IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Type Recovery in Stripped Binaries: A Comprehensive Survey of Inference Tools and Methods

RAISUL AREFIN[1], RYAN VRECENAR[2] SAMUEL MULDER[3]
[1]Auburn University, Auburn, AL 36849 USA (e-mail: ran0013@auburn.edu)
[2]Sandia National Laboratories, Albuquerque, NM 87123 USA (e-mail: rvrecen@sandia.gov)
[3]Auburn University, Auburn, AL 36849 USA (e-mail: szm0211@auburn.edu)

ABSTRACT Understanding binary programs is challenging due to the loss of high-level abstractions during compilation. Type inference plays a key role in recovering information such as variable types, data structures, and class hierarchies, which is crucial for reverse engineering, decompilation, and security analysis. This paper presents a survey of 39 binary type inference tools. We categorize the tools based on the types they recover and the methods they use, including dynamic analysis, static reasoning, symbolic execution, and machine learning. We also compare their input formats, supported languages, and evaluation strategies. In addition, the survey discusses the scope and origins of the area, its evolution over the past two decades, and the challenges that lie ahead.

Our study highlights recent progress, especially in learning-based methods, but also reveals ongoing limitations. These include limited scalability, lack of standardized output formats, and poor support for dynamically typed languages. We also observe a lack of user-friendly interfaces and limited availability of source code for many tools, which hinders adoption and further development. We conclude by outlining open research problems and recommending future directions to make type inference tools more accurate, accessible, and widely applicable.

INDEX TERMS Dynamic Analysis, Reverse Engineering, Static Analysis, Type Inference

I. INTRODUCTION

Software development is hard. A major goal of software engineering is to make the development process easier, correct, manageable, and efficient. To achieve these, most software development techniques rely on abstraction and the divide-and-conquer paradigm. These strategies may appear to introduce unnecessary complexity in some scenarios, and developers may at times avoid them for simplicity. However, they are fundamentally integrated into most modern development workflows. These techniques are designed to make the role of programmers more manageable. But often they make the life of a reverse engineer worse as these extra layers of abstraction often make a program more difficult to reverse.

At the machine level, there are no variables, data structures, or types—only raw 32- or 64-bit values stored in physical locations like 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 a CPU register. Among them, the CPU registers are the fastest but usually few in number. In cases the CPU has enough registers, the compiler might optimize the program to store all the local variables in then. In other cases, it spills 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 might need to persist even after the called function exits, which results in the termination of the functions stack memory. Sometimes the compiler will evaluate a variable as a constant and those variables get no real physical storage at all. Furthermore, some variables may

be unused and that can result in its complete absence in the executable. As we can see, 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. We attempt to study all the tools that have type prediction as a major feature. We have found 39 such tools. Our observation shows a variety of methods used by the tools and they also 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, and Binary Ninja 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 some on intermediate representations. This paper is structured in a way 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.

II. PROBLEM DEFINITION

The objective of type inference is to recover the type of variables used by the developer analyzing the information obtained from the executable. 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 provides 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.

III. SCOPE

A significant body of work has been conducted in the area of type inference for binaries. A previous survey on this topic, titled "Type Inference on Executables" [1], provided a comprehensive overview of research up to 2016. However, nearly a decade has passed since its publication, and many new, high-quality contributions have emerged. Additionally, several tools integrate type inference as a component of broader binary analysis pipelines, even if it is not their primary focus. To maintain a focused and manageable scope, we include tools where type inference is either the central objective or constitutes a substantial part of the methodology. We intentionally excluded works that, while related in spirit, do not substantially engage with the type inference problem itself. Some widely used tools are closed-source or lack formal documentation. In such cases, we relied on secondary sources, including research papers, project websites, tutorials, and community discussions, to understand their approaches. In total, we reviewed 49 tools for this study.

Early tools primarily relied on dynamic analysis. Over time, there has been a shift toward static approaches, particularly with the introduction of formal methods such as abstract interpretation, constraint solving, and logic-based reasoning. More recently, machine learning has become prominent—especially graph-based models and language models. Some tools now combine formal analysis with learning-based methods in hybrid pipelines. In this survey, we categorize tools into these major groups: Dynamic Analysis, Formal Methods, Learning Based Methods, and Hybrid Approaches.

IV. 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]–[4] or improve [5]–[8] the decompilation process from binaries. Decompiled code is a great way to understand the functionality of a binary and quality type information can improve the readability of the decompiled code significantly. Type information can also assist in malware and software vulnerability detection. Exe-

cutables containing function signatures, and data structures known to be used by known malware can be more accurately identified [9]. Type information is also essential for binary rewriting [10] which can be required 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 the existing code. Types have also proven to be useful for improving Control-Flow Integrity [10]. For example, TypeSqueezer utilized type information for improving their Control-Flow Integrity algorithm to prevent forward-edge attacks [11].

V. BACKGROUND

A. DEBUG INFORMATION: GROUND TRUTH FOR TYPE INFERENCE

Debug information is supplementary data that is included in a compiled executable which assists 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 [12]. 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. For obvious reasons, most closed-source programs are delivered without debugging information. Two popular debug formats are DWARF, used in Linux, and PDB, used in Windows programs. Debug information is crucial for creating the ground truth for type prediction task.

B. 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, VEX used by Angr, and Microcode used by IDA Pro.

C. 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 due to these differences. While Intermediate Representations can help mitigate some of these challenges by abstracting away those 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 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, public documentation, and uncertainty related to the architecture's design principle. 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 due to reasons 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 complexity, as the same source code can result in different instructions depending on the compiler.

VI. A TAXONOMY OF TYPE INFERENCE METHODS

VII. TYPE INFERENCE METHODOLOGIES

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.

A. 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*).
- **Flow-Based Analysis**: Tracks data-flow during execution to propagate type information (e.g., *Rewards*, *DynCompB*).
- **Memory Access Pattern Analysis**: Uses patterns in dereferencing and memory layout to infer structures (e.g., *Howard*, *DSIBin*).
- **Memory Graph Analysis**: Constructs dynamic graphs from memory allocations and links to recover structural types (e.g., *DDT*, *MemPick*, *ARTISTE*).

B. 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*, *Binary Ninja*, *angr*, *Retypd*).
- **Symbolic Execution**: Simulates program paths with symbolic inputs to infer constraints and behavior (e.g., *angr*, *OBJDIGGER*).
- **Forward Reasoