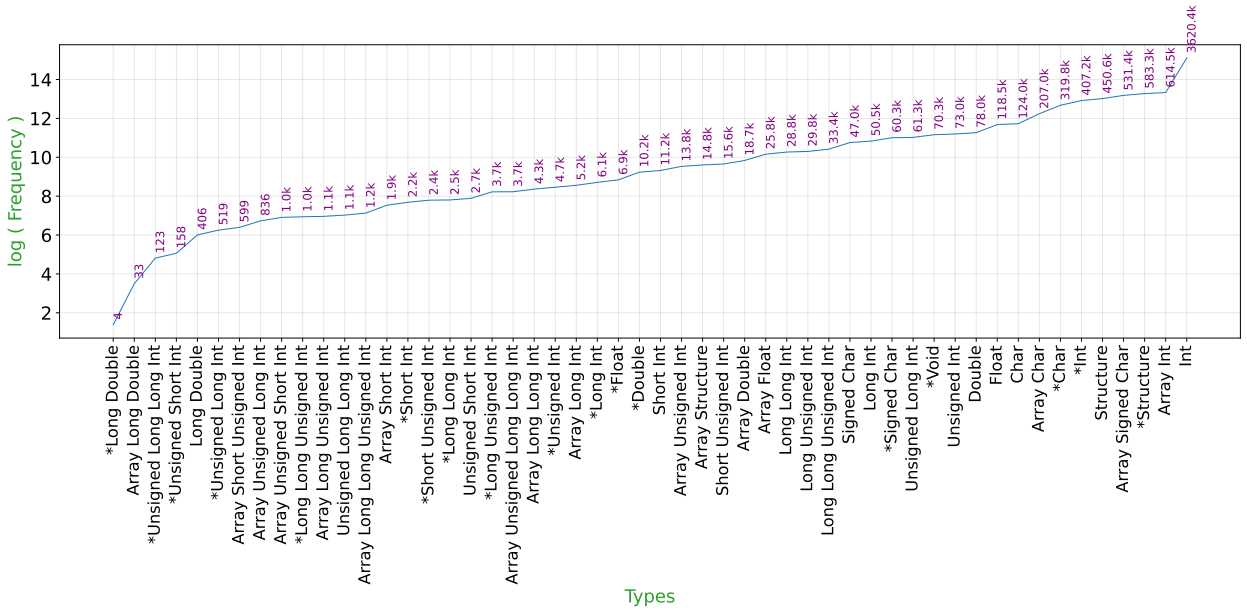


Figure 3.2: Frequencies of ground truth samples of different C types extracted from compiled binaries. The numbers on the curve represent the average number of samples per optimization level of that particular C type in our dataset.

3.4 Challenges of Longer Functions

There are certain challenges Transformer models face when analyzing binary programs with large functions due to their inherent complexity. Longer functions mean more instructions and more tokens the Transformer has to process. In addition, the control flow and data flow complexity often scale with function length. The full-attention mechanism in Transformers like BERT allows them to consider all positions in a sequence simultaneously, but this becomes computationally intensive for long sequences, leading to increased memory requirements and processing time [126]. Transformers also often struggle to capture long-range dependencies [127]. There has been significant research on how to handle longer sequences in NLP models such as GPT-2 [128], and Pegasus [129].

Type recovering tools such as BITY [37] and CATI [38] tried to utilize the context of the target instruction, instead of the whole function. BITY analyzed the dependency relation among instructions and selected a few instructions directly related to the target instruction as the input for their tool. However, they encode the instructions as a vector where the

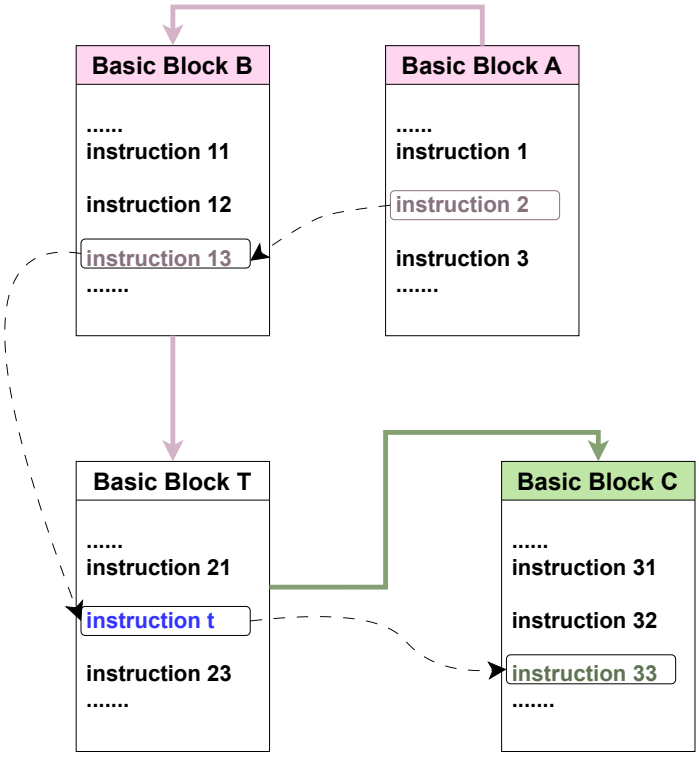


Figure 3.3: Basic Block T is dependent on Basic Block B because target instruction t in Basic Block T is dependent on instruction 13 from Basic Block B. Additionally, Basic Block A is dependent on Basic Block B because of the dependency relation between instruction 13 and instruction 2 from Basic Block B and A respectively. Similarly, Basic Block C is dependent on Basic Block T because instruction 33 from Basic Block C depends on the target instruction t from Basic Block T.

order of the instructions is lost. CATI used machine learning techniques for type prediction. They also tried to capture the context of the target instruction. They only considered the backward and forward 10 instructions to the target instruction as the context without considering the dependency relationship among instructions. Both of those efforts were for capturing the context of the target instruction, rather than an attempt to analyze large functions for type recovery. Besides Stride [16] uses n-grams before and after the target instruction for capturing the context. Tiara [43] uses only the forward slice for context.

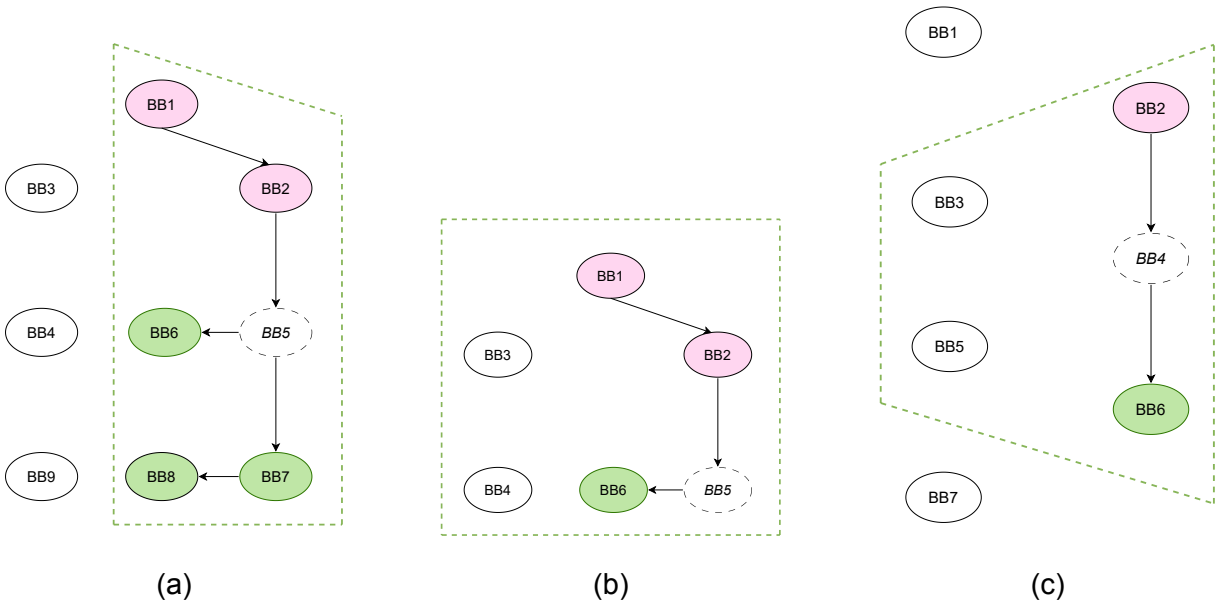


Figure 3.4: Three examples of selecting program context using program slicing. The nodes in the graph represent Basic Blocks (BB). The enclosed areas by the dashed lines represent the final model input. The green nodes are in forward slices and the pink nodes are in backward slices. In Figure (a), the target instruction is in BB5 and the context contains the backward and forward nodes. As the context size equals the model input length, the context is set to be the final input. In Figure (b), the function is small so the whole function is selected as the model input. In Figure (c), BB4 contains the target instruction. BB2 and BB6 are the only two nodes in the backward and forward slices respectively. However, as there is space left in the input, the BB3 and BB5 are included in the input because of their proximity to the target instruction.

3.4.1 Our Solution Using Program-Slicing

To avoid this limitation with large functions, previous related research that processes machine instructions with NLP models, such as StateFormer [1] discarded the longer functions that generated more instructions than the model can handle. We think this is an interesting area that deserves more attention as larger functions are very common. Despite BERT having a smaller capacity for sequence length handling, we decided to use it due to its performance and prevalence in the literature. This is acceptable because there will always be functions larger than the maximum limit of the largest model. Rather than trying to handle entire large functions, we decided to feed the Transformer only the portion of the function that is relevant for a particular instruction, the context. To do this, we used the idea of program-slicing [99]. This allowed us to handle both long sequences and distant dependencies. Program slicing is a disintegration process that discards parts of the program that are irrelevant to a particular computation process based on a criterion known as the slicing criterion. In our case, the criterion is the target instruction that we are trying to predict the type of. For longer functions, we try to discard the instructions irrelevant to the target instruction we are analyzing. Ideally, only the related instructions are included in the final input sequence, allowing us to handle functions of all sizes.

Program Slicing [100, 130]. Previous research such as [110, 131] used binary program slicing for analyzing binary programs. For our method, we are interested in both forward slicing and backward slicing. The backward slice contains all the instructions that affect the target instructions in any way. The forward slice includes all the instructions that are affected by the target instruction. Ideally, the final context for our problem should include all the instructions on which the target instruction depends on and all the instructions that depend on the target instruction.

Context Generation. For producing the context, we utilized the directed dataflow graph of a function. The nodes in the dataflow graph are instructions from the function and the edges are the dependency relation between the nodes. The nodes connected

to the target instruction constitute the primary context, which includes both forward and backward slices.

We wanted to preserve the basic block structures of the code in our model input. This means if an instruction is chosen to be in the context, we include the whole basic block containing that instruction. The reason behind this is there might be instructions in a basic block that do not have any dependency relation with the target instruction but might contain clues for the pattern-recognizing machine learning model. According to this scheme, the context should have all the instructions that have a dependency relation with the target instruction along with the complete basic blocks they are located. To achieve that, we generated another dependency graph where the basic blocks are the nodes. The nodes are connected with directed edges which are produced from the original dependency graph. We presented this procedure in Figure 3.3 with an example. The connected basic blocks in the dependency graph are the final context of the target instruction.

Finalizing Model Input. The maximum model input length is fixed. But, the length of the context can vary and we want to put as many instructions as possible in the input. There can be four cases. **Case 1.** The context can be equal to our model input. If we have that perfect size, we select all the instructions from those basic blocks in our final model input. See Figure 3.4 (a) for an example. **Case 2.** If all the instructions of the function fit our model input, then we do not need to do any analysis and the whole function can be used as the context, like in Figure 3.4 (b). **Case 3.** In case the context is smaller than the input limit, we add surplus unrelated basic blocks to the context given that the context length stays less or equal to the model input. To select unrelated basic blocks to include, we find the nearest basic blocks to the target instruction. An example of this can be found in Figure 3.4 (c). **Case 4.** In rare cases where the context size is greater than the maximum input limit, we follow a simple approach of selecting basic blocks from the context that are nearest to the target instruction.

3.5 Challenges of Using Models Specialised for Natural Languages with Assembly Language

LLM models are designed to work with natural languages such as English, so we devised our own pre-training method to make the model more optimized for assembly language rather than natural languages. Self-supervised pre-training has proven extremely valuable in preparing NLP Transformer models for a variety of tasks [132]. Using the inherent bidirectional context modeling in a Transformer architecture and providing the model an initial phase of unsupervised learning on a large corpus of unlabeled data allows the model to learn contextual embeddings. These embeddings encapsulate latent semantic structure, syntactic relationships, and nuanced patterns. When combined with fine-tuning task-specific labeled data, the model exhibits greatly improved performance.

Previous research on StateFormer [1] and PalmTree [133] and work by Ahn et. al. [134] have leveraged self-supervised training with binary instructions. For our task, we pre-train our model with two tasks, Masked Language Modeling (MLM) and Data Dependency Relation Modeling.

3.5.1 Pre-training SeeType with Masked Language Modeling

MLM, depicted in Figure 3.5, is frequently used for pre-training Transformers, where a subset of tokens in a given sentence is randomly masked, and the model is trained to predict the masked tokens based on the surrounding context. To predict the masked tokens, the model has to learn the syntax and semantic relationship between tokens within a relatively shorter range. In the example presented, we can see that one of the tokens from the instruction is masked. The model needs to predict the correct token which is `ptr`.

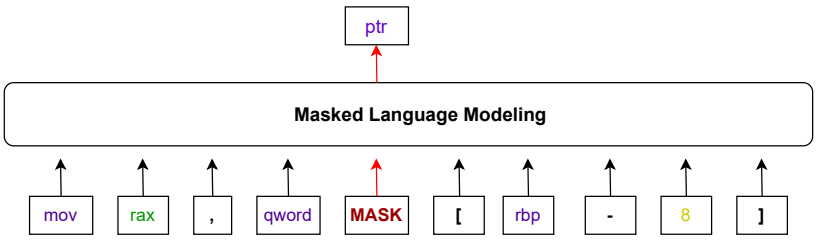


Figure 3.5: Masked Language Modeling

3.5.2 Pre-training SeeType with Data Dependency Modeling

The input of the model during training on data dependency is a sequence of instructions from a function with two particular instructions marked. The model predicts if there is any data dependency, direct or indirect, between that pair of instructions. Correctly predicting the data dependency of a pair of instructions within a function requires a deep understanding of the long-range dependencies and relationships among the instructions. Figure 3.6 shows a positive and negative example of data dependency modeling training data. In the positive example on the left, we can see that the instruction on `0x1201` uses the value of register `RAX` which was loaded by the instruction at `0x11fa`. We also find that the value loaded into the register `RAX` depends on the previous instruction at `0x11f6`. Thus the instruction at `0x1201` depends on `0x11f6`. To understand this kind of relationship the model needs to understand the operation of instructions.

3.6 Challenges of Tokenizing the Instructions

Recent ML-based systems such as TypeMiner [36] eliminate addresses and other numerical values from the instructions before tokenization. Transformers tailored for NLP tasks are not the best models for handling numerical values [103]. StateFormer [1] tries to mitigate the limitation by using Neural Arithmetic Units (NAU) [135].

For our application, however, the addresses and other numerical values are valuable because they often carry information about the control flow and the storage location of source-level variables [54]. They indicate relationships between instructions and are

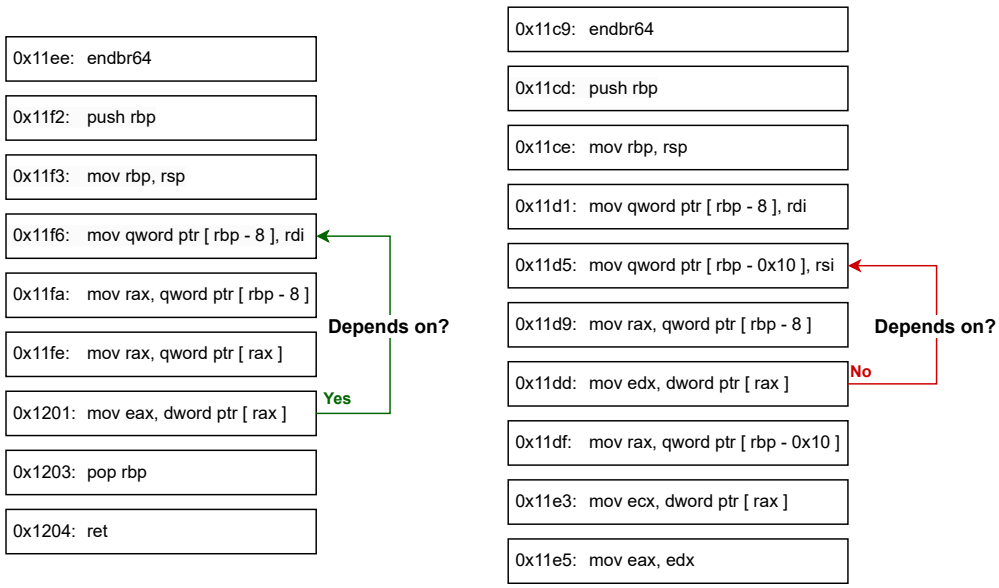


Figure 3.6: Examples of data dependency samples. The arrows indicate the data dependency relation between a pair of instructions.

necessary for reconstructing the control flow of the program. Therefore, we worked to find a system for handling addresses and other numerical values. However, the range of possible numerical values which includes addresses, offsets, and data is vast.

Our intuition was that, with a large enough dataset, the model may learn some relationships between instructions based on addressing. We set the vocabulary which is the maximum unique token count to 3000. This is much lower than the maximum allowed for a large language model. For instance, the BERT-Base-Uncased model has a vocabulary size of 30522 unique tokens [136]. We made this decision because the small numbers and file offsets are frequently repeated in the dataset. For example, small numbers from 0 to 100 are seen numerous times as those numbers are often used as offsets. Offsets are very important for type prediction because they can be used as clues for the size of variables which are indicative of their types. For instance, if a variable takes 2 bytes we can say that variable can't be a `long`. Similarly, if a variable takes only 1 byte, that is a hint that that variable might be a `char`. The size of a variable can be recovered from the offsets used to access the variable. Then the other kind of very frequent numbers are the addresses. Addresses are important for retaining the control flow of the program. Even

though these addresses might seem very random, they are not. Usually, programs are addressed with a virtual address, and the range of addresses the programs can span is not that vast. For example, if the base virtual address for all programs is set to 10000, addresses close to that will be seen repeated among all the programs. So the addresses close to that base address will be more frequent compared to some random integer like 3000000. The vocabulary is generated in such a way that it will retain the most frequently seen addresses and will discard the rarely seen numerical values. So, setting the vocabulary size low would force the tokenizer to keep all the usual tokens from the assembly language and the most frequently seen numerals like offsets and addresses and discard any outliers.

The problem of machine instruction tokenization for LLMs is often not addressed well in the current literature. Even though analyzing machine instructions using neural models [1, 15, 39, 133] is becoming more popular, we believe the tokenization problem should receive more attention.

3.7 Challenges of Fine-tuning for Type Prediction

We transfer the knowledge learned during pre-training to the final model. Pre-training gives a warm-start to the fine-tuning process and helps improve accuracy in the long run [137]. We faced two major challenges during fine-tuning. As shown in Figure 3.2, type data is unbalanced, which causes the model to become biased. The model might see some dominant types excessively and predict everything it sees as those types giving a higher accuracy score overall at the expense of consistently misidentifying rare types.

Undersampling is a technique employed in the training of models on unbalanced datasets. Undersampling involves randomly reducing the frequency of the dominant class to balance the class distribution [138]. This helps the model to pay proportional attention to each class during training, preventing it from being dominated by the more prevalent class [139]. Undersampling further helps to improve the generalization of the model across

all classes, ensuring that it learns meaningful patterns from both frequent and rare classes. We fixed a maximum limit of 20k samples a single type can have in the training data. Extra samples were rejected randomly so that we have all the variations present in our training set.

Another limitation of our dataset is that it is not complete. We used a combination of different tools for labeling the types of as many instructions as possible, but there is still a large portion of instructions without labels. This limitation is also present in all previous datasets presented by other groups because there is simply no way to get ground truth for all instructions in a binary. In many cases, this is because the register usage has no type equivalent in the source language. This is often the case for compiler generated idioms around managing the stack or address calculations. In other cases, it may be a limitation of the tools being used. For this experiment, we are electing to omit predicting the unlabeled instructions. All the scores presented are for predicting the types of only those instructions for which we have the ground truth.

The model receives a series of instructions as input. An efficient method would be to predict all the types of instructions in a single pass as StateFormer [1] did. While this method is fast, it has some serious drawbacks. One is that we do not have labels for all instructions. If the model predicts all the types for each instruction given to it, what about its prediction for the instructions that do not have ground truth? StateFormer tackles this issue by ignoring the prediction for those instructions in calculating evaluation metrics. During the training process, StateFormer will keep predicting the types for those unlabeled instructions and it would have no way to calculate loss for those instructions and learn from mistakes.

Another major problem with this approach is that there is no way to handle the imbalance in the dataset. For example, given the instruction set of a function as input for the model, and the model trying to predict types for all the instructions, there is no way to ignore some instructions for downsampling. Using the concept of program slicing would

also not be possible. Due to these limitations, we have taken the approach of predicting types for one instruction at a time. This process is slower but can handle the above mentioned problems better, giving a more generalized and better performance.

3.8 Implementation Details

3.8.1 Ground Truth Generation

We used a wide variety of tools for the experiment. For dataset generation, we used IDA Pro for disassembly, Ghidra, and pyelftools [140] for ground truth extraction, and Miasm [141] for calculating forward and backward slices. For generating the data dependency graph of a program we used Miasm. We used a 32-core Intel i9-13900k processor with 64 GB of RAM for processing. The repositories we downloaded took around 500 GBs of disk space. The compiled executables with all the optimization level variations and analysis temp files created by IDA Pro took another 1 TB of space. Processing those files with Ghidra using our 32-core machine took around 70 hours and for IDA Pro around 20 hours.

Ground Truth Labeling Method

This step includes aligning the disassembled instructions with the source information with the help of DWARF information and tagging the content of the registers in the instructions with their respective type information. We used Ghidra's type extraction feature which reads the debug information and correlates types for instructions. However, Ghidra cannot extract all labels as this is a complex task even with the DWARF information. Since we have access to the original C source, which Ghidra does not use, we wanted to develop a custom tool that uses both DWARF and source information to label type information in instructions.

The resulting process for labeling types per instruction was fairly complex. The first step was to read the binary and disassemble it. Any modern reverse-engineering tool such as Ghidra or IDA Pro can be used for disassembly. Then we used pyelftools to parse the DWARF information embedded in the binaries during compilation.

```
0x1149: endbr64                                 # {
0x114d: push rbp
0x114e: mov rbp, rsp
0x1151: sub rsp, 0x10
0x1155: mov dword ptr [rbp - 8], 1               #  unsigned |i| = 1;
TYPE:>  [ unsigned int ]

0x115c: mov dword ptr [rbp - 4], 0xfffffffc      #  int j = -4;
TYPE:>  [ int ]

0x1163: mov edx, dword ptr [rbp - 4]             #  printf("%d", i + j); //prints -3
TYPE:>  [ int ]

0x1166: mov eax, dword ptr [rbp - 8]
TYPE:>  [ unsigned int ]

0x1169: add eax, edx
0x116b: mov esi, eax
0x116d: lea rax, [rip + 0xe90]
0x1174: mov rdi, rax
0x1177: mov eax, 0
0x117c: call 0x1050
0x1181: mov eax, 0                               #  return |0|;
0x1186: leave                                    # }
0x1187: ret

```

Figure 3.7: Illustration of instruction and source code mapping using DWARF line information and how we assigned C types to instructions from their source code information.

Aligning Source Code and Machine Code with Line Information

DWARF line information provides essential details about the correlation between machine code instructions and the original source code. Embedded within binary executables compiled with required parameters, this information includes mappings between memory addresses and source code lines, file names, and function names. Using this, we were able to align machine instructions with the corresponding lines of source code. One limitation

of this method is that line information is only found for some instructions, as shown in Figure 3.7. To fill in more detail, we used the heuristic that all the instructions between two instructions with line info are related to the same source as the prior instruction. For example, in Figure 3.7, the instruction in line 1 is mapped with the beginning curly brace, which implies the start of a function. The next three instructions have no mapping information, but we include them with the start of the function. In fact, these instructions are setting up the function stack frame, which is not directly correlated with anything in the source code other than the beginning of a function.

Debuggers use DWARF information to dynamically map memory locations to variable names and their corresponding types during runtime. By associating memory addresses with specific variables and their data types, debuggers can provide real-time insights into variable values, enabling developers to inspect and monitor program state during execution. We also use that information to correlate source-level variables with the specific instructions. When we see an instruction accessing a location where a source-level variable is saved, we can associate the instruction with the variable. We then tagged the instruction with the appropriate source-level type.

3.8.2 Pre-training and Fine-tuning

For implementing SeeType with BERT, we used Huggingface's Transformer library [142]. For training and testing the model during different steps, we used an NVIDIA 4090 GPU with 24 GB of G6X memory which took around 30 hours.

3.8.3 Hyperparameters

During training the batch size was set to 32. We initially found that the validation score was far below the training score and we decided to implement a decaying learning rate which gave us a better validation score compared to the training score. The optimizer we

used was AdamOptimizer [143]. We split the dataset 80%-20% for training and validation. Cross Entropy loss was used for the fine-tuning task.

3.8.4 Evaluation Metrics

We selected our evaluation metrics to measure our model's performance and compare it with the baseline method used by StateFormer and Ghidra. Key measures including F1-Score, Recall, and Precision are crucial for classification tasks. F1-Score, a balance of recall and precision, addresses trade-offs between false positives and negatives. Recall gauges the model's ability to identify instances of a class and is crucial when missing positives have significant consequences. Precision emphasizes the accuracy of positive predictions. Together, these metrics provide a nuanced understanding, guiding decisions for optimizing classification accuracy.

3.9 Results

In this section, we aim to answer the following research questions.

- **RQ1:** How accurate is our method compared to the State-of-the-Art methods?
- **RQ2:** Does pre-training SeeType improve performance?
- **RQ3:** How does our approach to handling long functions perform?
- **RQ4:** How does our tokenization method perform compared to other methods?

3.9.1 RQ1: Correctness of Prediction

Creation of Baselines. We trained StateFormer with our dataset to use as a baseline. Initially, we pre-trained StateFormer with our dataset and then finetuned it with the respective optimization level we are going to compare it with. So, all the evaluation scores

Tool	Compiler Optimization	Accuracy	Precision	Recall	F1
StateFormer	O0	93.9	87.8	87.6	87.7
Ghidra	O0	87	93	87	89
SeeType	O0	**96.4**	**96.4**	**96.4**	**96.4**
StateFormer	O1	**97**	79.1	85.1	82.0
Ghidra	O1	84	91	84	86
SeeType	O1	96	**95.5**	**95.5**	**95.5**
StateFormer	O2	**97**	81.8	84.1	82.8
Ghidra	O2	83	90.2	83	85
SeeType	O2	94	**93.5**	**93.5**	**93.5**
StateFormer	O3	**97**	81.6	83.4	82.3
Ghidra	O3	82.3	90.0	82.3	84.2
SeeType	O3	93.6	**93.6**	**93.6**	**93.5**

Table 3.1: Accuracy, precision, recall, and F1-scores obtained after fine-tuning SeeType and StateFormer with various compiler-optimized binaries. We also present the performance of Ghidra on the same dataset.

presented for StateFormer in this paper are based on training and testing on the same dataset SeeType was trained and tested on. For pre-training and fine-tuning StateFormer, we used the code and other materials provided by StateFormer through github [144]. As for evaluating Ghidra on our dataset, we made Ghidra predict the type information without the debug information. To do that, we stripped each binary and generated the decompilation. We obtained the type prediction of Ghidra from the decompilation it generated.

Comparison with StateFormer. Table 3.1, Table 3.3, and Figure 3.8 present the performance of SeeType and StateFormer on datasets created with binaries compiled with various levels of optimization passes. Our methods outperform the baseline method in the measure of precision, recall, and F1 measures. While StateFormer has higher accuracy, the F1-score represents a better picture of the performance of classification models over an imbalanced dataset.

SeeType has outperformed the baseline due to the improved pre-training and training methods. Notably, as we can see in Table 3.3 and Figure 3.8, SeeType performs well for the majority of the types, irrespective of their sample frequency, as stated in Figure 3.2. For example for the type `*long int`, which has a comparatively lower frequency in the dataset, our model is able to outperform the baseline on Table 3.3 and Figure 3.8. The F1-score of the baseline method on different types depends on the frequency of that particular type in the training dataset. This shows that SeeType handles data imbalance better by using undersampling during training.

Comparison with Ghidra. Ghidra is one of the most popular reverse engineering tools. We can see that Ghidra performed better than StateFormer in all the metrics other than accuracy, like SeeType. SeeType performed better than Ghidra in all the metrics in Table 3.1. In the more fine-grained comparison in Figure 3.8, we can see that Ghidra is performing better than SeeType only for the `int` and `char` classes. For, all the other classes, SeeType is performing better than Ghidra. This shows the robustness of SeeType over all the different types.

Method	Accuracy	Precision	Recall	F1
SeeType (No Pre-training)	95.2	95.2	95.2	95.2
SeeType	**96.4**	**96.4**	**96.4**	**96.4**

Table 3.2: Accuracy, precision, recall, and F1-scores obtained for O0 binaries during fine-tuning without and with pre-training.

3.9.2 RQ2: Impact of Pre-training

We experimented with omitting pre-training. Instead of initiating fine-tuning from the pre-trained model, we started from a vanilla BERT base model. The experiment showed that the model that started fine-tuning from the pre-trained model outperformed its counterpart by 1.4% in the overall F1-score as reported in Table 3.2.

3.9.3 RQ3: Performance of our technique on Handling Longer Functions

For evaluating our approach to handling larger functions, we created a different training set, as the original dataset has functions both small and large. A naive approach for handling larger functions would be to take as many nearest instructions as possible, akin to a sliding window where the target instruction sits in the middle. Here we are defining proximity as the file offset of the instructions. This approach has some merit as the maximum number of instructions we were taking as input was 48, and there is a good chance that those 48 instructions contain enough context for the model to make a good inference. To evaluate our approach, we created another dataset that consists of functions only with more than 48 instructions. Then we discarded any sample with less than 10 instructions in their Symmetric difference of the instructions sets obtained by both methods. We found that our method resulted in a better F1-score compared to the naive approach as reported in Table 3.4. This experiment shows us that there are multiple ways to handle larger functions for feeding into NLP models and this might be interesting future work.

Optimization	Array Long Long Unsigned Int	Long Int	Array Long Int	Unsigned Long Long Int	Array Structure	Double	*Void	*Char	Signed Char	Array Char	*Signed Char	Long Unsigned Int	Long Double	*Int	Long Long Unsigned Int	*Float	Array Long Unsigned Int	*Long Unsigned Int	Structure	Array Signed Char	Array Short Unsigned Int	*Short Int
O0	97	93	97	-	96	99	-	94	98	95	100	96	80	96	98	98	96	98	92	99	96	99
O1	-	94	96	91	-	98	99	96	97	98	93	-	98	96	-	87	-	-	88	90	-	-
O2	-	90	96	87	-	98	99	92	96	97	89	-	83	93	-	80	-	-	83	87	-	98
O3	40	90	95	84	88	98	99	90	91	82	92	85	90	94	88	82	81	0	82	81	87	93

Optimization	*Long Int	*Unsigned Int	Unsigned Short Int	Array Unsigned Int	*Double	Array Float	Array Unsigned Long Int	Short Unsigned Int	Array Unsigned Long Long Int	*Structure	Array Short Int	*Long Long Int	Array Long Long Int	Unsigned Int	Array Int	Long Long Int	Short Int	Int	Unsigned Long Int	Char	Array Double	Float
O0	99	97	-	95	99	97	-	99	-	92	95	96	95	96	95	95	99	94	-	99	97	99
O1	88	96	99	98	91	98	98	-	100	95	99	90	96	96	93	93	97	91	97	98	99	98
O2	81	94	93	96	85	97	92	-	97	93	95	97	94	95	88	90	95	87	97	96	98	97
O3	72	86	92	96	89	97	89	94	98	94	96	94	93	94	90	90	96	89	96	91	98	97

Table 3.3: Percent F1-scores obtained for various C types of all the optimization levels. Not all classes and their scores are shown due to space constraints.

Method	Accuracy	Precision	Re-call	F1
Naive	85.4	86.0	85.4	85.4
Slicing	**87.5**	**87.6**	**88**	**87.4**

Table 3.4: The accuracy, precision, recall, and F1-scores obtained during the comparison of SeeType program slicing with a naive approach to handling longer functions.

Method	Accuracy	Precision	Recall	F1
No Numericals	91.6	91.6	91.7	91.6
SeeType	**93.6**	**93.6**	**93.6**	**93.5**

Table 3.5: Comparison of SeeType tokenization with baseline on O3 binaries.

3.9.4 RQ4: Performance of SeeType's tokenization method

We conducted experiments to evaluate the effectiveness of our tokenization scheme discussed in 3.6. We compared our method's performance with a naive baseline method. In the baseline method, we replaced all the numerical values with a single `<number>` token. Thus the numerical values in the instruction sequence were converted to the token `<number>`. Previous tools such as TypeMiner [36] and PalmTree [133] used a similar scheme for handling numerical values. In our approach, we kept the numerical values unchanged and treated them as general tokens. But, we set the maximum token count of the tokenizer to 3000. This means the tokenizer should train itself to recognize the most frequent 3000 tokens from all the instructions present in the dataset. The tokenizer should learn all the non-numerical tokens and the most frequent numerical values. The comparison of the performance of these two methods is presented in Table 3.5.

3.10 Limitations

The scores presented in Table 3.1 should be observed with an understanding of the limitations of the dataset. In a real-world scenario, this kind of performance may not be reflected.

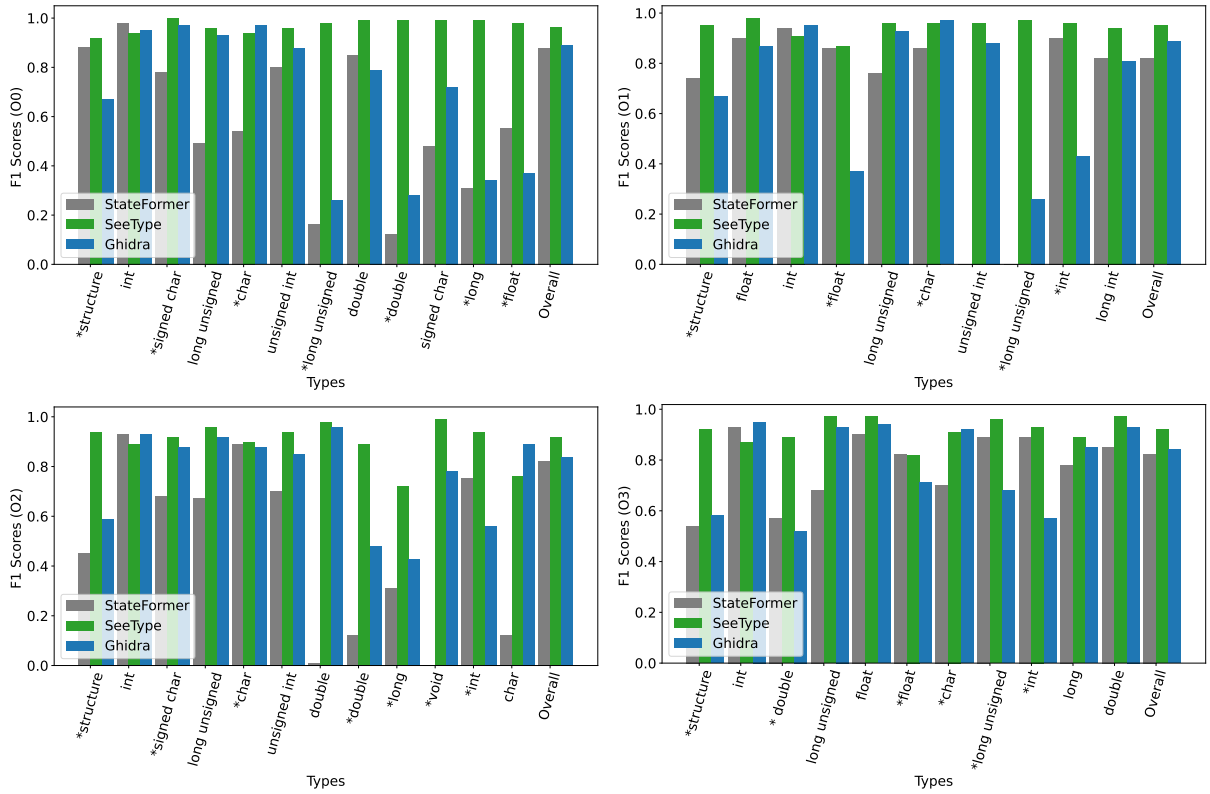


Figure 3.8: Comparison of F1-scores between SeeType, Ghidra, and StateFormer for different C types.

As previously discussed, the models are only as good as the samples they are trained on and the dataset was not complete. There is still work needed to improve the ground truth generation process.

Although our approach of training on a single instruction per sample takes more time to train per instruction, the total time required for the model to be trained is relatively low. This is because during down-sampling we are discarding a large number of samples. For example, 95% of the total samples are discarded during the downsampling of the O0 dataset. Therefore, SeeType is being trained on only 5% of the total dataset on which StateFormer needs to be trained. Training StateFormer is 5 times faster than SeeType's 30 hours on average.

Figure 3.9 shows the loss and F1 curves during training and validation of SeeType during fine-tuning with the optimization level O1 dataset. We can see that the model

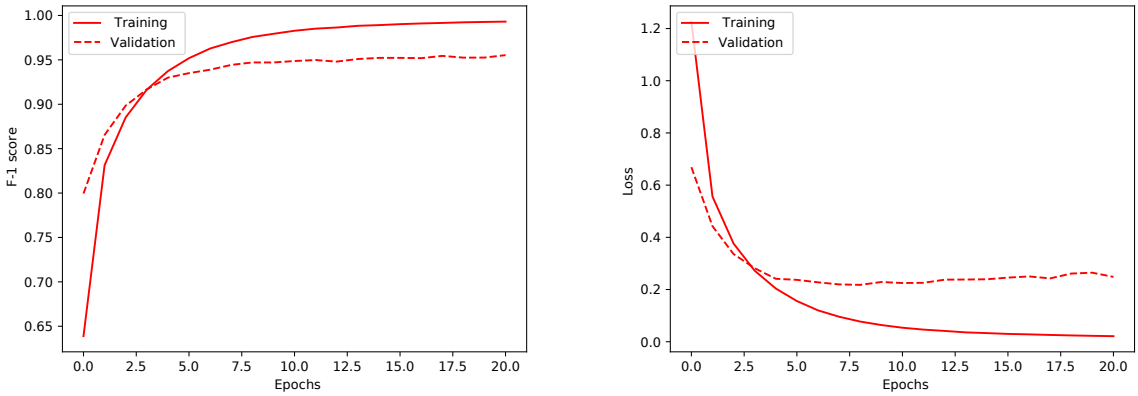


Figure 3.9: F1-score and Training loss curves obtained during fine-tuning SeeType with Optimization level 1 dataset.

gains its peak performance quite early and hence is suitable for fine-tuning with a massive dataset.

3.11 Conclusion

We have presented a progressive approach to train a Transformer-based binary analysis model SeeType. Our method provides a warm-start to the BERT classifier model by pre-training with two self-supervising tasks, Masked Language Modeling, and Data Dependency prediction. Then, our model transfers the learned knowledge to the data type prediction task. Additionally, we have developed an automated method to create a large-scale collection of binary programs along with their corresponding source code to facilitate the pre-training and fine-tuning tasks. We also developed a method to handle large programs using backward and forward slicing. Through training on the dataset we created, our method SeeType showed improvement over the baseline method. We covered a wide variety of granular C types and our model outperformed the baseline classifying the majority of the types. As we have an automated method to create a large set of binaries, we plan on investigating the broader implications of using data-driven binary analysis methods.

Chapter 4

Optimizing NLP Tokenizers for Assembly Code

Tokenization is critical in transforming raw input data into structured representations, a process of utmost importance for Machine Learning (ML) and NLM model tasks [145–147]. While tokenization strategies have been studied extensively for natural [148] and high-level programming languages [149], assembly code presents unique challenges due to its low-level operations, diverse instruction sets, and non-standardized syntax across architectures. These challenges highlight the need for specialized tokenization techniques that effectively capture assembly code's structural and semantic intricacies [146]. Despite its importance, the role of tokenization in assembly code processing remains underexplored, particularly in its impact on downstream tasks involving modern NLMs.

Recent research underscores the significant influence of tokenization on NLM model performance. Studies like Ali et al. [148] demonstrate that tokenization methods affect the model's efficiency and ability to generalize across NLP tasks. Additionally, Dagan et al. [149] emphasize the critical role of tokenizers in domain adaptation and fine-tuning, showing that pre-tokenization schemes and vocabulary size significantly impact model compression rates, training efficiency, and downstream performance. For binary code analysis, tokenization has also played a pivotal role in downstream tasks like binary similarity detection UniASM [146] and function name reassignment Gao et al. [147]. However, existing tokenization methods often fall short when applied to assembly code, primarily

due to their reliance on token patterns optimized for natural language or high-level code, leading to suboptimal results in binary-focused applications.

Assembly code's reliance on hardware-specific instructions and lack of high-level abstractions complicates the creation of structured representations. CP-BCS [150] addresses this challenge by integrating Control Flow Graphs (CFGs) and pseudo code to bridge the semantic gap between assembly and human-readable summaries. Particularly in stripped binaries, advanced tokenization approaches, such as bidirectional instruction-level CFGs and pseudo-code refinement, prove essential for capturing execution behavior and logic semantics. CP-BCS [150] work emphasizes the need for tailored preprocessing Gao et al. [147] and CP-BCS [150], tokenization, and post-tokenization methods in assembly code analysis.

Tokenizers are crucial tools in processing assembly code for NLP tasks in binary analysis, as poorly designed tokenization strategies can significantly hinder model performance. Despite their importance, no comprehensive study has systematically evaluated intrinsic and extrinsic tokenizer performance in assembly code. This study addresses these limitations by systematically evaluating tokenization strategies for assembly code. We focus on the performance of various tokenizers and tokenization methods when applied to decoder-only models, such as Llama 3.2 1B parameters (the model can be downloaded from here), encoder-only models, such as BERT [151], and encoder-decoder models such as BART [152]. Specifically, our contributions are as follows:

- We analyze intrinsic tokenizer performance in detail, assessing its ability to encode assembly instructions and components effectively.
- We evaluate extrinsic tokenizer performance by examining its impact on downstream tasks.
- We evaluate the impact of preprocessing the instructions before tokenization for downstream tasks.

This research contributes to the field by filling a critical gap in understanding tokenization for assembly code. It offers a framework for evaluating and optimizing tokenization strategies tailored to binary program analysis. By leveraging state-of-the-art models and domain-specific datasets, we provide actionable insights for developing more effective tokenization methods, ultimately advancing the capabilities of NLMs in binary analysis.

4.1 Background

Tokenization is crucial in natural language processing and binary analysis, which bridges raw data and machine understanding. It involves segmenting text or binary code into smaller units, known as tokens, enabling efficient processing by machine learning and large language models. The choice of tokenization strategy significantly influences model performance [148] mainly when dealing with specialized languages like assembly code.

4.1.1 Tokenization Algorithms

One weakness of using just words as tokens is that the vocabulary size will be unmanageable. If we have a maximum size limit of the vocabulary, there will be a lot of out-of-vocabulary (OOV) words. One solution might be to tokenize text based on characters. However, that would not generate meaningful tokens, and the number of tokens generated would be very large, even from a shorter text. The subword-based tokenization algorithms try to maintain a balance between these two approaches. This approach generally tries to keep frequent smaller words without splitting in the vocabulary. For example, the word “eat” might not be split, but “eating” might be split into “eat” and “ing”. The subword-based tokenization methods we will be using are:

WordPiece

WordPiece algorithm [153] is a subword-based tokenization formula. It was developed by Google to pretrain BERT and has been reused by other popular models like DistilBERT, MobileBERT, Funnel Transformers, and MPNET.

The WordPiece vocabulary is initialized with the special tokens and the initial alphabet. The initial alphabet is produced from the corpus. It first splits all the words in the corpus into subwords. For example, the word “one” is broken into: ‘o’, ‘##n’, and ‘##e’. Here, ‘o’ is different from the other two alphabets because it is at the beginning of a word. In short, the initial vocabulary contains all the initial letters of a word and all the other letters preceded by the prefix ‘##’. Then, these subwords are merged based on a rule to create longer subwords or whole words. This process is repeated until the vocabulary size is complete. The merging rule: For all the token pairs in the current vocabulary, a score is calculated according to the following formula:

$$score = \frac{\text{Frequency of pair}}{\text{Frequency of first element} \times \text{Frequency of second element}}$$

The pair of tokens with the highest score is merged and added to the vocabulary. Then, the process is repeated until the desired vocabulary size is reached.

During tokenizing new words, WordPiece looks for the longest match in the vocabulary. If the whole word is absent in the vocabulary, the longest match is used to split the word. For example, while tokenizing “cats”, if ”cats” is not present in the vocabulary but “cat” is, it will tokenize as “cat” and ‘##s’.

Byte Pair Encoding

BPE [154] is a data compression technique adapted as a subword tokenization method for natural language processing tasks. It was originally designed for compressing text and later used by models like GPT, GPT-2, BART, and DeBERTA.

The token selection method for vocabulary building is very similar to WordPiece. The major difference is how the score of each pair is calculated before merging. BPE starts with splitting the corpus into characters. So, the vocabulary will start with all the ASCII characters, at the very least, and probably some Unicode characters. Then, it will find the most frequent pair instead of using the equation like in WordPiece. The most frequent pair is merged and added to the vocabulary. This method is repeated until the desired vocabulary size is reached.

For tokenizing new words, BPE uses both the vocabulary and merging rules that it learned during vocabulary production. For example, for tokenizing ”cats”, it will first split the word into characters like ’c’, ’a’, ’t’, and ’s’. Then, it will use the learned merged rules to merge them and form the longest token possible.

Unigram

The Unigram model [155] is a language model that takes a different approach to building its vocabulary than algorithms like WordPiece and BPE. It starts with a large vocabulary and gradually trims it down. Unigram prunes tokens based on how much they impact the model’s likelihood over the entire corpus. In each iteration, Unigram calculates a loss. This loss is computed by tokenizing every word in the corpus, using the current vocabulary and the Unigram model determined by the frequencies of each token in the corpus. Subsequently, it evaluates the potential increase in this overall loss for each symbol in the vocabulary if that symbol were to be eliminated. Then, it removes the percentage of tokens whose log increase is the lowest. This process is iterated until the desired vocabulary size is obtained.

Tokenizing a new word involves examining every possible segmentation of the word into tokens. Each segmentation is evaluated by calculating the probability of that specific sequence according to the Unigram model. The general idea is to split a word into the least number of tokens possible.

Unigram is used in SentencePiece, which is the tokenization algorithm used by popular models like AlBERT, T5, mBART, Big Bird, and XLNet.

4.1.2 Related Works

Preprocessing

Preprocessing text before tokenization involves modifying the original input in such a way that better fits the target task. For example, converting all text to lowercase to ensure consistency, removing punctuation and special characters, etc. In the case of binary analysis, preprocessing is done a little differently than natural language.

Some tools ignore all the numeric values, such as Escalada et al. [18], TypeMiner [156]. Cati [157] and Stride [16] keep relatively small numeric values but remove large numbers.

Palmtree [133] performed a study for instruction representation learning. The authors preprocessed the code by replacing strings and large numeric values with special tokens. However, they kept the smaller numeric values as they often convey important information about accessed local variables, function arguments, or data structure fields.

Gao et al. [147] and CP-BCS [150] performed an instruction normalization method to mitigate data sparsity and OOV issues in binary analysis by simplifying instruction representation. Key steps include retaining mnemonics and registers, generalizing constants, and substituting function addresses and local jumps with placeholder tokens. This approach improves model generalization and learning efficiency.

Tokenization

Tokenization is the process of splitting text into smaller units called tokens, which can be words, subwords, or even characters. This is a crucial step in preparing text for NLP

models, as it transforms raw text into a structured format that the models can understand and analyze. There are various ways binary analysis tools perform tokenization:

Learning-based Encoding: SnowWhite [158] is a machine learning approach to predict type information from stripped binaries. They analyzed their dataset and found that the number of unique tokens was vast due to the prevalence of numeric values. To keep the vocabulary of their tokenizer feasible, they developed a subword model tokenizer based on BPE.

Karampatsis et al. [104] performed an empirical study to find the best way to tokenize source code. Source codes are rich with identifiers, which can cause the vocabulary to explode. They came up with the idea of using character subsequences of tokens (subword units) to reduce the final vocabulary size.

Raw Bytes Encoding: Some tools encode the instructions as raw byte encoding and feed that to the NLP model. The one-hot encoding algorithm commonly uses this scheme. A byte consists of 8 bits and offers a range of 256. A vector of length 256 with one active dimension effectively encodes bytes as vectors. A few tools that pass similar input to NLP models are StateFormer [1], DEEPVSA [159], EKLAVYA [160] and [161].

Instruction-level tokenization: UniASM [146] evaluated the BPE, WordPiece, and three instruction-level (Full, Half, and Piece Instruction) tokenization algorithms to assess their effectiveness in binary code similarity detection tasks. The evaluation results demonstrated that the Full-Instruction tokenization method, which treats a single instruction as a token, consistently outperformed the other approaches by preserving instructions' structural and semantic integrity. Despite the effectiveness of the Full-Instruction tokenization technique, it can result in a more extensive vocabulary and is more susceptible to OOV issues.

StateFormer [1] took a different approach to handling numeric values. They used Neural Arithmetic Unit (NAU) [105], embeddings produced by which are supposed to capture the semantics of numerical values involved in arithmetic operations.

4.2 Approach

4.2.1 Tokenizers

To evaluate the impact of tokenizers on model performance, we conducted an ablation study focusing on the pre-trained models: the decoder-only Llama 3.2 1B parameters, the encoder-only BERT, and the encoder-decoder BART-Base [152] model. Specifically, we created a diverse dataset for training customized tokenizers and models, including 80,000 disassembled C functions for model training and 20,000 disassembled functions for testing. We trained the models for each tokenizer while fixing the remaining configurations, such as datasets, training procedures, and hyperparameters. This controlled setup enabled us to isolate and quantify each tokenizer's effect on the models' downstream performance.

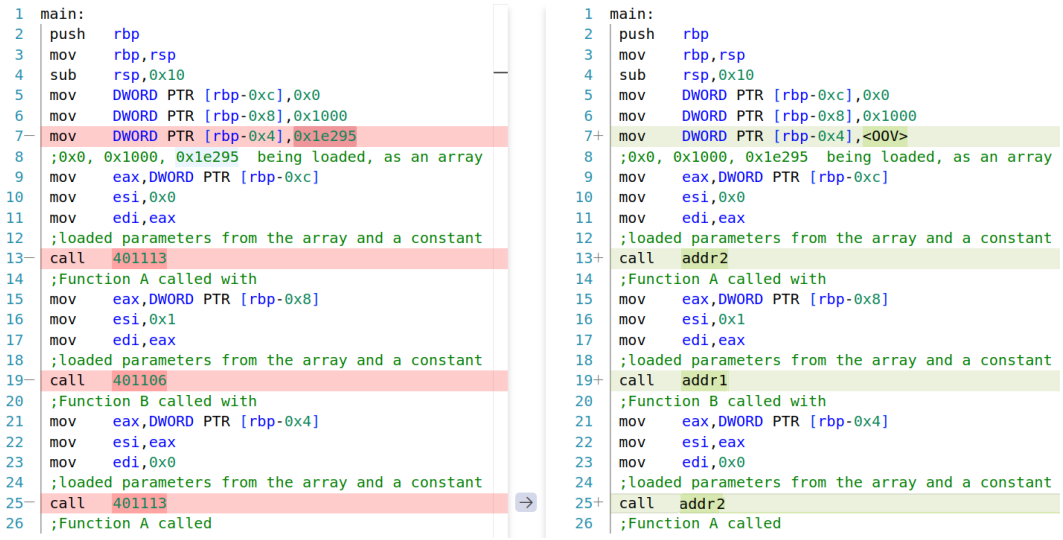


Figure 4.1: An example of address-to-sequential-identifiers preprocessing. The code on the left represents the original code before preprocessing, while the code on the right shows the result after preprocessing.

Our study uses the Hugging Face tokenizer library to implement three well-established tokenization algorithms: BPE, Unigram, and WordPiece. Each tokenization algorithm was tested with three vocabulary sizes: 3K, 25K, and 35K. Additionally, only the Llama 3.2

model was tested with a 128K vocabulary size across all tokenization algorithms on the function signature prediction downstream task. The 128K vocabulary size is comparable to the Llama 3.2 model's default tokenizer's vocabulary size.

To further evaluate the impact of the code preprocessing, each of these tokenizers was trained on two versions of the dataset: the default disassembly without any preprocessing and the other customized using a preprocessing method. The preprocessing method is discussed in detail in the subsection (***C. Dataset***) from this section.

In addition, the base tokenizer that comes pre-trained with the Llama 3.2, BERT, and BART models was evaluated on both versions of the dataset without any customization or additional training. This, combined with assessing the trained tokenizers, resulted in **(86 models)** across the (Llama 3.2, BERT, and BART) models, with three tokenization algorithms (BPE, Unigram, and WordPiece) and four vocabulary sizes (3K, 25K, 35K, and 128K). This experimental setup ensures a comprehensive and comparative evaluation of the Llama 3.2, BERT, and BART models. This design allows us to systematically analyze the effects of tokenization algorithms, vocabulary sizes, and dataset preprocessing on downstream model performance.

The choice of vocabulary sizes was driven by the need to balance efficiency and expressiveness. Smaller vocabularies, such as 3K, are expected to reduce memory and computational overhead while increasing sequence length due to more granular subword segmentation. On the other hand, larger vocabularies, like 35K and 128K, may better capture semantic and syntactic patterns by encoding longer subwords, reducing sequence length but increasing memory usage. The 25K vocabulary serves as a middle ground to assess trade-offs between granularity and model efficiency. Additionally, the 128K vocabulary size was included to evaluate the tokenizers' ability to handle an extensive vocabulary, capturing a wide range of tokens and potential rare subwords, which could be beneficial for highly complex and diverse datasets.

By varying the tokenization algorithm and vocabulary sizes, we aim to analyze the effect of tokenization granularity on downstream model performance, particularly on the decoder-only, encoder-only, and encoder-decoder models. This approach enables us to identify the optimal tokenizer configuration for assembly code analysis tailored to a specific downstream task while accounting for algorithmic and vocabulary design choices.

4.2.2 Preprocessing

NLP models often struggle with understanding and processing numbers effectively because they are primarily trained on textual data, where numbers appear in diverse and inconsistent formats [162, 163]. Unlike words, numbers require precise mathematical reasoning, comparison, or context-specific understanding, which standard tokenization and embedding techniques fail to capture adequately. Experiments have shown that representing numbers in a better way can improve NLP model performance [163]. Considering this and the fact that instructions contain many numerical values and significantly impact the semantics of the code, we must emphasize finding a better numeric representation.

One of the significant challenges in handling numeric values within disassembled code is their extensive range. The sheer variety of possible numbers makes it infeasible for any model to learn embeddings for all of them effectively. Previous tools have tackled this issue in different ways. Some entirely removed numeric values from the disassembled code, while others retained smaller numbers and assigned a special token for larger ones. However, these approaches often lack justification for choosing one method over another. Our work addresses this gap by clearly and systematically comparing different approaches, offering valuable insights into their relative performance and effectiveness.

We are comparing two different variations of representing numeric values in disassembled code. They are:

Default

This approach represents the baseline method, in which the disassembled code, including its numeric values, is used without modification. This unaltered input serves as a reference point for evaluating and comparing the performance of alternative preprocessing methods.

Address to Sequential Identifiers & Hexadecimal Numeric Values to Decimal

This preprocessing method replaces memory addresses in the code with sequential identifiers and converts all hexadecimal numeric values into their corresponding decimal representations. Large and widely varying memory addresses in disassembled code are highly specific to individual programs or runtime environments, while the representation of numeric values in hexadecimal adds further complexity. These variations make memory addresses and hexadecimal values challenging for tokenizers to process and limit their semantic usefulness for downstream tasks.

The proposed preprocessing method replaces every distinct memory address in the code with a sequential identifier. For example, if the code contains addresses such as 0x1FF0, 0x1FF4, 0x2000, and 0x2AB8, they will be converted into tokens like addr1, addr2, addr3, and addr4. Each distinct address is assigned a unique token, preserving its identity while normalizing its representation into a manageable vocabulary. Similarly, all hexadecimal numeric values in the code are replaced with corresponding decimal representations, ensuring uniformity in numeric formats. Suppose the vocabulary size is set to 3000. The most frequent tokens in the code, such as mnemonics and other operational strings, will appear multiple times and naturally occupy slots in the vocabulary. Frequently occurring small numbers will also be included in the vocabulary. In contrast, less frequent large numbers are unlikely to appear in the vocabulary and will instead be replaced with a special token, e.g., $<$OOV$>$. The intuition behind this approach is to normalize memory addresses using unique identifiers, retain frequently used smaller numbers, and eliminate

less frequent outliers. This strategy balances the need for effective representation with the constraints of a fixed vocabulary size. An example illustrating the address to sequential identifiers method is shown in Figure 4.1.

Two memory addresses appear in lines 13, 19, and 25. Additionally, other numeric values are used across different lines. After applying the preprocessing method, the exact values of the addresses are replaced with sequential identifiers, addr1 and addr2. This transformation retains all relevant information. We can still identify these as addresses and understand that lines 12 and 25 call the same function while line 19 calls a different one. The precise values of the addresses are irrelevant for analysis.

Similarly, smaller numbers remain unchanged, while the large numeric value in line 7 is replaced with an OOV token due to its rarity. This replacement highlights its status as an outlier that does not occur frequently. Normalizing memory addresses and handling numeric values appropriately adds value by making the code more structured and interpretable, potentially simplifying the model's task in downstream applications.

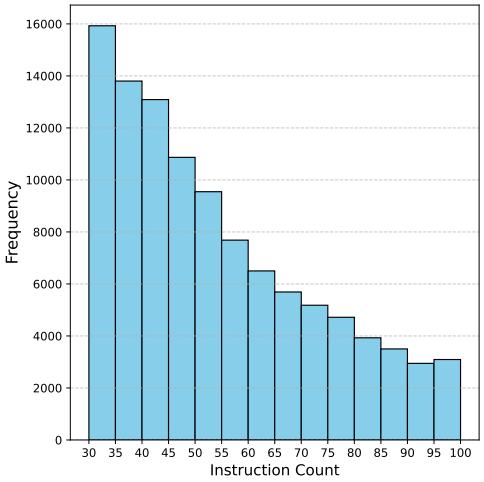


Figure 4.2: Frequency distribution of disassembled functions based on the number of instructions per function.

4.2.3 Dataset

We scraped code from publicly available GitHub repositories containing C source code to create our dataset. The goal was to ensure a diverse and representative collection of C programs. After collecting the source code, we compiled the programs using the GCC compiler with optimization level 2 and debugging information to preserve metadata. After compilation, we disassembled the binaries using Ghidra [2] to extract individual functions and their signatures. We employed TLSH [164], a fuzzy hashing technique, to identify and remove duplicate functions to ensure uniqueness and eliminate redundancy in the dataset. Following deduplication, we randomly selected 100,000 functions to form the final dataset. Although the initial pool of functions was significantly more extensive, we opted for this subset due to resource constraints, as our experiments involved extensive computational demands. We filtered the functions in the dataset to include only those containing at least 30 and at most 100 instructions. A histogram of the Frequency distribution of disassembled functions based on the number of instructions per function is depicted in Figure 4.2. This range was chosen to capture meaningful functionality while avoiding excessively long functions. The resulting dataset provides a robust basis for experimentation, combining diversity from the initial scraping process, uniqueness ensured by deduplication, and a controlled size and complexity range for efficient analysis. We will make our dataset publicly available upon publication.

4.2.4 Models

To assess the impact of the trained tokenizers on downstream model performance, we fine-tuned three models: Llama 3.2 1B, a decoder-only model with causal language modeling (CLM) training objective, the BERT, an encoder-only transformer model, and the BART, encoder-decoder transformer model. The trained tokenizers were used to fine-tune the models, with evaluations conducted on their respective versions of the default

and preprocessing disassembly datasets. This setup allowed for a detailed analysis of how tokenization algorithms, vocabulary sizes, and dataset preprocessing influence the models' downstream task performance.

4.2.5 Evaluation

Our study was structured into two key phases to evaluate the impact of tokenization strategies on downstream model performance: **intrinsic evaluation**, and **extrinsic evaluation**.

Intrinsic evaluation overview

focused on analyzing tokenizer performance independently of the models, particularly emphasizing the fertility metric, the overlap between the vocabulary generated by different tokenizers, and testing all tokenizers against the out-of-vocabulary issue. The evaluation was performed using a held-out set of 20,000 disassembled functions, ensuring the evaluation data was not used during tokenizer training. This phase aimed to provide insights into the tokenizers' properties without considering their direct impact on model performance.

Fertility, a widely used metric in NLP Scao et al. [165]; Stollenwerk [166]; Rust et al. [167], measures the average number of tokens required to represent a word or document. In our study, we adapted this metric to the domain of **functions' disassembly code**, where a function's disassembly serves as an analogous unit to a document in NLP. Specifically, fertility in this context quantifies the average number of tokens required to represent the instructions and operands of a disassembled function. This adaptation allows us to assess the compression efficiency and granularity of the tokenizers when applied to assembly code, which, like natural language, contains structural and semantic patterns.

To calculate fertility, we divided the total number of tokens generated by a tokenizer for a dataset of disassembled functions by the total number of instructions in those functions. Instructions were identified using a standardized parsing process that splits assembly code at line breaks. A higher fertility value indicates lower compression efficiency, suggesting

that the tokenizer produces more tokens per instruction, which can impact the downstream processing of binary code.

By applying the fertility metric to functions' disassembly code, we gained critical insights into how tokenizers such as BPE, Unigram, and WordPiece with varying vocabulary sizes capture low-level code's structural and semantic information. These insights laid the groundwork for the extrinsic evaluations, where we examined the impact of these tokenizers on the downstream performance of the models.

Extrinsic evaluation overview

Extrinsic evaluation assesses the performance of models on downstream tasks to understand the impact of different tokenizers on their effectiveness. In this study, we evaluate the BERT model on masked token prediction accuracy, which aligns with its masked language modeling (MLM) training objective. For the Llama 3.2 model, we evaluate its ability to recover entire disassembled functions, including instructions or parts of instructions, after masking randomly selected tokens. The masked regions are determined by the tokenization strategy employed during the experiments. Although masked token prediction is traditionally a pre-training objective, we included it in our evaluation to measure the models' understanding of disassembly code structure, semantics, and syntax. This analysis underscores the models' ability to interpret low-level assembly instructions and accurately reconstruct missing or obscured tokens.

For the Llama 3.2 model, recovering the entire function disassembly, rather than just the masked tokens, is crucial for assessing its ability to generate coherent and accurate outputs for real-world applications such as code completion, reverse engineering, and vulnerability detection. This evaluation highlights the model's capacity to reconstruct the full execution logic of functions, providing insights into its generative capabilities and robustness in binary program analysis tasks.

Additionally, the Llama 3.2 and BART models were evaluated on the function signature prediction task, a critical downstream task in binary analysis. Function signature prediction involves inferring high-level function prototypes (parameter and return types) from low-level code. This task is important for recovering meaningful symbolic information from stripped binaries, enabling better code comprehension and facilitating downstream tasks such as debugging, optimization, and malware analysis. These evaluations collectively provide a comprehensive view of the tokenizers' impact on model performance across tasks requiring generation and understanding capabilities.

The decision to select the BART-Base model instead of the BERT model to evaluate the function signature prediction downstream task performance was because the BERT is not a generative model and is thus unsuitable for this task. Our choice of models aimed to include a very large model (Llama 3.2) and a smaller model (BART-Base) to provide a comparative perspective. The BART-Base model, being a smaller generative model, offers valuable insights into how model size impacts performance on function signature prediction compared to a larger model like Llama 3.2.

4.3 Intrinsic Evaluation of Tokenizers

We begin by analyzing the fertility scores of the trained tokenizers using an unseen dataset consisting of 20,000 disassembled functions and then examine their vocabulary's overlapping insights.

4.3.1 Analysis of Fertility Scores

The fertility study, as described, evaluates the number of tokens BPE, Unigram, and WordPiece tokenizers require to represent instructions in the unseen default and preprocessed

disassembly datasets. Fertility measures each tokenizer's efficiency and compression capability. Our observations, based on the fertility score comparison depicted in Figures 4.3a and 4.3b, are as follows:

Default Disassembly Dataset

- **WordPiece** consistently shows the highest fertility score across all vocabulary sizes (4.5 tokens per instruction), indicating it produces the most tokens per instruction and demonstrates the lowest compression efficiency.
- **Unigram** achieves the lowest fertility score, consistently around 2.0 tokens per instruction, showcasing the highest compression capability among the three tokenizers.
- **BPE** lies between WordPiece and Unigram, with fertility scores decreasing from approximately 3.0 to 2.5 as the vocabulary size increases.

Preprocessed Disassembly Dataset

- Similar trends are observed, where **WordPiece** maintains the highest fertility score, followed by BPE and Unigram.
- o The preprocessing step slightly reduces fertility for all tokenizers, suggesting better alignment between the tokenizers and the structure of preprocessed disassembly.

We conclude that the **Unigram** is the most efficient tokenizer in terms of compression for both datasets, requiring fewer tokens per instruction, making it ideal for tasks prioritizing compact representations. **WordPiece**, due to its high fertility, may preserve more granularity in tokenization, which could benefit specific tasks requiring detailed token-level information but at the cost of efficiency. The **BPE** tokenizer balances compression and granularity, making it a versatile choice for disassembly tasks.

4.3.2 Vocabulary Overlap Study

The vocabulary overlap study examines the percentage of shared vocabularies across BPE, Unigram, and WordPiece tokenizers for four vocabulary sizes (3K, 25K, 35K, 128K). We measure the overlap for both the default and preprocessed disassembly datasets. Our observations, based on vocabulary overlap percentage shown in Table 4.1, are as follows:

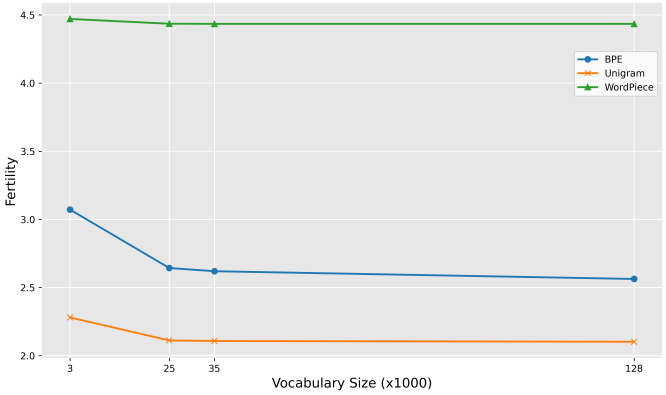
Overlap Trends

- The vocabulary overlap percentage decreases as the vocabulary size increases, indicating less agreement between tokenizers with larger vocabularies.
- **For the default disassembly dataset:** At a vocabulary size of 3K, the overlap percentage is relatively low (0.75 or 63 tokens), which diminishes as the vocabulary size increases to 128K (0.09 or 187 tokens).
- **For the preprocessed disassembly dataset:** The overlap is slightly higher than in the default dataset, especially at smaller vocabulary sizes (1.04 or 86 tokens at 3K).

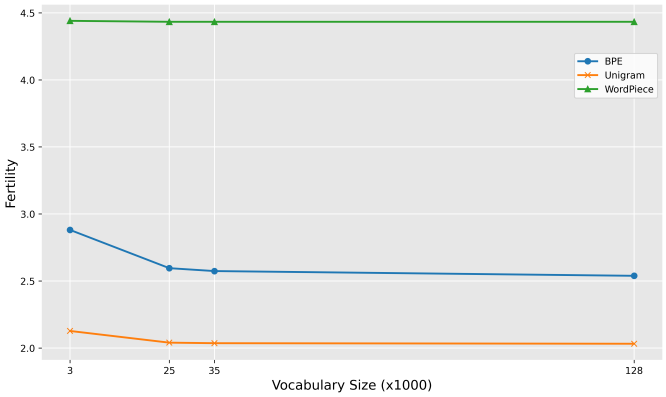
Impact of Preprocessing

Preprocessing enhances vocabulary alignment across tokenizers, as reflected in the higher overlap values for all vocabulary sizes.

We conclude that the overlap between vocabularies is minimal, suggesting that each tokenizer captures unique aspects of the data and may tokenize differently based on its underlying algorithm. Preprocessing the dataset enhances tokenization pattern alignment across the tokenizers, slightly increasing the shared vocabulary. Tasks requiring consistency across tokenizers may benefit from preprocessing to improve uniformity, although



(a) Default disassembly dataset



(b) Preprocessed disassembly dataset

Figure 4.3: Fertility evaluation comparison between BPE, Unigram, and WordPiece tokenizers on (a) The default disassembly dataset and (b) The Preprocessed disassembly dataset.

the distinct tokenization mechanisms (subword segmentation strategies) will continue to produce unique vocabularies.

4.3.3 Out-of-Vocabulary Analysis

All tokenizers (BPE, Unigram, WordPiece) with various vocabulary sizes were evaluated on the test dataset for OOV issues. None of the tokenizers exhibited the OOV problem; they successfully recognized all tokens within their respective vocabulary lists, and no tokens were classified as unk_token.

Table 4.1: Vocabulary Overlap Percentage Across Tokenizers for Different Vocabulary Sizes

Vocabulary-Size	**Default Disassembly**		**Preprocessed Disassembly**	
	Percentage	# Tokens	Percentage	# Tokens
3K	0.75%	63	1.04%	86
25K	0.13%	92	0.25%	174
35K	0.1%	102	0.31%	267
128K	0.09%	187	0.44%	845

Table 4.2: **average accuracy** of Masked Token Prediction on the Default and Preprocessed datasets for BERT Across Tokenizers and Vocabulary Sizes

Tokenizer	**Default Disassembly**		**Preprocessed Disassembly**	
	Llama-Accuracy	BERT-Accuracy	Llama-Accuracy	BERT-Accuracy
Unigram-3k	70.20	73.98	71.93	74.25
Unigram-25k	71.65	82.86	73.64	83.47
Unigram-35k	71.69	83.24	73.62	84.54
WordPiece-3k	71.20	73.38	70.92	75.73
WordPiece-25k	71.00	82.17	71.23	84.45
WordPiece-35k	71.67	83.45	72.40	86.49
BPE-3k	70.90	72.94	72.32	75.28
BPE-25k	70.58	82.84	72.86	85.65
BPE-35k	71.60	84.28	72.86	**86.58**
Model-default	80.48	85.37	82.30	78.08

Table 4.3: **Average Accuracy** of Function Parameter and Return Type Prediction on the Default and Preprocessed Datasets for Llama 3.2 and BART Across Tokenizers and Vocabulary Sizes

Tokenizer	**Default Disassembly**		**Preprocessed Disassembly**	
	Llama-Accuracy	BART-Accuracy	Llama-Accuracy	BART-Accuracy
BPE-3K	82.44	85.19	81.99	87.48
BPE-25K	85.42	86.42	85.45	87.62
BPE-35K	**85.76**	86.94	85.25	87.18
BPE-128K	85.23	-	84.56	-
Unigram-3K	74.78	84.97	75.58	**88.81**
Unigram-25K	77.66	80.57	78.12	86.03
Unigram-35K	77.74	66.36	77.96	81.40
Unigram-128K	76.88	-	77.95	-
WordPiece-3K	82.40	87.44	80.40	84.97
WordPiece-25K	83.25	87.53	81.75	86.48
WordPiece-35K	76.31	87.01	83.48	87.35
WordPiece-128K	83.66	-	82.04	-
Model-default	84.87	86.92	85.71	87.16

4.4 Extrinsic Evaluation of Tokenizers

This section provides an in-depth analysis of how various tokenization algorithms, coupled with different vocabulary sizes and dataset representations, influence performance

in two key areas: masked token prediction and the downstream task of function signature prediction. We applied a masking strategy where 15% of the tokens in each disassembled function were randomly selected and replaced with a special [MASK] token. In contrast, the remaining 85% of the tokens were left unmasked. This approach ensures a consistent proportion of masked tokens across all functions.

The evaluation aims to uncover the interplay between tokenization strategies and model effectiveness in accurately understanding and generating disassembly code. The experimental results for the Llama 3.2 model are discussed in subsections B and C below, while the experimental results for the BERT and BART models are discussed in subsections A and D, respectively below.

4.4.1 Performance Evaluation of Masked Token Prediction with BERT

In this experiment, we evaluated the performance of several tokenizers in the context of masked token prediction accuracy. This task involves predicting the original token based on its context within a sequence where specific tokens have been replaced with a mask placeholder. We focused solely on the accuracy of predicting the masked tokens, which comprised 15% of the input sequence, rather than including unmasked tokens in the evaluation. This approach assesses BERT's ability to accurately infer the masked tokens from the surrounding context.

BERT performed better than Llama in most of the experiments. BERT excels in masked token prediction mainly due to its bidirectional context processing. This allows BERT to effectively understand and use the context around masked tokens. Besides, BERT was originally designed to be pre-trained in masked token prediction, which makes it a better fit for this task.

The evaluation scores presented in Table 4.2 show several trends. The model performed better with higher vocabulary-sized tokenizers. The average accuracy increases

with higher vocabulary size across all the different tokenizers with both the default and preprocessed datasets.

Likely, the reason for that trend is that with a larger vocabulary size, there are fewer or even no OOV tokens, like in our case. Machine code has a high frequency of numerical values, and the range of the numerical values can be very wide. If the vocabulary size is large, the tokenizers can recognize more outliers, assisting in a better token prediction.

An additional improvement is evident when using the preprocessed dataset compared to the default dataset. This outcome is expected, as the preprocessed dataset is normalized by replacing all addresses with sequential identifiers. Address values in the default dataset can vary widely, introducing significant outliers. Normalizing these values reduces variability and eliminates potential outliers, improving model performance. Among the tokenization algorithms, BPE performed best with larger vocabulary sizes.

Notably, the average masked token prediction accuracy across all tokenizers paired with the preprocessed dataset is consistently higher than with the default dataset. Interestingly, while the BERT default tokenizer performed well with the default dataset, it showed significantly poorer performance when paired with the preprocessed dataset, emphasizing the need to align tokenization strategies with the preprocessing approach.

4.4.2 Performance Evaluation of Masked Token Prediction with Llama 3.2

The evaluation scores can be found in Table 4.2. For masked token prediction, Llama did not perform similarly to BERT, which is expected. BERT's specific design and training make it particularly strong in this area. That's why even with a simpler and lighter design, BERT performed better than Llama. However, the highest accuracy is not the goal of this experiment. The impact of the variations in the tokenizing algorithm, vocabulary size, and preprocessing is much more interesting. The preprocessing algorithm consistently enabled higher performance across different tokenizers and vocabulary sizes. The Llama's default pre-trained tokenizer, trained with the dataset, showed the highest performance,

80.48%, and 82.30% accuracy for the default and preprocessed dataset, respectively. For the custom tokenizers, the highest accuracy we obtained is 71.69% for the Unigram-35k tokenizer on the default disassembly. For the preprocessed dataset, the highest accuracy observed is 73.64%.

4.4.3 Performance Evaluation of Function Parameters and Return Types Prediction with Llama 3.2

The Llama 3.2 model evaluation results presented in Table 4.3 focus on function signature prediction, a downstream task of predicting function parameters and return types from the function's disassembly:

Impact of Vocabulary Size

Table 4.3 shows that vocabulary size can marginally impact function signature prediction average accuracy. Increasing the vocabulary size improves accuracy for all tokenizers across the small and moderate vocabulary sizes 3K to 35K (e.g., BPE improves accuracy from 82.44% for 3K to 85.76% for 35K on the default dataset and from 81.99% for 3K to 85.45% for 25K on the preprocessed dataset).

However, the impact of vocabulary size is less pronounced for WordPiece, where improvements are relatively marginal (e.g., WordPiece improves accuracy from 82.40% for 3K to 83.25% for 25K on the default dataset).

Preprocessed vs. Default Datasets

On average, the preprocessed dataset improves the default dataset in function signature prediction accuracy. Preprocessing likely enhances the disassembly functions' structural uniformity and semantic clarity, leading to more accurate parameter and return type predictions.

Impact of Tokenization Algorithms

The performance of the tokenization algorithms differs across datasets and vocabulary sizes:

- **BPE Tokenizer:** Consistently achieves the highest average accuracy across all vocabulary sizes for the default and preprocessed datasets (e.g., 85.76% for BPE-35K on the default dataset).
- **Unigram Tokenizer:** This tokenizer shows the lowest average accuracy among all tokenizers, with results generally below 80% across vocabulary sizes (e.g., 74.78% for Unigram-3K on the default dataset and 75.58% for Unigram-3K on the preprocessed dataset).
- **WorPiece Tokenizer:** Performs moderately well, with accuracy slightly behind BPE but significantly better than Unigram (e.g., 83.66% for WordPiece-128K on the default dataset).

The Llama 3.2 model's default pre-trained tokenizer outperformed the Unigram tokenizer on both datasets and all vocabulary sizes, and on average, it achieved slightly higher accuracy than the WordPiece tokenizer across both datasets and all vocabulary sizes. However, the BPE tokenizer achieved very close average accuracy to the Llama 3.2 pre-trained tokenizer, particularly on vocabulary sizes 25K to 128K.

The BPE achieved an average accuracy of 85.76% for 35K vocabulary size on the default dataset, slightly higher than the average accuracy of the Llama 3.2 model's default pre-trained tokenizer on both datasets.

4.4.4 Performance Evaluation of Function Parameters and Return Types Prediction with BART

The evaluation scores of the signature prediction task with BART are presented in Table 4.3. BART performed best with the smallest vocabulary size and preprocessed disassembly for function signature prediction. Across the tokenizers, the performance is mixed. For the default disassembly, WordPiece performed well consistently across all the vocabulary sizes. With preprocessed disassembly, Unigram-3k performed best with 88.81% accuracy. Similarly, BPE also performed well with smaller vocabulary sizes. This means a combination of preprocessing and smaller vocabulary size represents the token in a better way for the model to understand the parameters and return type work in a function. The preprocessing step normalizes the addresses, which is beneficial for the signature prediction task because the specific values of the addresses are irrelevant for this particular task. It is enough to know that a token is an address, as the exact value is irrelevant.

4.5 Discussion

The choice of tokenization algorithm, vocabulary size, and dataset representation is crucial and should align with the model type and task. For instance, in the masked token prediction pre-training task, the BERT model paired with a BPE tokenizer and a moderate vocabulary size of 35K, applied to a preprocessed machine code dataset, emerged as the optimal configuration among the evaluated options.

Similarly, for function signature prediction, the BART model paired with a Unigram tokenizer with a small vocabulary size of 3K and preprocessed machine code demonstrated the best performance.

A notable finding is the consistent benefit of dataset preprocessing, which enhances downstream performance, particularly for smaller to moderate vocabulary sizes across all tokenizers and models.

However, the performance gains from preprocessing were negligible for the models' default pre-trained tokenizers. Interestingly, BERT's default tokenizer achieved higher average accuracy on masked token prediction tasks using the default dataset compared to the preprocessed version, underscoring the importance of task-specific dataset-tokenizer alignment.

Insights from the Intrinsic Evaluation of Tokenizers: The intrinsic evaluation highlights key trade-offs between tokenization efficiency, granularity, and alignment. Unigram demonstrates superior compression with the lowest fertility scores, making it ideal for tasks prioritizing compact representations. WordPiece, with higher fertility scores, provides granular tokenization, which may benefit tasks requiring detailed token-level information. BPE balances efficiency and granularity, offering versatility for disassembly tasks.

Preprocessing improves tokenization efficiency and slightly enhances vocabulary alignment across tokenizers. However, minimal overlap among tokenizers, especially with larger vocabularies, suggests that each algorithm captures unique data characteristics. Despite this, preprocessing aids in achieving greater uniformity in tokenization patterns, which could benefit tasks requiring consistency.

4.6 Limitations

Despite the scope of our study, it faces the following limitations:

4.6.1 Lack of Hyperparameter Optimization

We did not conduct extensive hyperparameter optimization for each tokenizer to minimize computational time costs and maintain focus on the study's primary objectives. This decision, however, may have constrained the potential performance gains achievable with fine-tuned configurations. Additionally, exploring the impact of hyperparameters such as

learning rate, batch size, or dropout rates on downstream tasks could provide valuable insights. Future work could investigate these interactions to identify optimal configurations that enhance the alignment between tokenizers and models across diverse tasks.

4.6.2 Tokenizer Implementation Variants

Our study relied primarily on specific implementations of tokenizers, such as those provided by the Hugging Face library. While this ensures compatibility with the models used in our study, alternative implementations, such as SentencePiece, may yield different results. Investigating the impact of implementation details on tokenization and downstream performance remains an area for further exploration.

4.6.3 Intrinsic and Extrinsic Correlation Analysis

We did not study the correlation between the intrinsic properties of tokenizers (e.g., vocabulary size, token overlap, token distribution) and their extrinsic evaluation on downstream task performance. Understanding this relationship could provide deeper insights into how tokenizer design impacts model behavior and performance across tasks, and we encourage future work to explore this dimension.

4.6.4 Scaling to Larger Models

While our study focused on models with up to 1 billion parameters, we did not evaluate the tokenizers' performance on larger models. Extending the evaluation to larger architectures may uncover additional insights, as tokenization effects could behave differently in models with significantly more parameters.

4.6.5 Real-world Dataset and Task Coverage

Our evaluation was conducted on specific downstream tasks and machine code datasets. While these tasks and datasets are relevant to the context of this work, they may not fully represent the diversity of real-world applications. Future studies should extend the evaluation to a broader range of datasets and tasks to validate the generalizability of our findings.

By addressing these limitations, future research can refine our understanding of tokenization algorithms, exploring their intrinsic properties, broader applicability, and robustness across diverse tasks, model architectures, and evaluation settings.

4.7 Conclusion & Future Work

This study on tokenization algorithms for binary code analysis highlights the critical role of tokenization strategies in optimizing the performance of LLMs and transformer-based models. By evaluating the intrinsic properties of various tokenizers and their extrinsic performance on downstream tasks like function signature prediction, we demonstrated that both the choice of tokenization algorithm and vocabulary size significantly influence model outcomes. Although we did not directly study the impact of intrinsic tokenizer properties on downstream task performance, our findings emphasize the importance of selecting tokenization strategies that align with task-specific requirements. Additionally, our experiments underscore the value of preprocessing machine code tailored to the context of the downstream task. The optimal preprocessing strategy, however, is highly task-dependent and requires careful consideration.

Since NLMs were not originally designed for binary analysis tasks, our findings provide valuable insights into how binary code can be effectively represented for such models. This representation step is essential and has a large potential impact on the model's performance. Overall, this study establishes a foundational understanding of selecting

appropriate tokenization and preprocessing strategies for leveraging NLMs in binary code analysis tasks.

In future work, we aim to expand our investigation by applying tokenizers to larger datasets that introduce greater structural diversity and dependency varieties in machine code. This will allow us to better understand how tokenization approaches perform with more complex data representations. Additionally, we plan to explore alternative tokenization methodologies, such as comparing the SentencePiece implementation with Hugging Face's implementation, to identify nuances in their impact on model performance.

Furthermore, we intend to scale our evaluations to larger language models exceeding 1 billion parameters, enabling us to assess how tokenizer choices influence performance in more powerful architectures. Finally, we will broaden our evaluation scope by testing tokenizers on a wider range of real-world downstream tasks, ensuring the practical relevance of our findings.

Chapter 5

Understanding the Role of Emergent Capabilities vs Task Supervision in Binary Analysis with Machine Learning

5.1 Introduction

Binary analysis is a core problem in software security and reverse engineering. It supports tasks such as malware analysis, vulnerability discovery, binary rewriting, and program understanding when source code is not available. Many of these tasks are difficult because binaries lack high-level information such as variable names, types, and structure.

Recently, general-purpose large language models (LLMs) have been applied to binary analysis with encouraging results. Prior work shows that LLMs can perform tasks such as binary code similarity, function name prediction, and even decompilation using only prompts, without task-specific training. For example, LLMs have been shown to generate meaningful summaries and embeddings for binary functions, enabling code similarity comparison [168–170]. Other work demonstrates that LLMs can recover function names from stripped binaries using generative decoding [171, 172].

More recently, LLMs have also been used to assist with type inference and structure recovery in binaries. In these approaches, LLMs are typically combined with static analysis to infer or refine variable and structure field types based on recovered code semantics, rather than performing end-to-end type inference directly from raw binaries [7]. In addition, several studies show that LLMs can directly decompile binary code into high-level source representations, with performance improving as model scale increases [173].

These results suggest that LLMs are attractive in settings where labeled data is limited or expensive to obtain.

At the same time, smaller models trained or fine-tuned for specific binary analysis tasks have been widely studied [1, 97, 174–176]. These models are often designed with task-specific strategies and trained using labeled datasets. In many cases, they achieve strong accuracy and are efficient to deploy due to their comparatively smaller scale. However, training a model is expensive, often requiring substantial training data, specialized infrastructure, and machine learning expertise.

Both approaches have their pros and cons. Using off-the-shelf LLMs is attractive as they are ready to use and need almost no effort. On the other hand, commercial LLMs are costly to use. Task-specific models are cheaper to use because they are smaller, but training them is expensive. It is still unclear which approach is best suited to different binary analysis tasks. Most existing studies evaluate only one model type in isolation or focus on a single task. This makes it difficult to understand whether LLMs can replace task-trained models or whether specialization remains important. A clear and systematic comparison is needed to guide both researchers and practitioners.

5.2 Motivation

Binary analysis is a fundamental problem in software security and reverse engineering. It helps program understanding when the source code is unavailable. It supports critical tasks such as malware analysis, vulnerability discovery, binary rewriting, and software behavior verification. These tasks are inherently challenging because compilation removes or obscures high-level program semantics, including variable and function names, data types, and structural abstractions. As a result, binary analysis must infer developer intent from low-level instruction sequences and limited contextual information.

Traditional binary analysis techniques rely on static [2, 177, 178] and dynamic [19, 24, 179] analysis for program understanding. These methods simulate different program

properties and infer program behavior based on them. While these approaches are often sound and interpretable, they require significant engineering effort and tend to be brittle in the presence of compiler optimizations, obfuscation, or unfamiliar coding patterns. To address these limitations, learning-based approaches have increasingly been explored, reframing binary analysis tasks as classification, prediction, or generation problems.

Learning-based work typically employs task-specific models trained on labeled binary datasets. These include neural models for binary similarity detection, function name and signature prediction, type inference, and decompilation. Initial architectures relied on recurrent neural networks [39, 40] and classifier models [15, 36, 37], while more recent approaches have adopted Transformers and Graph-Neural Networks [38, 43, 44]. Such models are generally trained or fine-tuned for a specific task and often achieve strong accuracy with relatively modest model sizes. However, their development depends on curated labeled data and substantial training infrastructure.

More recently, general-purpose large language models (LLMs) have emerged as an alternative way for binary analysis. These models are pretrained at scale on massive corpora of natural language and software code. They exhibit strong generalization capabilities and can be applied to binary analysis tasks in a prompt-based manner, without task-specific training. In this setting, binary artifacts—such as assembly code, intermediate representations, or decompiled output—are serialized as text and provided to the model as input. Surprisingly, off-the-shelf LLMs have demonstrated encouraging performance across a range of binary analysis tasks.

To choose—smaller, task-trained models or large, general-purpose LLMs—raises a fundamental question for binary analysis research: to what extent can model scale compensate for the lack of task-specific supervision? Task-specific models are typically efficient at inference time and can be optimized for well-defined objectives, but they incur high upfront costs for data collection and training. On the other hand, off-the-shelf LLMs

are easy to use with minimal domain expertise, but they are computationally expensive and often lack task-specific performance evaluation.

Binary analysis tasks themselves span a spectrum of complexity, from binary classification problems such as similarity detection to fine-grained semantic tasks such as type inference and decompilation. More insights are needed to decide when general LLMs are a good option and when specialized models remain necessary.

5.3 Foundations of PLMs, LLMs, and Emergent Capabilities for Binary Analysis

This section provides an overview of Pre-trained Language Models (PLMs) and Large Language Models (LLMs). It explains their definitions, shared properties, and key differences, with special attention to what enables them to perform binary analysis tasks.

5.3.1 Definitions and Evolution

Language modeling has evolved through four stages. The first stage used statistical models based on n-gram probabilities [180]. These models relied on counting word sequences and estimating likelihood from frequency. The second stage introduced neural language models, replacing manual statistics with neural sequence models [181]. During this stage, recurrent neural networks (RNNs) [182] and later Long Short-Term Memory (LSTM) [183] networks became the dominant architectures for modeling token order and long-range dependencies. The third stage introduced pre-trained language models (PLMs), where Transformers [81] replaced RNNs and LSTMs and enabled efficient large-scale training with contextual representations (e.g., BERT [184], GPT-1/2 [128], T5 [185]). The fourth stage is large language models (LLMs), which scale Transformer-based PLMs to very large sizes and allow models to perform many tasks directly through prompting without task-specific training [186].

Pre-trained Language Models (PLMs). PLMs such as BERT, GPT-1, GPT-2, and T5 are Transformer-based models trained on large unlabeled corpora and later fine-tuned for specific tasks. They introduced the pre-training + fine-tuning paradigm, in which models first learn general language structure during pre-training and then adapt to downstream objectives during fine-tuning.

PLMs usually:

- Have millions to a few billion parameters
- Require labeled data for task adaptation
- Work as feature learners rather than task solvers

Large Language Models (LLMs). LLMs can be considered as PLMs scaled to very large sizes, typically tens or hundreds of billions of parameters (e.g., GPT-3, PaLM, LLaMA). Their scale allows them to perform tasks directly through prompting, without supervised training on each task.

LLMs generally:

- Exceed 10B parameters
- Are trained on massive corpora
- Show new behaviors not present in PLMs

5.3.2 Shared Characteristics

PLMs and LLMs share common foundations:

- Both rely on Transformer architectures
- Both are trained with self-supervised language modeling
- Both improve as the model and data scale increase

Scaling laws describe predictable improvements in loss as model size grows. However, loss reduction alone does not explain new abilities seen only in large models [187].

5.3.3 Emergent Abilities in LLMs

Emergent abilities are capabilities that appear only at large enough model scales. [188] These abilities are not observed in smaller PLMs, even when trained on the same objectives.

An ability is considered emergent when:

- Small models perform near random
- Larger models suddenly perform well
- The improvement cannot be predicted from smaller-scale trends

Emergence is often seen as a sharp transition in performance when a model crosses a threshold in parameter count or compute. [189]

Common Emergent Behaviors

Large models show several important emergent capabilities:

- **In-context learning(ICL):** The model learns new tasks from examples given only in the prompt. Through ICL, LLMs can act as general-purpose task solvers. While smaller models like BERT or GPT-2 require task-specific fine-tuning, LLMs can solve various real-world tasks directly via prompt-based completion.
- **Instruction following:** Instruction following is the ability of a language model to perform tasks when they are described only through natural-language directions, without needing examples in the prompt. This ability is learned through instruction tuning, where the model is fine-tuned on many tasks rewritten as instructions, enabling it to

map textual directions to actions. It significantly improves zero-shot generalization, but reliable instruction-following typically emerges only in larger models [190].

- **Step-by-step reasoning:** Step-by-step reasoning refers to a model's ability to solve complex problems by producing intermediate reasoning steps before giving the final answer. Small language models typically struggle with multi-step tasks such as math word problems, but large models can handle them when prompted using chain-of-thought (CoT), which explicitly encourages reasoning in stages. [191, 192]

These abilities are weak or absent in PLMs and appear only when the scale becomes very large.

Why Emergence Happens

Emergence requires:

- Very large model capacity
- Sufficient and diverse training data
- Training compute beyond a critical level [188]

When these conditions are met, the model begins to generalize across tasks rather than memorize patterns for specific objectives.

5.3.4 Why LLMs Can Perform Binary Analysis Without Training

Large language models can perform binary analysis tasks without task-specific training, mainly because of emergent abilities that arise at scale. These abilities allow LLMs to generalize to unseen problem types when the task can be expressed in text form.

Binary artifacts such as assembly code can be serialized as sequences similar to natural language or programming code. Because LLMs are trained on massive corpora

that include diverse code and structured text, they learn broad patterns about control flow, memory usage, function behavior, and symbolic relationships [193–195]. When the model is large enough, these learned representations become flexible and transferable.

Emergent abilities play a central role here:

- In-context learning allows LLMs to adapt to binary tasks from examples in the prompt.
- Instruction following allows the model to interpret task descriptions without retraining.
- Step-by-step reasoning allows the model to infer structure and semantics from low-level instructions.

An especially important emergent behavior for binary analysis is Chain-of-Thought reasoning. Binary analysis naturally requires multi-step reasoning. Understanding a function often involves tracking register values, following control flow, interpreting memory accesses, and connecting instruction patterns to higher-level semantics. CoT enables LLMs to explicitly perform these reasoning steps in text, allowing recovery of structure and behavior without supervised binary training.

These emergent capabilities enable LLMs to treat binary analysis as another reasoning and code understanding problem. The model does not need explicit binary supervision because the scale provides sufficient capacity to map unfamiliar token patterns to higher-level meaning.

In contrast, PLMs lack these emergent behaviors. Smaller PLMs mainly learn language features and require supervised fine-tuning to connect representations to binary analysis tasks. They do not naturally generalize from text patterns to complex unseen domains such as binary semantics. Without labeled binary data and training, PLMs cannot reliably infer function behavior or structure.

Therefore, PLMs depend on task-specific datasets and explicit training to perform binary analysis. LLMs, due to emergent capabilities and scale, can perform similar tasks

directly from prompts, even when they have never been trained on binary analysis objectives.

5.4 Problem Formulation

This paper studies how different modeling choices affect performance on binary analysis tasks. In particular, we examine whether large, general-purpose language models can perform binary analysis tasks without task-specific training, and how they compare to smaller models trained with explicit supervision.

We focus on several binary analysis tasks that are widely studied and commonly used in practice. We initially chose a state-of-the-art tool that performs well on this task and compare its performance with Multiple LLMs of different sizes. This will give us a better understanding of how LLMs are suitable for binary analysis tasks and whether using a bigger model helps.

Binary similarity detection is formulated as a binary classification problem. Given a pair of binary functions, determine whether they implement the same functionality. This task requires recognizing high-level behavioral similarity from low-level instruction sequences.

Function name prediction is formulated as a multi-class classification problem. Given a binary function, the task is to predict its original function name from a fixed set of labels. This task focuses on mapping low-level code patterns to high-level semantic intent.

Function signature prediction is formulated as a structured prediction problem. Given a binary function, predict its parameter types and return type. This task requires understanding calling conventions and how values are passed and returned.

Type inference is formulated as a fine-grained semantic prediction problem. Given a binary function, the task is to infer the data types of the variables and parameters used within it. Unlike signature prediction, this task requires reasoning about value usage, memory access patterns, and variable interactions.

Decompilation is formulated as a code generation problem. Given a binary function, produce a high-level source representation that reflects the original program logic. This task requires recovering control flow, data flow, and program structure.

Together, these tasks form a spectrum of increasing semantic complexity. Binary similarity and function name prediction focus on discriminative classification, while function signature prediction, type inference, and decompilation require progressively richer semantic understanding and more structured outputs. Studying this range of tasks allows us to analyze how model scale and supervision interact across different levels of difficulty.

Our goal is not to introduce new algorithms or task definitions. Instead, we use these tasks as a common framework to compare large, unspecialized language models with smaller, task-trained models, and to understand when each approach is most effective.

5.5 Spectrum of Binary Analysis Tasks Under Study

There are many possible binary analysis tasks. To keep the experiment manageable, we select a small set of tasks that still covers a broad range of problem types. Since this work studies language models, we choose tasks where prior work shows that language models perform reasonably well. We focus on tasks for which prior work exists, using task-specific, trained, or fine-tuned models. In this work, we do not train or fine-tune any models. Instead, we reuse state-of-the-art task-specific models and compare their performance with open-source LLMs.

A key motivation for this design comes from recent findings in the literature on generative vs. discriminative modeling. Discriminative models directly predict target labels from input features by learning decision boundaries between classes. In contrast, generative models produce outputs as sequences, typically using autoregressive token generation.

Prior research indicates that utilizing LLMs as generative classifiers is often computationally inefficient and may yield lower accuracy in closed-set classification tasks [196]. This performance gap persists despite the human-centric intuition that recognition and

discrimination are inherently simpler than generation. Recent empirical evaluations of the ”self-[in]correct” hypothesis suggest that autoregressive LLMs lack robust intrinsic self-discrimination; they are often no more effective at identifying a correct solution among candidates than they are at generating the initial response [197]. This limitation is attributed to both the training objectives of decoder-only architectures and the “semantic overlap” in the output space, where shared tokens across labels introduce ambiguity during autoregressive decoding. In contrast, discriminative models—which map inputs to specific labels by optimizing a $P(y|x)$ objective—establish precise decision boundaries and execute classification in a single inference step. Consequently, these specialized models frequently outperform generative approaches in accuracy, calibration, and inference latency.

These findings suggest that generative modeling is poorly aligned with classification tasks, even with large models. However, the same work also shows that generative modeling is still useful for representation learning and can complement discriminative models when used as an auxiliary objective.

In this work, we test whether these observations also hold for binary analysis tasks. Specifically, we study whether large language models perform better on generation-style tasks and whether task-specific trained models perform better on classification-style tasks in this domain.

5.5.1 Binary Analysis Tasks Under Study

The binary analysis tasks we selected for this work are:

- **Binary Classification:** Binary similarity detection
- **Multi-class Classification:** Type inference , Function name prediction
- **Structured prediction:** Function signature prediction
- **Semantic Summarization:** Binary function summarization

- **Code generation:** Decompilation

This set of tasks allows us to study how model behavior changes with task complexity, from simple classification to structured and long-sequence generation. By evaluating performance across this range, we identify where model scale is beneficial and where task-specific supervision remains important. This design also enables us to extrapolate findings to related binary analysis tasks not explicitly studied, such as vulnerability detection, variable recovery, and control-flow reconstruction.

Binary Similarity Detection (Binary Classification, Low Complexity). This task is formulated as a binary classification problem. Given two binary functions, the model predicts whether they implement the same functionality. It requires learning compact semantic representations and comparing them. This is the simplest task in our spectrum, with a small output space and no structured output.

Type Inference (Multi-class / Fine-grained Prediction, Moderate Complexity). This task predicts the types of variables or parameters in a function. It can be viewed as a multi-class prediction problem at a finer granularity. The model must use context, such as value usage and memory patterns. It is more complex than standard classification because of dependencies across multiple program locations.

Function Name Prediction (Multi-class Classification, Moderate–High Complexity). This task is a multi-class classification problem. Given a binary function, the model predicts its function name from a fixed vocabulary. It requires mapping low-level code patterns to high-level semantics. Compared to binary classification or type inference, it has a much larger label space and higher ambiguity.

Function Signature Prediction (Structured Prediction, Moderate–High Complexity). This task is a structured prediction problem. The model predicts both return and parameter types as structured outputs. The predictions are interdependent and must be consistent. This increases the complexity compared to independent-label prediction.

Binary Function Summarization (Text Generation, High Complexity). This task is a semantic text generation problem. Given a binary function, the model generates a short natural language description of its behavior. It requires abstraction from low-level instructions to high-level meaning. The output is free-form, making it more complex than classification.

Decompilation (Code Generation, Highest Complexity). This task is a code generation problem. The model generates high-level source code from binary input. It requires recovering control flow, data flow, and program structure. This is the most complex task in the spectrum due to long, structured outputs and strict correctness requirements.

5.6 Experimental Setup

5.6.1 Models Under Study

In this work, we compare two categories of models: (1) task-specific trained models and (2) general-purpose large language models (LLMs). This comparison allows us to study the trade-off between task-specific supervision and model scale.

1) Task-Specific Trained Models.

For each task, we select a compatible, well-established non-LLM model explicitly trained for that objective. These models represent the *supervised* setting in our study and serve as a contrast to general-purpose LLMs, allowing us to isolate the effect of model scale versus task-specific supervision.

The selected task-specific baselines are summarized in Table 5.1. These models are chosen based on reproducibility, availability of trained checkpoints or pipelines, and strong prior performance on their respective tasks.

2) General-Purpose Large Language Models. We evaluate two open-source LLMs: **GPT-OSS-20B** and **GPT-OSS-120B**. The two models differ primarily in scale, with roughly

Task	Model	Architecture	Model Size
Binary Similarity Detection	JTrans [198]	BERT [184] type Transformer (encoder)	∼110M
Function Name Prediction	AsmDepictor [199]	Custom Transformer (enc-dec)	∼40M
Type Inference	SeeType [97]	BERT type Transformer (encoder)	∼110M
Function Signature Prediction	Fine-tuned BART [200]	BART [201] Type Transformer (enc-dec)	∼139M
Binary Function Summarization	CP-BCS (CodeT5) [202]	T5 [185] type Transformer (enc-dec)	∼220M
Decompilation	SLaDe [203]	BART like Transformer (seq2seq)	∼200M

Table 5.1: Task-specific trained models used as supervised baselines across binary analysis tasks.

20B and 120B parameters, respectively. In addition, we include a commercial large language model, **Claude Sonnet 4.6**, to provide a reference point for modern instruction-tuned LLM performance.

Using models of different scales and training paradigms allows us to study how both model size and alignment strategies affect performance across different binary analysis tasks. Larger models are expected to exhibit stronger emergent capabilities, such as improved reasoning, instruction following, and generalization, which may be beneficial for more complex tasks. Instruction-tuned commercial models, such as Claude Sonnet 4.6, further emphasize robustness in zero-shot and prompt-based settings due to extensive alignment and optimization.

Unlike task-specific models, these LLMs are not trained for any particular binary analysis task. Instead, they are applied in a zero-shot or prompt-based setting. This allows us to evaluate how well general-purpose models can perform binary analysis without explicit supervision.

3) Comparison Objective. The goal of this setup is to analyze how performance changes along two key dimensions:

- **Model Type:** Task-specific (discriminative) vs. general-purpose (generative)
- **Model Scale:** Smaller LLM (20B) vs. larger LLM (120B) vs. large-scale commercial LLM

By comparing these models across the task spectrum defined in Section V, we aim to understand: (i) whether increasing model scale improves performance across all tasks, and (ii) whether LLMs are more effective for generation tasks while task-specific models remain stronger for classification tasks.

This setup directly supports our central research question: Can model scale compensate for the lack of task-specific supervision in binary analysis?

5.7 Experiments and Findings

5.7.1 Binary Similarity Detection

Task-Specific Model Selection

For this task, we use *jTrans* [198] as the task-specific baseline model. jTrans is a Transformer-based model designed for binary code similarity detection. It learns embeddings of binary functions by modeling both instruction semantics and control-flow information. Similarity between two functions is computed based on the distance between their embeddings in the learned vector space.

jTrans is originally evaluated in a ranking setting, where a query function is matched against a pool of candidate functions. It outputs similarity scores that are used to rank candidates based on semantic similarity.

Setup

In this work, we reformulate binary similarity detection as a binary classification task. Given a pair of binary functions, the goal is to predict whether they implement the same functionality. All models, except Claude Sonnet 4.6, are evaluated on a dataset of approximately 4,000 samples, ensuring a consistent and comparable benchmark across approaches. The binaries are x86 and compiled with `gcc` using the `-O2` optimization level.

Model	Acc. (%)	Prec. (%)	Recall (%)	F1 (%)
JTrans	**97.48**	**90.21**	**95.22**	**92.66**
GPT-OSS-20B	64.2	84.62	40.89	55.14
GPT-OSS-120B	69.7	73.75	58.71	65.37
Claude-Sonnet-4.6	89.06	100.0	78.46	87.93

Table 5.2: Binary similarity detection performance (%). Comparison of a task-specific model (jTrans) and general-purpose LLMs under a unified binary classification formulation on x86 (-O2) binaries. Claude Sonnet 4.6 is evaluated on a 128-sample subset due to cost constraints.

To align with this formulation, we convert the original ranking-based outputs of jTrans into binary decisions. Specifically, similarity scores for each function pair are thresholded to produce *yes/no* predictions.

For LLMs, we directly prompt the model with a pair of binary functions and ask it to predict whether they are similar. The output is constrained to a binary decision to ensure consistency across models. For the open-source LLMs (GPT-OSS-20B and GPT-OSS-120B), we use `temperature=0.3`, `top_p=1.0`, and `top_k=0`. For Claude Sonnet 4.6, we use `temperature=0.3`, `top_p=0.7`, and `top_k=5`. Due to cost constraints, Claude is evaluated on a subset of 128 samples, while all other models are evaluated on the full dataset.

Results

Table 5.2 shows the performance on the binary similarity detection task. JTrans achieves the best results across all metrics, with 97.48% accuracy and 92.66% F1.

Among LLMs, Claude Sonnet 4.6 achieves the strongest performance, reaching 89.06% accuracy and 87.93% F1, substantially outperforming both open-source models. GPT-OSS-120B outperforms GPT-OSS-20B, achieving 69.7% accuracy and 65.37% F1, compared to 64.2% accuracy and 55.14% F1 for the smaller model. While Claude significantly narrows the gap with the task-specific model, all LLMs still underperform compared to JTrans.

Model	Accuracy	Precision	Recall	F1
GPT-OSS-20B	0.236	0.210	0.144	0.121
GPT-OSS-120B	0.330	0.272	0.213	0.183
Claude-Sonnet-4.6	0.430	0.393	0.293	0.308
SeeType	**0.916**	**0.916**	**0.916**	**0.915**

Table 5.3: Type prediction performance comparison between general-purpose GPT-OSS models and the task-specific SeeType model. SeeType substantially outperforms both LLMs across accuracy, precision, recall, and F1, highlighting the advantage of domain-specific modeling for closed-set binary type prediction.

Analysis and Takeaway

The results show a clear performance gap between task-specific and general-purpose models for binary similarity detection, with jTrans achieving the best performance across all metrics. This reflects a fundamental difference in modeling approach. Binary similarity is a binary classification task with well-defined decision boundaries. jTrans is trained to learn discriminative embeddings optimized for semantic equivalence, enabling precise separation between classes.

Among LLMs, performance varies significantly. The open-source models (GPT-OSS-20B and GPT-OSS-120B) lag far behind the supervised baseline, confirming that generative, prompt-based inference is not well aligned with classification tasks. Their autoregressive nature introduces ambiguity and weaker decision boundaries, leading to lower recall and overall performance. While increasing model scale improves results (120B $>$ 20B), the gains remain limited.

In contrast, Claude Sonnet 4.6 achieves substantially stronger performance, significantly narrowing the gap with jTrans. This suggests that sufficiently large and well-aligned LLMs can approximate discriminative behavior even in low-complexity classification settings with small label spaces. However, despite this improvement, Claude still does not surpass the task-specific model.

Overall, these results indicate that for binary similarity detection, task-specific supervised models remain the most effective.

5.7.2 Type Inference

Task-Specific Model Selection

For this task, we use SeeType [97] as the task-specific baseline. SeeType is a Transformer-based model that formulates type inference as a classification problem, predicting source-level data types from disassembled code. It leverages contextual instruction sequences to capture patterns in register usage, memory access, and data flow, and is trained on a large dataset of over 559,000 compiled executables. The model uses program slicing to extract relevant context, enabling efficient handling of long functions while preserving semantic dependencies. Prior results show that SeeType outperforms established tools such as Ghidra and StateFormer [1] for type inference, demonstrating the effectiveness of task-specific Transformer models for this problem.

Setup

We formulate type inference as a multi-class classification problem: given the disassembly of a function and a target instruction, predict the data type of the variable it manipulates. The input consists of instruction sequences with a marked target location, following the formulation used in SeeType [97]. The binaries are x86 and compiled with `gcc` using the `-O2` optimization level.

To ensure a fair and consistent evaluation, all models, except Claude Sonnet 4.6, are tested on a dataset of approximately 4,000 samples. Since type distributions in binaries are inherently imbalanced—where common types such as `int` dominate while others are significantly rarer—we apply data balancing during sample selection. Specifically, we limit the number of samples per type and downsample dominant classes, ensuring that both frequent and rare types are adequately represented. This follows the motivation that imbalanced data can bias models toward majority classes and degrade generalization across the full label space.

For LLM-based evaluation, we provide the same disassembly and target instruction, explicitly constrain the output space by listing all possible type classes in the prompt, ensure predictions map to valid labels, and enable a controlled comparison with classification models. For the open-source LLMs (GPT-OSS-20B and GPT-OSS-120B), we use `temperature=0.3`, `top_p=1.0`, and `top_k=0`. For Claude Sonnet 4.6, we use `temperature=0.3`, `top_p=0.7`, and `top_k=5`. Due to cost constraints, Claude is evaluated on a subset of 128 samples, while all other models are evaluated on the full dataset.

Results

Table 5.3 shows the performance on the type inference task. The task-specific model, SeeType, achieves the best results across all metrics, with 0.916 accuracy and 0.915 F1.

Among LLMs, Claude Sonnet 4.6 achieves the strongest performance, reaching 0.430 accuracy and 0.308 F1, outperforming both open-source models. GPT-OSS-120B outperforms GPT-OSS-20B, achieving 0.330 accuracy and 0.183 F1, compared to 0.236 accuracy and 0.121 F1 for the smaller model. Despite these improvements, all LLMs substantially underperform SeeType.

Analysis and Takeaway

The results reveal a substantial performance gap between task-specific and general-purpose models for type inference, significantly larger than in binary similarity detection.

Type inference is a fine-grained multi-class classification task that requires understanding variable usage, memory access patterns, and contextual dependencies across instructions. Task-specific training enables precise decision boundaries over this complex label space, which LLMs struggle to approximate.

While stronger models such as Claude improve performance over smaller LLMs, the gains remain insufficient. Unlike low-complexity classification tasks, the larger output space makes accurate prediction substantially harder for generative models.

Model	ROUGE-L F1	Token F1	Jaccard	Cosine (mean)	Cosine (median)
AsmDepictor	**0.748**	**0.730**	**0.767**	**0.858**	**1.000**
GPT-OSS-120B (capped)	0.045	0.021	0.039	0.463	0.448
GPT-OSS-20B (capped)	0.036	0.015	0.034	0.456	0.446
GPT-OSS-20B (uncapped)	0.041	0.015	0.036	0.461	0.446

Table 5.4: Function name prediction performance comparison between OSS models and a fine-tuned custom-trained transformer-based model. Custom-trained models with strict vocabularies outperform general language models on token-overlap metrics. The cosine similarity column (computed with `Qwen/Qwen3-Embedding-0.6B [204]`) shows the gap narrows but does not close under semantic scoring.

Model	ROUGE-L F1	Token F1	Jaccard	Cosine (mean)	Cosine (median)
AsmDepictor	**0.779**	**0.767**	**0.800**	**0.877**	**1.000**
Claude Sonnet 4.6	0.123	0.092	0.118	0.523	0.472
GPT-OSS-120B (capped)	0.050	0.023	0.052	0.453	0.447
GPT-OSS-20B (capped)	0.016	0.010	0.022	0.442	0.433
GPT-OSS-20B (uncapped)	0.020	0.010	0.027	0.457	0.444

Table 5.5: Function name prediction performance on a 128-sample subset of the evaluation set used in Table 5.4, with Claude Sonnet 4.6 included for comparison. Even on the small subset the frontier model performs better but not to the same level as the fine-tuned model.

Overall, these results reinforce that for fine-grained, multi-class classification tasks, task-specific supervision remains critical.

5.7.3 Function Name Prediction

Task-Specific Model Selection

There are many recent works that aim to generate accurate labels for functions from both disassembly and decompilation [175, 199]. We chose to evaluate against a model trained on disassembly rather than decompilation. The choice between disassembly and decompilation involves a trade-off: Decompilation is a lossy process that removes precise details, whereas disassembly preserves them, yet most large language models have been trained on orders of magnitude more C code than assembly. We chose disassembly because preserving the precise binary representation matters more for this evaluation than matching the LLMs' pretraining distribution. Among disassembly-based approaches, we

selected AsmDepictor for its reproducibility and performance. It is trained on publicly available datasets and includes complete, usable build scripts. Many alternatives had outdated training scripts or lacked an accessible pretrained checkpoint.

Setup

We evaluate AsmDepictor [199], a custom Transformer encoder–decoder trained from scratch on x86-64 disassembly, against two general-purpose open-source large language models: OpenAI's GPT-OSS-20B and GPT-OSS-120B. AsmDepictor was trained on a dataset of over 300,000 labeled function disassemblies. All three models are evaluated on the same 2048 function subset of the AsmDepictor [199] test split. AsmDepictor takes the first 300 assembly tokens of each function as input, matching its training configuration. The LLMs receive the same disassembly, reformatted into a human-readable layout, and are queried with a fixed system prompt that asks for a single space-delimited function name, with no commentary. We ran two LLM variants to check whether AsmDepictor's 300-token limit was affecting the LLMs: a capped variant truncated to the same 300-token window, and an uncapped variant that passes the full disassembly. Both LLMs were queried with a `temperature=1.0` and a reasoning effort set to `medium`. Beyond AsmDepictor's built-in ROUGE-L [205], token F1, and Jaccard scores, we report mean and median cosine similarity between predicted and ground-truth names, computed with `Qwen/Qwen3-Embedding-0.6B` [204]. Cosine similarity rewards semantically close predictions (e.g. "find leaf node" vs. "search leaf node") that token overlap metrics miss.

Results

Table 5.4 reports results on the shared 2048 function test subset. AsmDepictor wins on every metric by a wide margin: ROUGE-L F1 of 0.748 versus 0.045 for the best LLM, and mean cosine similarity of 0.858 versus 0.463. AsmDepictor's median cosine is 1.0, indicating that at least half of its predictions match the ground-truth name exactly. Both

Model	Exact Match Acc.	Precision	Recall	F1
GPT-OSS-20B	0.399	0.548	0.548	0.548
GPT-OSS-120B	0.480	0.619	0.619	0.619
Claude-Sonnet-4.6	0.59	0.73	0.73	0.73
BERT (Fine-tuned)	**0.806**	**0.815**	**0.806**	**0.798**

Table 5.6: Signature prediction performance comparison between general-purpose GPT-OSS models and a domain-pretrained BART model. Domain-specific pretraining yields substantially higher exact-match and slot-level accuracy.

GPT-OSS models sit near a median of 0.45. Among the LLMs, GPT-OSS-120B edges out GPT-OSS-20B on every metric, but the gap between the two is small relative to the gap of AsmDepictor. Removing the 300-token input cap for GPT-OSS-20B produces only a marginal change, indicating that the 300-token truncation is not what is holding the LLMs back. In line with the other tasks a frontier model was tested against a subset of the test split used for the original test. Claude Sonnet naturally outperforms both GPT-OSS variants by a small margin on every metric, but once again does not close the gap with the fined tuned models.

Analysis and Takeaway

The token overlap metrics overstate the gap. They reward a system only for emitting tokens that match AsmDepictor's training label vocabulary. AsmDepictor's decoder is constrained to exactly that vocabulary. The LLMs, while likely exposed to similar code during pretraining, have no targeted prior on the benchmark's specific labeling conventions. Substituting cosine similarity over a sentence-embedding model narrows the measured gap by capturing semantic agreement across paraphrases, but does not close it. AsmDepictor still leads by roughly 0.40 absolute cosine. For function name prediction, where the target distribution is a constrained, learnable label vocabulary, a specialized encoder and decoder trained on that vocabulary outperforms substantially larger general-purpose models.

5.7.4 Function Signature Prediction

Task-Specific Model Selection

For this task, we use a Transformer-based signature prediction framework developed in this work [200] as the task-specific baseline. This approach models function signature prediction as a sequence generation problem, where the model predicts both parameter types and return type from disassembled binary code.

The framework leverages an encoder–decoder architecture (BART) trained on large-scale disassembly datasets extracted from compiled binaries. It learns to map low-level instruction sequences to high-level function prototypes, capturing calling conventions, argument usage patterns, and return semantics.

Empirically, the BART-based model achieves strong performance (up to 85–87% accuracy) on parameter- and return-type prediction tasks. Given its reproducibility, the availability of the dataset and the training pipeline, and its strong performance, this framework serves as an effective task-specific baseline for function signature prediction.

Setup

We formulate function signature prediction as a structured prediction problem over a discrete label space. Given a function's disassembly, the model generates the parameter types and return type sequentially as a structured output. Each predicted type corresponds to a class from a predefined set of possible types. The generated sequence is then parsed and converted into the corresponding function signature representation. The binaries are x86 and compiled with `gcc` using the `-O2` optimization level.

To ensure a fair and consistent evaluation, all models, except Claude Sonnet 4.6, are evaluated on a dataset of approximately 4,000 samples. Unlike type inference, we do not explicitly balance the dataset, thereby preserving the natural distribution of signatures. This results in a skewed distribution dominated by common patterns, reflecting real-world

scenarios and making the task comparatively easier for models to exploit frequent structures.

For LLM-based evaluation, we provide the same disassembly as input and constrain the output format to produce structured signatures (e.g., `RET=<type>, PARAMS=<t1,t2,...>`), ensuring consistency across models and enabling direct comparison with sequence-based baselines. For the open-source LLMs (GPT-OSS-20B and GPT-OSS-120B), we use `temperature=0.3`, `top_p=1.0`, and `top_k=0`. For Claude Sonnet 4.6, we use `temperature=0.3`, `top_p=0.7`, and `top_k=5`. Due to cost constraints, Claude is evaluated on a subset of 128 samples, while all other models are evaluated on the full dataset.

Prior work has also modeled this task as a multi-step classification problem, where separate models (or classifiers) are trained to predict the return type and each parameter position independently. In contrast, we adopt a sequence generation formulation, as it aligns naturally with both LLMs and encoder–decoder models like BART and enables a direct, fair comparison across model families.

Results

Table 5.6 presents the performance on the function signature prediction task. The task-specific model, fine-tuned BART, achieves the best results across all metrics, with 0.806 exact match accuracy and 0.798 F1.

Among LLMs, Claude Sonnet 4.6 achieves the strongest performance, achieving 0.59 exact-match accuracy and 0.73 F1, substantially outperforming both open-source models. GPT-OSS-120B outperforms GPT-OSS-20B, achieving 0.480 exact match accuracy and 0.619 F1, compared to 0.399 accuracy and 0.548 F1 for the smaller model. Despite these improvements, all LLMs still underperform the task-specific model.

Analysis and Takeaway

Function signature prediction shows a clear gap between supervised and LLM approaches, but it is notably smaller than in type inference, indicating better alignment with LLM capabilities. Stronger and better-aligned models significantly improve performance, with Claude outperforming the open-source LLMs. While scaling contributes to these gains, alignment and training quality play a more critical role than parameter size alone.

Unlike type inference, which is evaluated on a balanced dataset, signature prediction is naturally skewed, dominated by common patterns. This makes the task comparatively easier, allowing models to exploit frequent signature structures.

This task is structured generation rather than pure classification, which better aligns with LLM capabilities. As a result, LLMs can generate plausible parameters and return types, with Claude producing more consistent, semantically coherent outputs than GPT-OSS models. However, LLMs still struggle to maintain global consistency across the full signature, leading to lower exact-match performance.

Overall, signature prediction represents a transition point in the task spectrum. LLMs become competitive as tasks align with structured generation, and scale, combined with alignment, significantly improve performance. However, precise and consistent outputs still depend on task-specific supervision, which remains the most effective approach for end-to-end correctness.

5.7.5 Binary Function Summarization

Task-Specific Model Selection

We chose to replicate the CP-BCS framework [202] for binary function summarization, as the dataset and source code are available and reproducible. This work provides a framework with a custom CodeT5 model that can take input disassembly, control flow graphs, and pseudo-code. For our purposes, we use the disassembly option and omit

Model	Dataset	BLEU	ROUGE_L	Meteor	Cosine (mean)	Cosine (median)
GPT-OSS-120B	Valid	0.129	0.041	0.022	0.4238	0.4004
GPT-OSS-120B	Testing	0.131	0.041	0.024	0.4375	0.4082
GPT-OSS-20B	Valid	0.125	0.039	0.022	0.4160	0.3906
GPT-OSS-20B	Testing	0.127	0.037	0.022	0.4297	0.4004
CP-BCS	Valid	0.216	0.178	0.101	0.4668	0.3965
CP-BCS	Testing	0.226	0.183	0.098	0.4746	0.4141
Claude-Sonnet-4.6	Testing*	0.131	0.070	0.038	0.4727	0.4648

Table 5.7: Binary function summarization between OSS and fine-tuned CP-BCS model. The fine-tuned CP-BCS model outperforms GPT-OSS-20B and GPT-OSS-120B, while Sonnet performs similarly to CP-BCS with a higher median cosine similarity. Sonnet is evaluated on a subset of the test set, comprising 128 of the 1,494 functions.

the other inputs. The work presents multiple architectures, from which we select x86_64 binaries compiled with the `-O2` optimization level.

Setup

Using the available source code, we retrained on the provided dataset to replicate the existing work [202]. We then use the same dataset to query GPT-OSS-20B and GPT-OSS-120B, with a `temperature=1.0` and a reasoning setting of `medium`. We further query Claude Sonnet 4.6 using a `temperature=1.0`, `seed=42`, reasoning set to `medium`, and `top_p=1.0`. We prompt GPT-OSS and Sonnet with a zero-shot prompt that has instructions to examine the given disassembly and provide a short summary that matches the formatting of the ground-truth in the dataset. The x86_64 dataset of `-O2` compiled functions contained 1,494 functions in the valid dataset and 1,493 in the test dataset. Due to limited resources, we limited testing of Sonnet to 128 functions from the testing dataset (out of 1,494).

We evaluate summarization using BLEU, ROUGE-L, and METEOR, which measure n-gram overlap with ground-truth summaries but favor exact wording. Since binary function summarization allows multiple valid phrasings, we additionally use cosine similarity between embedding representations to capture semantic alignment. Cosine similarity is

a better fit in this setting because it measures meaning-level similarity in vector space, rewarding semantically correct summaries even when lexical overlap is low, and thus complements token-based metrics for a more balanced evaluation

Results

Under exact-match metrics (BLEU, ROUGE-L, and METEOR), the fine-tuned model consistently outperforms GPT-OSS-20B and GPT-OSS-120B. The GPT-OSS models achieve lower token-overlap scores (e.g., BLEU $\approx$ 0.12–0.13 vs. $\approx$ 0.22 for CP-BCS). However, this gap narrows under cosine similarity, where GPT-OSS reaches $\approx$ 0.43 compared to $\approx$ 0.47 for CP-BCS. Claude Sonnet 4.6 further improves semantic performance, achieving cosine similarity comparable to CP-BCS and the highest median cosine score, indicating stronger semantic alignment despite weaker lexical matching. Overall, these results suggest that token-based metrics underestimate LLM performance, while cosine similarity better captures their ability to generate semantically accurate summaries.

Analysis

As shown in Table 5.7, token-overlap metrics disadvantage LLMs, as we found that enforcing stricter adherence to ground-truth phrasing via modifying the prompt improves their scores but does not achieve the scores of CP-BCS. This indicates that part of the performance difference arises from evaluation bias rather than purely model capability. Under semantic evaluation the gap narrows substantially, with cosine similarity showing that GPT-OSS models capture meaningful function behavior despite weaker lexical alignment. Claude Sonnet 4.6 further strengthens this trend, achieving semantic performance comparable to CP-BCS and outperforming GPT-OSS, demonstrating that both scale and alignment improve semantic generation quality.

Model	Function Compilation	Unit-Test Execution
SLaDe	49.0%	8.0%
GPT-OSS-20B	84.0%	58.0%
GPT-OSS-120B	94.0%	67.0%
Claude-Sonnet-4.6	**96.0%**	**80.0%**

Table 5.8: Decompilation pass rates on the shared 100-function HumanEval-Decompile subset. Function compilation reports the percentage of recovered C functions that compile; unit-test execution reports the percentage that also pass the benchmark's unit tests.

Takeaway

Evaluating "accuracy" of function summaries against ground-truth is biased against the expected words. We found that when we didn't specify the length of the code comment, GPT-OSS would be more verbose, and while it might have been an acceptable description of the function, it wasn't a good match given the provided metrics. We also found that GPT-OSS was not as performant as a model trained specifically for binary function summarization, while a larger model such as Claude Sonnet was closer in performance to the pre-trained model.

5.7.6 Decompilation

Task-Specific Trained Model

We use **SLaDe**, a 200M-parameter sequence-to-sequence Transformer introduced by Armengol-Estapé et al. [203] for recovering C code from optimized assembly. Unlike general-purpose LLMs, SLaDe is trained directly on the assembly-to-C mapping and ships with architecture- and optimization-specific checkpoints, making it a natural representative of the small, supervised-model regime. We use the authors' released x86 `-O3` checkpoint from the public artifact repository. A practical advantage of this model class is deployability: SLaDe runs locally without GPU acceleration, and our 100-function decompilation run completed in about 6 minutes on a workstation running Windows 11 Pro with a 12th Gen Intel Core i9-12900K CPU (16 cores, 24 threads) @ 3.20GHz and 64GB RAM. This offline,

CPU-only profile stands in contrast to the general-purpose LLMs, which are impractical to host under the same constraints.

Setup

We use **HumanEval-Decompile**, the C adaptation of OpenAI's HumanEval benchmark [193] released by Tan et al. [206], who translated the 164 Python tasks and their assertions into self-contained C functions that compile with GCC and the standard libraries. Each reference function is compiled at `-O3`, and the resulting x86-64 disassembly is the sole input to each evaluated model. Of the 164 functions, 64 exceed SLaDe's 1024-token input limit and are excluded; all models are therefore evaluated on the same **100-function** subset. Each generated function is checked in two stages: (i) whether the recovered C code compiles at the function level, and (ii) whether it passes its HumanEval-Decompile unit tests under a 300-second test timeout *(per function)*. Unit-test success is the primary criterion, since compilation alone does not guarantee semantic correctness; outputs that fail to compile or to execute within the test timeout are counted as failures, though no run reached the test timeout in our experiments.

For the general-purpose models, we query GPT-OSS-20B, GPT-OSS-120B, and Claude Sonnet 4.6 through the OpenRouter chat-completions API [207], adopting the minimal prompt template from SLaDe's LLM-comparison methodology modified only to name the target architecture: `Decompile the following x86-64 assembly function into C code. Return the code and only the code, no explanations or introductions.`, followed directly by the raw disassembly of one function, with no demonstrations, no chain-of-thought scaffolding, and no auxiliary metadata. All three models are queried with a single user-role message per function and no system prompt, under the same sampling configuration (`seed=42`, `temperature=1.0`, `top_p=1.0`) and a 180-second per-request inference timeout. Reasoning is enabled for every call, but OpenRouter exposes the control surface differently across vendors: for the GPT-OSS models, we set `reasoning_effort=high`,

whereas for Claude Sonnet 4.6, we set `reasoning.enabled=true` with `verbosity=high`, since OpenRouter's Claude 4.6 interface uses adaptive thinking and ignores fixed effort levels. SLaDe is not prompted in natural language; we run it through its released decompilation checkpoint and tokenizer pipeline. All four systems share the same disassembly inputs, output format, and compile-and-test harness, so any performance gap reflects model behavior rather than differences in task formulation.

Results

Table 5.8 reports the decompilation pass rates on the 100-function subset for all four systems, covering both function compilation and unit-test execution.

Analysis

On the 100-function subset, the general-purpose LLMs outperform the task-specific decompiler by a wide margin. Claude Sonnet 4.6 leads overall at 96.0% function compilation and 80.0% unit-test pass rate, followed by GPT-OSS-120B at 94.0%/67.0%, GPT-OSS-20B at 84.0%/58.0%, and SLaDe at 49.0%/8.0%. Across all three general-purpose LLMs, scale and general reasoning ability outweigh specialized training on the assembly-to-C task.

SLaDe exhibits two distinct weaknesses. The first is *coverage*: its 1024-token input limit excludes 64 of the 164 HumanEval-Decompile functions before inference. The second is *semantic recovery*: of the 49 functions SLaDe compiles, only 8 pass the unit tests.

The general-purpose LLMs show the opposite profile: broader reasoning capacity and larger effective context translate into much stronger decompilation quality, despite no task-specific training. This advantage matters most at `-O3`, where recovering semantics often requires more than local instruction-pattern matching. Among the GPT-OSS models, GPT-OSS-120B clearly leads GPT-OSS-20B, but the smaller model trails by only 9

percentage points on unit-test success (58.0% vs. 67.0%) and offers an attractive quality–efficiency trade-off.

Claude Sonnet 4.6 sits at the top of the ranking, and its advantage lies in semantic recovery rather than syntactic recoverability. On compilation, it leads GPT-OSS-120B by only 2.0 percentage points (96.0% vs. 94.0%); on unit-test execution, the gap widens to 13.0 percentage points (80.0% vs. 67.0%). The compile-vs-test gap also tightens sharply: 83.3% of Claude's compilable outputs pass the unit tests, compared with 71.3% for GPT-OSS-120B, 69.0% for GPT-OSS-20B, and 16.3% for SLaDe. In other words, Claude's compilable outputs are themselves more likely to be behaviorally correct than those of the other LLMs.

None of the LLMs is a perfect decompiler. For all three, the function compilation rate exceeds the unit-test pass rate: compilation alone would overstate end-to-end decompilation quality, since many outputs are syntactically valid C yet semantically wrong on edge cases, arithmetic, or control flow.

Takeaway

On optimized x86-64 binaries, general-purpose LLMs are the stronger decompilers by a wide margin—Claude Sonnet 4.6 leads at 80.0% unit-test pass rate—while SLaDe retains one practical advantage: lightweight, offline, CPU-only deployment on commodity hardware. Closing this gap will require future task-specific decompilers to overcome both limitations we observed in SLaDe: substantially larger context windows to avoid excluding long-optimized functions, and stronger semantic recovery for the functions they do process.

That said, three caveats bound the scope of the comparison above. First, SLaDe is one of several small seq2seq decompilers trained from scratch on assembly-to-C mapping; other task-specific baselines may narrow the gap, so our claims hold against SLaDe

rather than against this group as a whole. Second, our two metrics—function-level compilation and unit-test pass rate—capture syntactic and behavioral correctness but not output qualities such as readability, idiomatic C structure, or type-recovery fidelity, on which the ranking could differ. Third, the evaluation set comprises 100 short single functions filtered to fit SLaDe's 1024-token input limit; this gives the comparison limited resolution for narrow gaps (Claude Sonnet 4.6 leads GPT-OSS-120B on compilation by only 96.0% vs. 94.0%) and limits how far the findings extrapolate to longer or full-program decompilation. We adopt the HumanEval-Decompile rather than a held-out split of SLaDe's training corpus, ExeBench [208]. This avoids evaluating SLaDe on inputs drawn from its own training distribution, although we cannot verify whether the general-purpose LLMs encountered HumanEval-Decompile or related code during pretraining.

5.7.7 Cross-Task Analysis

Model Type Comparison

Across tasks, task-specific models consistently outperform LLMs on classification-oriented problems, including binary similarity and type inference. These tasks require precise decision boundaries over a fixed label space, where discriminative training provides a strong advantage. Even with stronger models such as Claude Sonnet 4.6, LLMs still lag behind supervised baselines.

In contrast, LLMs perform better on generation-oriented tasks, such as signature prediction, function summarization, and decompilation, where outputs are structured or free-form sequences. Larger, better-aligned models significantly improve performance in these settings.

However, tasks with constrained outputs and strict evaluation metrics present a limitation. In function name prediction and summarization, normalized formats and token-overlap metrics (e.g., BLEU, ROUGE) favor exact matches. This penalizes LLMs—even

when outputs are semantically correct but differently structured—while benefiting supervised models trained on the target distribution.

Overall, model effectiveness depends not only on task type, but also on output constraints and evaluation design. LLMs perform well with flexible outputs, but struggle when exact token-level conformity is required.

Effect of Model Scale

Increasing model scale improves LLM performance across all tasks, but the effect is not uniform. Moving from GPT-OSS-20B to 120B yields consistent but modest gains, particularly for classification-oriented tasks such as binary similarity and type inference, where the gap with task-specific models remains large. Even with a stronger model like Claude Sonnet 4.6, improvements do not fully close this gap, indicating that scale and alignment cannot replace explicit decision boundary learning.

In contrast, scale and model quality have a much larger impact on generation-oriented tasks. Claude demonstrates a significant improvement over both GPT-OSS models, producing more coherent, semantically accurate outputs. This effect is most pronounced in decompilation, where larger models substantially improve functional correctness.

Overall, scale enhances performance most when tasks align with generative reasoning and semantic reconstruction, but has limited impact on tasks requiring precise, closed-set classification.

Performance Across The Task Spectrum

Performance trends follow a clear pattern across the spectrum of binary analysis tasks, ranging from *classification* to *structured prediction* to *full generation*.

For classification-oriented tasks, LLM performance is strongly affected by the size of the label space. In binary similarity detection (2 classes), LLMs remain reasonably competitive, with Claude Sonnet 4.6 reaching 87.93% F1 compared to 92.66% for jTrans, significantly narrowing the gap relative to smaller models. However, as the label space grows, performance degrades sharply. In type inference (45+ classes), even Claude achieves only 0.308 F1 compared to 0.915 for SeeType, highlighting the difficulty of fine-grained classification.

This limitation is most evident in function name prediction. Although generative in formulation, the task effectively behaves as large-scale classification with a highly skewed label space (50K+ labels). Here, task-specific models dominate, while LLMs—including Claude—remain far behind, reflecting challenges in selecting with precision across large, imbalanced output spaces.

For generation-oriented tasks, LLMs are a better fit. Larger, better-aligned models, particularly Claude, produce more coherent, semantically accurate outputs. This is most evident in decompilation, where LLMs substantially outperform task-specific models, and in signature prediction, where the gap is significantly reduced.

However, binary function summarization presents an exception. Although generative, the task is highly constrained, requiring short outputs with a normalized vocabulary. Metrics such as BLEU and ROUGE favor exact token matches, benefiting supervised models. In contrast, under semantic metrics (e.g., cosine similarity), LLMs—especially Claude—perform competitively or better, indicating stronger semantic understanding. This suggests LLMs are well-suited when semantic quality is prioritized, and standard metrics may underestimate their performance.

Overall, these results show that **task formulation and output space characteristics play a critical role in model performance**. LLMs struggle with large, structured, and imbalanced label spaces, but excel as tasks shift toward open-ended reasoning and generation.

Key Observations

- **Task formulation dominates performance.** Model effectiveness is primarily determined by whether the task is *discriminative* or *generative*. Task-specific models excel in closed-set classification, while LLMs perform better as tasks shift toward open-ended generation and reasoning.

- **Scale does not replace supervision.** Increasing model size improves LLM performance, but does not bridge the gap for classification tasks. Explicit supervision remains critical for learning precise decision boundaries.

- **Label space size is a key bottleneck.** LLM performance degrades significantly as the output space grows (e.g., type inference, function name prediction), especially under skewed distributions. Large, imbalanced label spaces favor supervised models.

- **Generative alignment benefits LLMs.** Tasks aligned with sequence generation (e.g., signature prediction, function summarization, decompilation) allow LLMs—especially stronger models like Claude Sonnet 4.6—to leverage reasoning and produce semantically coherent outputs.

- **Not all generation tasks favor LLMs.** In constrained settings (e.g., binary summarization with normalized outputs), supervised models perform better under token-based metrics. However, LLMs often match or exceed them under semantic metrics (e.g., cosine similarity), indicating stronger underlying semantic understanding.

- **Transition point exists.** Structured prediction tasks represent a middle ground where LLMs become competitive, but still lack the consistency and precision of supervised models.

5.8 Discussion

This study highlights that model performance in binary analysis is governed not only by scale or supervision, but by how well the modeling paradigm aligns with task structure and output constraints.

5.8.1 Scale vs Supervision

Our results confirm that scale alone is insufficient to replace task-specific supervision. While larger LLMs improve consistently, they remain limited in tasks requiring precise decision boundaries. Supervised models retain a clear advantage in such settings, whereas LLMs excel when tasks involve semantic reasoning and flexible generation. Additionally, LLMs can provide natural-language rationales and chain-of-thought style outputs, which may improve interpretability [209] during analysis workflows.

5.8.2 Label Space and Output Constraints

Performance differences are strongly influenced by the structure of the output space. As label spaces grow or become more constrained, LLM performance degrades due to reliance on autoregressive token prediction rather than explicit class separation. In contrast, supervised models remain robust under large or imbalanced label distributions.

5.8.3 Task Formulation Matters More Than Task Type

The results reinforce that task formulation is more critical than nominal task type. Tasks that appear generative but impose strict output constraints behave similarly to classification problems and favor supervised models. LLMs are most effective when outputs are open-ended and allow semantic flexibility.

5.8.4 Role of Dataset and Evaluation Design

Dataset construction and evaluation metrics significantly shape observed performance. Balanced datasets increase difficulty for classification tasks, while skewed distributions favor pattern exploitation. Similarly, token-based metrics can undervalue semantically correct LLM outputs, particularly in generation tasks.

5.8.5 Practical Implications

- Prefer task-specific models for closed-set prediction and large label spaces.
- Prefer LLMs for reasoning-heavy and open-ended generation tasks.
- Hybrid approaches combining LLM reasoning with structured prediction are a promising direction.

5.8.6 Limitations and Future Work

This study has several limitations that open directions for future work.

First, while we evaluate a diverse set of binary analysis tasks, the space of such tasks is much broader. Important problems such as vulnerability detection, control-flow recovery, variable tracking, and data structure reconstruction are not included in this study. These tasks may exhibit different behaviors with respect to model scale and supervision, and evaluating them would provide a more comprehensive understanding of the trade-offs.

Second, due to resource constraints, we evaluate two open-source LLMs (20B and 120B) and one commercial model (Claude Sonnet 4.6). While Claude provides insight into the performance of modern, well-aligned LLMs, it is evaluated only on limited subsets (e.g., 128 samples) due to cost constraints. As a result, our evaluation does not fully capture the behavior of large commercial models at scale.

Third, the datasets used in our experiments are relatively small, and in some cases further reduced for commercial models. This is primarily due to computational and cost constraints. Larger and more diverse datasets, along with full-scale evaluation of frontier models, could yield more robust and generalizable conclusions, particularly for complex tasks such as decompilation and type inference.

Fourth, we employ a limited set of prompting strategies for LLM evaluation. More advanced techniques—such as chain-of-thought prompting, few-shot examples, or structured decoding—may improve LLM performance. However, due to time constraints and the need to maintain consistency across tasks, we restrict our experiments to a controlled prompting setup.

Finally, there exists an inherent evaluation bias between the two model paradigms. Task-specific models are trained on large labeled datasets tailored to the target task and often evaluated on distributions closely aligned with their training data. In contrast, LLMs are evaluated in a zero-shot setting without task-specific adaptation. This asymmetry may favor supervised models in certain tasks. Developing more balanced evaluation frameworks for comparing generative and discriminative approaches remains an important direction for future research.

5.9 Conclusion

In this work, we set out to understand which class of models—task-specific supervised models or large general-purpose LLMs—is better suited for different binary analysis tasks. Through systematic evaluation across a spectrum of problems, we find that performance is not determined solely by model scale or supervision, but also by factors such as task formulation and the structure of the prediction space.

Our experiments show that, for many binary analysis tasks, particularly those that are classification-oriented or involve large, constrained label spaces, task-specifically trained models remain highly effective and often necessary. In contrast, LLMs perform well on

tasks that require reasoning, abstraction, and open-ended generation, such as decompilation.

These results suggest that no single model paradigm universally dominates. Instead, the choice of model should be guided by the nature of the task. Practitioners must consider the problem formulation, output space, and desired level of precision when selecting an approach.

Overall, our findings highlight that **model scale improves reasoning, while supervision ensures precision**, and the balance between the two depends on the specific binary analysis problem being addressed.

Chapter 6

Future Work

The framework developed in this dissertation advances type inference for stripped binaries, but it also exposes two natural directions for extending the work. The primary direction looks *beyond* the primitive types targeted in this dissertation, toward recovering the richer, open-ended abstractions found in C++ binaries; this is the direction we have begun to pursue, and whose fact-extraction stage we have already started to implement. A second, more preliminary direction looks *upstream* of type inference, at the disassembly stage on which every later analysis depends: improving disassembly reliability would strengthen the entire pipeline, including the type recovery developed here. We describe the C++ direction first and in the most detail, followed by the proposed disassembly work.

6.1 Inferring High-Level Types in C++ Binaries

6.1.1 Motivation

A natural extension of our type inference work is to recover higher-level type abstractions from C++ binaries. Compared to C, C++ introduces substantial additional complexity through its object-oriented features, including classes, templates, inheritance, and virtual dispatch. Inferring even primitive types from stripped binaries is already challenging; recovering the layout and relationships of user-defined classes from type-agnostic assembly is considerably harder.

A key difficulty is that high-level abstractions are not drawn from a small, fixed vocabulary. Unlike primitive types, which fall within a finite and well-defined set, the space of user-defined classes and their layouts is effectively unbounded—there are practically infinite variations in how classes are defined, composed, and used. This is precisely the property that makes the transformer-based, classification-style models we used for primitive type inference a poor fit for class layout recovery. A model that predicts a label from a finite vocabulary cannot, in general, emit an arbitrary, program-specific class definition. Recovering a class layout is closer to a structured generation problem: the output is a record describing the size of an object, the offsets and types of its fields, the methods defined on it, the location of its virtual function table, and its position in an inheritance hierarchy.

Because of this mismatch, the most accurate systems for C++ abstraction recovery today are not learning-based but rely on formal, rule-based reasoning. OOAnalyzer [32] is a representative example. It first uses a lightweight symbolic binary analysis to extract a set of low-level *facts* about the program—object pointer allocations, method invocations on object pointers, member accesses at fixed offsets, virtual function table installations, and similar evidence—and then feeds these facts to a Prolog-based reasoning engine. The reasoning engine applies hand-written inference rules that encode C++ and compiler domain knowledge, makes educated guesses to resolve ambiguous properties, and backtracks when its guesses lead to contradictions, ultimately producing a consistent model of the program's classes and methods. Its predecessor, ObjDigger [31], performed similar recovery using procedurally written analysis code, but became difficult to extend as the reasoning grew more complex.

The strength of this approach is that fact extraction is grounded in concrete, observable program behavior, and the reasoning step is sound and interpretable. Its weakness is that the reasoning rules are written and tuned by hand. Encoding C++ and compiler knowledge as Prolog rules requires significant expert effort, the rule set is tightly coupled

to a particular compiler and ABI, and extending the system to new patterns, optimizations, or platforms is laborious.

6.1.2 Proposed Approach

Our goal is to retain the part of this design that works well—grounded, dataflow-based fact extraction from the binary—while replacing the hand-written reasoning step with a large language model. In other words, we keep a fact exporter that observes low-level program behavior, but instead of deducing the class model with a fixed set of Prolog rules, we provide the extracted facts to an LLM and let it reason about the most likely class layout. This is motivated directly by our evaluation of model paradigms (Chapter 5): LLMs are weakest on closed-set classification but strongest on open-ended reasoning and structured generation, which is exactly the regime that class layout recovery falls into. The reasoning that OOAnalyzer performs—combining many weak, sometimes contradictory pieces of evidence using domain knowledge and educated guessing—is the kind of task LLMs are comparatively well suited for, and an LLM can draw on its pretrained knowledge of C++ and compiler conventions rather than requiring that knowledge to be manually encoded.

Extracting Facts

The first stage of the proposed pipeline collects evidence about objects and their usage directly from the binary, mirroring the categories of information that formal tools rely on but that an LLM would otherwise lack. The signals we plan to extract include:

- **RTTI and the class hierarchy.** When present, runtime type information directly reveals class names and base classes. Because RTTI is optional and only describes polymorphic classes, it is treated as strong but incomplete evidence.

- **Virtual function tables and call sites.** The presence, location, and contents of virtual function tables, together with the sites where virtual calls are dispatched, provide evidence of polymorphism and inheritance.
- **Object layout clues from `this` pointer usage.** The most general source of evidence comes from tracking the object pointer. Functions that receive an object pointer (typically through the `this`-pointer calling convention) and the offsets they access reveal both class membership and field layout.
- **Cross-references and usage.** Allocation sites, the methods invoked on each allocated object, and the propagation of object pointers across functions tie individual methods to the classes they belong to.

Identifying the `this` Pointer and Field Offsets

Reliably identifying the object pointer is central to fact extraction, since most layout and membership evidence is expressed relative to it. We plan to identify candidate object pointers through dataflow analysis over register and memory accesses of the form `[reg + offset]`. A register that is dereferenced at multiple distinct offsets (for example `0x0`, `0x18`, and `0x28`) within a function is a strong candidate for the object pointer, and the set of offsets observed against it gives the field offsets of the class. A second, complementary signal is the installation of a virtual table pointer in a constructor or destructor: a pattern that materializes the address of a vtable and stores it into `[this + offset]` (often at offset zero) both identifies the object pointer and locates the vtable within the object. For example, an instruction sequence that loads a vtable address and writes it to `[rdi+0]` indicates that `rdi` holds the object pointer and that the vtable resides at offset zero, while the Microsoft ABI uses `rcx` for the same role.

Reasoning with an LLM

Once facts are extracted, they are assembled into a structured description and presented to the model. The model is asked to group methods into classes, infer the fields and their offsets, locate virtual tables, and reconstruct inheritance relationships, producing a class signature for each recovered class. Conceptually, this replaces OOAnalyzer's forward reasoning, hypothetical reasoning, and consistency checking with a single model that weighs the evidence and emits the most plausible consistent layout. We expect the model to benefit from grouping methods that share an object pointer, clustering allocation sites with similar signatures into the same class, and inferring inheritance from shared vtable hierarchies, constructor chaining, and matching field prefixes.

6.1.3 Current Pipeline

We have begun implementing the fact-extraction stage of this pipeline. At a high level, the current pipeline proceeds as follows:

1. Detect object allocations by locating calls to allocation routines (e.g., `operator new`) and recording the requested allocation size.
2. Identify the methods called on the object pointer returned from each allocation site.
3. Detect constructors and, within them, locate virtual table installations.
4. Extract field accesses of the form `[this + offset]` within each method to recover the object's layout.
5. Build a class signature for each object from its constructor, size, vtable address, methods, and fields.
6. Cluster allocation sites with similar signatures into classes, and infer inheritance from vtable hierarchies, constructor chaining, and field-prefix matches.

Scope

Following the scope adopted by formal recovery tools, we focus on classes whose implementation is present in the binary. This includes user-defined classes as well as library classes (such as STL containers) whose code is embedded in the executable through templating or static linking. For dynamically linked external classes whose implementations reside in separate libraries, only a minimal understanding can be formed from relocation and linking information, so these are out of scope for full layout recovery.

Ground Truth

To train and evaluate the approach, we construct ground truth from debugging information. We compile C++ programs with debug symbols and parse the resulting DWARF records to recover, for each class, its size, the offsets and types of its members, its methods, and its base classes. We distinguish classes that are locally defined in the program under analysis from those pulled in from system headers and libraries, so that evaluation can focus on the classes of interest. As with our earlier type inference work, debugging information is used only to establish ground truth and is never provided to the model.

Evaluation Considerations

Evaluating recovered class layouts raises several ambiguities that we must account for, many of which also arise in formal recovery systems. A member function that does not actually use its object can be indistinguishable from an ordinary free function at the binary level, and a class method that is linked into the executable but never invoked may be impossible to attribute to any class. Aligning recovered classes with the ground truth is itself nontrivial, since there is not always a clean identifier connecting a recovered class to a source-level class; the alignment must instead be established through the addresses of member functions and the membership of methods and data members. We plan to adopt

evaluation strategies that tolerate these ambiguities, such as measuring how much the recovered method-to-class assignment must be edited to match the ground truth.

Preliminary Observations

On small toy programs containing a mix of polymorphic and non-polymorphic classes (for example, a logger, an account, and a sensor class, alongside non-class utility functions), the current fact-extraction stage is able to recover, for each allocation site, the object size, the constructor, the virtual table pointer when present, the set of methods invoked on the object, and the accessed field offsets. These recovered facts form the input that the reasoning model will consume. The immediate next step is to integrate the LLM-based reasoning stage and evaluate end-to-end class layout recovery against the DWARF-derived ground truth, and then to extend the approach from toy programs to the larger, real-world C++ binaries that motivate this work.

6.2 Improving Disassembly with Free-Flow Disassembly

While the type inference pipeline developed in this dissertation operates on preprocessed assembly, its accuracy is ultimately bounded by the quality of the underlying disassembly. Disassembling stripped binaries remains difficult because symbols are absent and code is interleaved with data, and no state-of-the-art tool guarantees complete and correct disassembly. Since disassembly is the first stage of any binary analysis pipeline, errors here propagate downstream and degrade type inference, control-flow recovery, and decompilation. As a more preliminary direction for future work, we therefore propose *free-flow disassembly*, a technique that augments probabilistic disassembly with machine-learning-driven hints that are independent of control flow.

Existing disassemblers fall into a few broad families. *Linear sweep* decodes instructions sequentially without following control flow [210], as in tools such as OBJDUMP [211],

PSI [212], and UROBOROS [213], but it readily misinterprets embedded data as code. *Recursive descent* follows control flow from known entry points, giving strong correctness but poor coverage of indirectly reached code; tools such as Ghidra [2], Dyninst [214], ANGR [23], BAP [63], Radare2 [215], and Datalog [216] recover additional code through prologue matching and cross-references at some cost to precision. *Machine-learning* methods such as XDA [217] and DEEPDI [218] improve robustness across compilers and optimization levels.

Probabilistic disassembly [219] frames instruction recovery as statistical inference over candidates. It builds on *superset disassembly* [220], which decodes an instruction at *every* byte offset, guaranteeing *zero false negatives* but producing many false positives. To filter the superset while preserving this property, probabilistic disassembly scores candidates using structural *hints*—control-flow convergence, control-flow crossing, and register define-use relations—that are unlikely to arise by chance, then propagates these scores to select the most probable non-overlapping subset.

Despite these gains, probabilistic disassembly still yields a notable false positive rate, because its hints are entirely *structural*: instructions lacking any convergence, crossing, or def-use relation must be retained by default. We therefore propose enriching the hint space with cues independent of control flow—for example, predicting the likelihood that a candidate is a true instruction from the *instruction sequence* it produces, and exploiting the *alignment* regularities of x86 code. Because these hints can be computed for nearly every candidate, they sharply reduce the number of hintless instructions and, with them, the false positive rate. This control-flow-independent approach—*free-flow disassembly*—builds directly on superset and probabilistic disassembly and aims to deliver cleaner, more semantically consistent instruction streams that strengthen the type inference pipeline and other downstream analyses such as binary rewriting and decompilation.

References

[1] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 690–702. [Online]. Available: https://doi.org/10.1145/3468264.3468607

[2] C. Eagle and K. Nance, *The Ghidra Book: The Definitive Guide*. No Starch Press, 2020.

[3] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. USA: No Starch Press, 2011.

[4] V. 35, "Binary ninja," https://github.com/Vector35/binaryninja-api.

[5] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, may 2016. [Online]. Available: https://doi.org/10.1145/2896499

[6] Z. Lin, J. Li, B. Li, H. Ma, D. Gao, and J. Ma, "Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2725–2739. [Online]. Available: https://doi.org/10.1145/3576915.3623214

[7] Z. Sha, H. Shu, H. Wang, Z. Gao, Y. Lan, and C. Zhang, “Hyres: Recovering data structures in binaries via semantic enhanced hybrid reasoning,” vol. 35, no. 3. New York, NY, USA: Association for Computing Machinery, Feb. 2026. [Online]. Available: https://doi.org/10.1145/3736719

[8] Y. Wang, R. Liang, Y. Li, P. Hu, K. Chen, and B. Zhang, “Typeforge: Synthesizing and selecting best-fit composite data types for stripped binaries,” in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 1–18.

[9] M. Noonan, A. Loginov, and D. Cok, “Polymorphic type inference for machine code,” *SIGPLAN Not.*, vol. 51, no. 6, p. 27–41, jun 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908119

[10] I. Smith, “Binsub: The simple essence of polymorphic type inference for machine code,” *arXiv preprint arXiv:2409.01841*, 2024.

[11] J. Bosamiya, M. Woo, and B. Parno, “Trex: Practical type reconstruction for binary code,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6897–6915.

[12] C. Ye, Y. Cai, A. Zhou, H. Huang, H. Ling, and C. Zhang, “Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024, pp. 170–187.

[13] A. Mycroft, “Type-based decompilation (or program reconstruction via type reconstruction),” in *European Symposium on Programming*. Springer, 1999, pp. 208–223.

[14] A. Retdec, “Retdec github repository,” https://github.com/avast/retdec.

[15] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, “Debin: Predicting debug information in stripped binaries,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[16] H. Green, E. J. Schwartz, C. L. Goues, and B. Vasilescu, “Stride: Simple type recognition in decompiled executables,” *arXiv preprint arXiv:2407.02733*, 2024.

[17] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, “Augmenting decompiler output with learned variable names and types,” 2021.

[18] J. Escalada, T. Scully, and F. Ortin, “Improving type information inferred by decompilers with supervised machine learning,” *ArXiv*, vol. abs/2101.08116, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231648247

[19] A. Cozzie, F. Stratton, H. Xue, and S. T. King, “Digging for data structures,” in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA: USENIX Association, Dec. 2008. [Online]. Available: https://www.usenix.org/conference/osdi-08/digging-data-structures

[20] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, nov 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[22] M. J. Eager, “Introduction to the dwarf debugging format,” 2012.

[23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[24] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, ser. CERIAS '10. West Lafayette, IN: CERIAS - Purdue University, 2010.

[25] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 255–265.

[26] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Network and Distributed System Security Symposium*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:43281

[27] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos, "Dsibin: Identifying dynamic data structures in c/c++ binaries," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 331–341.

[28] C. Jung and N. Clark, "Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 56–66.

[29] I. Haller, A. Slowinska, and H. Bos, "Mempick: High-level data structure detection in c/c++ binaries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 32–41.

[30] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. I. Urbina, “Artiste: Automatic generation of hybrid data structure signatures from binary code executions,” 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:54749001

[31] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, “Recovering c++ objects from binaries using inter-procedural data-flow analysis,” in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, ser. PPREW’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2556464.2556465

[32] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, “Using logic programming to recover c++ classes and methods from compiled executables,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 426–441.

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

[34] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, “Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 813–832.

[35] J. Choi, K. Kim, D. Lee, and S. K. Cha, “Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 677–693.

[36] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, “Typeminer: Recovering types in binary programs using machine learning,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 288–308.

[37] Z. Xu, C. Wen, and S. Qin, “Type learning for binaries and its applications,” *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 893–912, 2019.

[38] L. Chen, Z. He, and B. Mao, “Cati: Context-assisted type inference from stripped binaries,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 88–98.

[39] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC’17. USA: USENIX Association, 2017, p. 99–116.

[40] D. Lehmann and M. Pradel, “Finding the dwarf: Recovering precise types from webassembly binaries,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 410–425. [Online]. Available: https://doi.org/10.1145/3519939.3523449

[41] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, “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*, 2024, pp. 4554–4568.

[42] L. Dramko, C. L. Goues, and E. J. Schwartz, “Idioms: Neural decompilation with joint code and type prediction,” 2025.

[43] X. Wang, X. Xu, Q. Li, M. Yuan, and J. Xue, “Recovering container class types in c++ binaries,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 131–143.

[44] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé *et al.*, “Tygr: Type inference on stripped binaries using graph neural networks.”

[45] C. Stewart, R. K. Gaede, and J. H. Kulick, “Dragon: Predicting decompiled variable data types with learned confidence estimates,” *Workshop on Binary Analysis Research (BAR)*, 2025.

[46] G. Li, X. Shang, S. Cheng, J. Zhang, L. Hu, X. Zhu, W. Zhang, and N. Yu, “Beyond the edge of function: Unraveling the patterns of type recovery in binary code.” New York, NY, USA: Association for Computing Machinery, Mar. 2026, just Accepted. [Online]. Available: https://doi.org/10.1145/3787853

[47] V. Srinivasan and T. Reps, “Recovery of class hierarchies and composition relationships from machine code,” in *International Conference on Compiler Construction*. Springer, 2014, pp. 61–84.

[48] K. Dolgova and A. Chernov, “Automatic type reconstruction in disassembled c programs,” in *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008, pp. 202–206.

[49] D. H. White, T. Rupprecht, and G. Lüttgen, “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*, 2016, pp. 259–269.

[50] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[51] X. Rival and K. Yi, *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[52] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[53] ——, “Abstract interpretation frameworks,” *Journal of logic and computation*, vol. 2, no. 4, pp. 511–547, 1992.

[54] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *Compiler Construction*, E. Duesterwald, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 5–23.

[55] ——, “Wysinwyx: What you see is not what you execute,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[56] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[57] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[58] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[59] D. A. Ramos and D. Engler, “{Under-Constrained} symbolic execution: Correctness checking for real code,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[61] M. J. Van Emmerik, *Static single assignment for decompilation*. University of Queensland, 2007.

[62] E. J. Schwartz, C. F. Cohen, and S. Schwartz, “Systematic testing of c++ abstraction recovery systems by iterative compiler model refinement,” in *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering*, 2025, pp. 1–16.

[63] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 463–469.

[64] J. Rehof and M. Fähndrich, “Type-base flow analysis: from polymorphic subtyping to cfl-reachability,” *ACM SIGPLAN Notices*, vol. 36, no. 3, pp. 54–66, 2001.

[65] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, “Codesurfer/x86—a platform for analyzing x86 executables,” in *International conference on compiler construction*. Springer, 2005, pp. 250–254.

[66] J. Lim and T. Reps, “Tsl: A system for generating abstract interpreters and its application to machine-code analysis,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 1, pp. 1–59, 2013.

[67] M. Fahndrich and A. Aiken, *Making set-constraint program analyses scale*. University of California at Berkeley, 1996.

[68] S. Dolan and A. Mycroft, “Polymorphism, subtyping, and type inference in mlsub,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 60–72.

[69] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, “Class-based n-gram models of natural language,” *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[70] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, “Razzer: Finding kernel race bugs through fuzzing,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[71] R. Team, “The official radare 2 book,” 2017.

[72] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[73] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, “Sok: Demystifying binary lifters through the lens of downstream applications,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1100–1119.

[74] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[75] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[76] K. O’Shea and R. Nash, “An introduction to convolutional neural networks,” *ArXiv*, vol. abs/1511.08458, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:9398408

[77] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” Tech. Rep., 1985.

[78] M. I. Jordan, “Serial order: A parallel distributed processing approach,” in *Advances in psychology*. Elsevier, 1997, vol. 121, pp. 471–495.

[79] M. Sundermeyer, R. Schlüter, and H. Ney, “Lstm neural networks for language modeling.” in *Interspeech*, vol. 2012, 2012, pp. 194–197.

[80] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[81] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[82] J. R. Quinlan, “Induction of decision trees,” *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

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

[84] G. Salton, “Introduction to modern information retrieval,” *McGraw-Hill*, 1983.

[85] A. Parmar, R. Katariya, and V. Patel, “A review on random forest: An ensemble classifier,” in *International conference on intelligent data communication technologies and internet of things (ICICI) 2018*. Springer, 2019, pp. 758–763.

[86] Y. Zhang, “Support vector machine classification algorithm and its application,” in *Information Computing and Applications: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings, Part II 3*. Springer, 2012, pp. 179–186.

[87] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Machine learning*, vol. 63, pp. 3–42, 2006.

[88] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 282–289.

[89] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *International Conference on Learning Representations*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:5959482

[90] L. Chen, Z. He, Y. Qian, and B. Mao, “Cati++: empirical study and evaluation for adjacent instruction enhanced type inference.” Oxford University Press, 2025.

[91] L. R. Medsker, L. Jain *et al.*, “Recurrent neural networks,” *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.

[92] K. Cao and K. Leach, “Revisiting deep learning for variable type recovery,” in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 275–279.

[93] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, L. Màrquez, C. Callison-Burch, and J. Su, Eds. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421. [Online]. Available: https://aclanthology.org/D15-1166

[94] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[95] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:11212020

[96] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, “A robustly optimized BERT pre-training approach with post-training,” in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, S. Li, M. Sun, Y. Liu, H. Wu, K. Liu, W. Che, S. He, and G. Rao, Eds. Huhhot, China: Chinese Information Processing Society of China, Aug. 2021, pp. 1218–1227. [Online]. Available: https://aclanthology.org/2021.ccl-1.108

[97] R. Arefin, K. Baker, and S. Mulder, “From bytes to types: Enhancing type prediction in executables with transformer-based binary analysis tool seetype,” *IEEE Access*, vol. 13, pp. 212 591–212 607, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:282620401

[98] J. D. M.-W. C. Kenton and L. K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of naacL-HLT*, vol. 1. Minneapolis, Minnesota, 2019, p. 2.

[99] M. Weiser, “Program slicing,” *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[100] C. Cifuentes and A. Fraboulet, “Intraprocedural static slicing of binary executables,” in *1997 Proceedings International Conference on Software Maintenance*, 1997, pp. 188–195.

[101] R. Vaidya and P. A. Kulkarni, “Transformers can do it: Recovering types from executables using transformer based llms,” *International Conference on Information Systems Security and Privacy*, 2026.

[102] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *ArXiv*, vol. abs/1910.01108, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:203626972

[103] A. Thawani, J. Pujara, P. A. Szekely, and F. Ilievski, “Representing numbers in NLP: a survey and a vision,” *CoRR*, vol. abs/2103.13136, 2021. [Online]. Available: https://arxiv.org/abs/2103.13136

[104] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, “Big code!= big vocabulary: Open-vocabulary models for source code,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1073–1085.

[105] A. Madsen and A. R. Johansen, “Neural arithmetic units,” *arXiv preprint arXiv:2001.05016*, 2020.

[106] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[107] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” *arXiv preprint arXiv:1810.00826*, 2018.

[108] M. H. Austern, *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[109] J. Tian, W. Xing, and Z. Li, “Bvdetector: A program slice-based binary code vulnerability intelligent detection system,” *Information and Software Technology*, vol. 123, p. 106289, 2020.

[110] H. Xue, G. Venkataramani, and T. Lan, “Clone-slicer: Detecting domain specific binary code clones through program slicing,” in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018, pp. 27–33.

[111] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[112] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-flow integrity: Precision, security, and performance,” *ACM Comput. Surv.*, vol. 50, no. 1, apr 2017. [Online]. Available: https://doi.org/10.1145/3054924

[113] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[114] K. Zhao, Z. Li, W. Chen, X. Luo, T. Chen, G. Meng, and Y. Zhou, “Recasting type hints from webassembly contracts,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729388

[115] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[116] R. O’Callahan and D. Jackson, “Lackwit: A program understanding tool based on type inference,” in *Proceedings of the (19th) International Conference on Software Engineering*. IEEE, 1997, pp. 338–348.

[117] ClamAV, “ClamAV anti-virus system,” http://www.clamav.net, [Online]. Available: http://www.clamav.net.

[118] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, “Dire: A neural approach to decompiled identifier naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[119] H. Tan, Q. Luo, J. Li, and Y. Zhang, “Llm4decompile: Decompiling binary code with large language models,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 3473–3487.

[120] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, “Differentiating code from data in x86 binaries,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[121] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 24–35. [Online]. Available: https://doi.org/10.1145/2931037.2931047

[122] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, pp. 358–366, 1953. [Online]. Available: https://api.semanticscholar.org/CorpusID:120980829

[123] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[124] B. Wang, L. Shang, C. Lioma, X. Jiang, H. Yang, Q. Liu, and J. G. Simonsen, "On position embeddings in bert," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=onxoVA9FxMw

[125] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks," *arXiv preprint arXiv:1612.02295*, 2016.

[126] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang *et al.*, "Big bird: Transformers for longer sequences," *Advances in neural information processing systems*, vol. 33, pp. 17 283–17 297, 2020.

[127] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long range arena : A benchmark for efficient transformers," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=qVyeW-grC2k

[128] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533

[129] J. Zhang, Y. Zhao, M. Saleh, and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 328–11 339.

[130] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 118–127.

[131] J. Tian, W. Xing, and Z. Li, “Bvdetector: A program slice-based binary code vulnerability intelligent detection system,” *Information and Software Technology*, vol. 123, p. 106289, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300392

[132] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, “Well-read students learn better: On the importance of pre-training compact models,” 2020. [Online]. Available: https://openreview.net/forum?id=BJg7x1HFvB

[133] X. Li, Y. Qu, and H. Yin, “Palmtree: Learning an assembly language model for instruction embedding,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[134] S. Ahn, S. Ahn, H. Koo, and Y. Paek, “Practical binary code similarity detection with bert-based transferable similarity learning,” in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 361–374.

[135] A. Madsen and A. R. Johansen, “Neural arithmetic units,” *CoRR*, vol. abs/2001.05016, 2020. [Online]. Available: https://arxiv.org/abs/2001.05016

[136] A. Nayak, H. Timmapathini, K. Ponnalagu, and V. G. Venkoparao, “Domain adaptation challenges of bert in tokenization and sub-word representations of out-of-vocabulary words,” in *Proceedings of the first workshop on insights from negative results in NLP*, 2020, pp. 1–5.

[137] A. B. Siddique, S. Oymak, and V. Hristidis, “Unsupervised paraphrasing via deep reinforcement learning,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1800–1809. [Online]. Available: https://doi.org/10.1145/3394486.3403231

[138] R. Mohammed, J. Rawashdeh, and M. Abdullah, “Machine learning with oversampling and undersampling techniques: Overview study and experimental results,” in *2020 11th International Conference on Information and Communication Systems (ICICS)*, 2020, pp. 243–248.

[139] X.-Y. Liu, J. Wu, and Z.-H. Zhou, “Exploratory undersampling for class-imbalance learning,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 2, pp. 539–550, 2009.

[140] E. Bendersky, “pyelftools,” https://github.com/eliben/pyelftools.

[141] C. I. Security, “miasm,” https://github.com/cea-sec/miasm.

[142] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Q. Liu and D. Schlangen, Eds. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[143] Z. Zhang, “Improved adam optimizer for deep neural networks,” in *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*. Ieee, 2018, pp. 1–2.

[144] K. Pei, “StateFormer: Fine-grained type recovery from binaries using generative state modeling,” https://github.com/CUMLSec/stateformer, 2023, gitHub repository, accessed June 26, 2026.

[145] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[146] Y. Gu, H. Shu, and F. Hu, “Uniasm: Binary code similarity detection without fine-tuning,” *arXiv preprint arXiv:2211.01144*, 2022.

[147] H. Gao, S. Cheng, Y. Xue, and W. Zhang, “A lightweight framework for function name reassignment based on large-scale stripped binaries,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.

[148] M. Ali, M. Fromm, K. Thellmann, R. Rutmann, M. Lübbering, J. Leveling, K. Klug, J. Ebert, N. Doll, J. Buschhoff, C. Jain, A. Weber, L. Jurkschat, H. Abdelwahab, C. John, P. Ortiz Suarez, M. Ostendorff, S. Weinbach, R. Sifa, S. Kesselheim, and N. Flores-Herr, “Tokenizer choice for LLM training: Negligible or crucial?” in *Findings of the Association for Computational Linguistics: NAACL 2024*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 3907–3924. [Online]. Available: https://aclanthology.org/2024.findings-naacl.247

[149] G. Dagan, G. Synnaeve, and B. Roziere, “Getting the most out of your tokenizer for pre-training and domain adaptation,” *arXiv preprint arXiv:2402.01035*, 2024.

[150] T. Ye, L. Wu, T. Ma, X. Zhang, Y. Du, P. Liu, S. Ji, and W. Wang, “CP-BCS: Binary code summarization guided by control flow graph and pseudo code,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 14 740–14 752. [Online]. Available: https://aclanthology.org/2023.emnlp-main.911

[151] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[152] M. Lewis, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” *arXiv preprint arXiv:1910.13461*, 2019.

[153] M. Schuster and K. Nakajima, “Japanese and korean voice search,” in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 5149–5152.

[154] R. Sennrich, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[155] T. Kudo, “Subword regularization: Improving neural network translation models with multiple subword candidates,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych and Y. Miyao, Eds. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75. [Online]. Available: https://aclanthology.org/P18-1007

[156] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, “Typeminer: Recovering types in binary programs using machine learning,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 288–308.

[157] L. Chen, Z. He, and B. Mao, “Cati: Context-assisted type inference from stripped binaries,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 88–98.

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

[159] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, “{DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1787–1804.

[160] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116.

[161] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 611–626.

[162] A. Thawani, J. Pujara, F. Ilievski, and P. Szekely, “Representing numbers in NLP: a survey and a vision,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Online: Association for Computational Linguistics, Jun. 2021, pp. 644–656. [Online]. Available: https://aclanthology.org/2021.naacl-main.53

[163] A. Thawani, J. Pujara, and F. Ilievski, “Numeracy enhances the literacy of language models,” in *Proceedings of the 2021 conference on empirical methods in natural language processing*, 2021, pp. 6960–6967.

[164] J. Oliver, C. Cheng, and Y. Chen, “Tlsh–a locality sensitive hash,” in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 2013, pp. 7–13.

[165] T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilic, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. Villanova del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, and et al., “Bloom: A 176b-parameter open-access multilingual language model,” *CoRR*, vol. abs/2211.05100, 2022. [Online]. Available: https://arxiv.org/abs/2211.05100

[166] F. Stollenwerk, “Training and evaluation of a multilingual tokenizer for gpt-sw3,” *arXiv preprint arXiv:2304.14780*, 2023.

[167] P. Rust, J. Pfeiffer, I. Vulić, S. Ruder, and I. Gurevych, “How good is your tokenizer? on the monolingual performance of multilingual language models,” in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 3118–3135. [Online]. Available: https://aclanthology.org/2021.acl-long.243

[168] L. Li, L. Song, S. Ding, B. C. Fung, and P. Charland, “Transforming generic coder llms to effective binary code embedding models for similarity detection,” in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[169] X. Shang, G. Chen, S. Cheng, B. Wu, L. Hu, G. Li, W. Zhang, and N. Yu, “Binmetric: a comprehensive binary code analysis benchmark for large language models,” in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, ser. IJCAI ’25, 2025. [Online]. Available: https://doi.org/10.24963/ijcai.2025/858

[170] B. Wan, S. Wang, Z. Wei, J. Huang, and C. Hu, “Binary code similarity detection via llm-based source code conversion,” *IEEE Internet of Things Journal*, vol. 12, no. 24, pp. 51 842–51 853, 2025.

[171] L. Jiang, X. Jin, and Z. Lin, “Beyond classification: Inferring function names in stripped binaries via domain adapted llms,” in *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[172] Z. Sha, H. Wang, Z. Gao, H. Shu, B. Zhang, Z. Wang, and C. Zhang, “llasm: Naming functions in binaries by fusing encoder-only and decoder-only llms,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, Apr. 2025. [Online]. Available: https://doi.org/10.1145/3702988

[173] H. Tan, Q. Luo, J. Li, and Y. Zhang, “LLM4Decompile: Decompiling binary code with large language models,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 3473–3487. [Online]. Available: https://aclanthology.org/2024.emnlp-main.203/

[174] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, “Bindeep: A deep learning approach to binary code similarity detection,” *Expert Systems with Applications*, vol. 168, p. 114348, 2021.

[175] X. Jin, K. Pei, J. Y. Won, and Z. Lin, “Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1631–1645.

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

[177] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[178] C. Eagle, *The IDA pro book*. no starch press, 2011.

[179] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.” in *NDSS*, vol. 5, 2005, pp. 3–4.

[180] F. Jelinek, “Self-organized language modeling for speech recognition,” *Readings in speech recognition*, pp. 450–506, 1990.

[181] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[182] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, “Recurrent neural network based language model.” in *Interspeech*, vol. 2, no. 3. Makuhari, 2010, pp. 1045–1048.

[183] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[184] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[185] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[186] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[187] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.

[188] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, “Emergent abilities of large language models,” *TMLR*, 2022.

[189] D. Ganguli, D. Hernandez, L. Lovitt, A. Askell, Y. Bai, A. Chen, T. Conerly, N. Dassarma, D. Drain, N. Elhage *et al.*, “Predictability and surprise in large generative models,” in *Proceedings of the 2022 ACM conference on fairness, accountability, and transparency*, 2022, pp. 1747–1764.

[190] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[191] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[192] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[193] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Pondé, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *ArXiv*, vol. abs/2107.03374, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235755472

[194] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *ArXiv*, vol. abs/1909.09436, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:202712680

[195] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, “The pile: An 800gb dataset of diverse text for language modeling,” *ArXiv*, vol. abs/2101.00027, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:230435736

[196] Z. Pang, D. Chatterjee, F. Sener, and A. Yao, “On discriminative vs. generative classifiers: Rethinking mllms for action understanding,” *ICLR*, 2026.

[197] S. R. Kasa, K. Gupta, S. Roychowdhury, A. Kumar, Y. Biruduraju, S. K. Kasa, P. N. Priyatam, A. Bhattacharya, S. Agarwal, and V. Huddar, “Generative or discriminative? revisiting text classification in the era of transformers,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 9615–9637.

[198] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “Jtrans: Jump-aware transformer for binary code similarity detection,” in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 1–13.

[199] H. Kim, J. Bak, K. Cho, and H. Koo, “A transformer-based function symbol name inference model from an assembly language for binary reversing,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 951–965. [Online]. Available: https://doi.org/10.1145/3579856.3582823

[200] A. Mostafa, R. Nahid, and S. Mulder, “How different tokenization algorithms impact llms and transformer models for binary code analysis,” *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 11 2025, co-located with NDSS 2025.

[201] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 7871–7880.

[202] T. Ye, L. Wu, T. Ma, X. Zhang, Y. Du, P. Liu, S. Ji, and W. Wang, “Cp-bcs: Binary code summarization guided by control flow graph and pseudo code,” in *Conference on Empirical Methods in Natural Language Processing*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264490533

[203] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O’Boyle, “Slade: A portable small language model decompiler for optimized assembly,” in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 67–80. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444788

[204] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou, “Qwen3 embedding: Advancing text embedding and reranking through foundation models,” *arXiv preprint arXiv:2506.05176*, 2025.

[205] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

[206] H. Tan, Q. Luo, J. Li, and Y. Zhang, “LLM4Decompile: Decompiling binary code with large language models,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Miami, Florida, USA: Association for Computational Linguistics, 2024, pp. 3473–3487. [Online]. Available: https://aclanthology.org/2024.emnlp-main.203/

[207] OpenRouter, “Create a chat completion,” https://openrouter.ai/docs/api/api-reference/chat/send-chat-completion-request, 2026, accessed 2026-04-15.

[208] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. 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*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 50–59. [Online]. Available: https://doi.org/10.1145/3520312.3534867

[209] R. Arefin, M. D. Samad, F. A. Akyelken, and A. Davanian, “Non-transfer deep learning of optical coherence tomography for post-hoc explanation of macular disease classification,” in *2021 IEEE 9th International Conference on Healthcare Informatics (ICHI)*. IEEE, 2021, pp. 48–52.

[210] C. Pang, T. Zhang, R. Yu, B. Mao, and J. Xu, “Ground truth for binary disassembly is not easy,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2479–2495.

[211] GNU Project, *GNU objdump*, Free Software Foundation, 2024, version 2.12. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_6.html

[212] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, “A platform for secure static binary instrumentation,” in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2014, pp. 129–140.

[213] S. Wang, P. Wang, and D. Wu, “Reassembleable disassembling,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[214] A. R. Bernat and B. P. Miller, “Anywhere, any-time binary instrumentation,” in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011, pp. 9–16.

[215] “radare2,” https://github.com/radareorg/radare2.

[216] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1075–1092.

[217] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, “Xda: Accurate, robust disassembly with transfer learning,” *arXiv preprint arXiv:2010.00770*, 2020.

[218] S. Yu, Y. Qu, X. Hu, and H. Yin, “{DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[219] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, “Probabilistic disassembly,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1187–1198.

[220] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, “Superset disassembly: Statically rewriting x86 binaries without heuristics.” in *NDSS*, 2018.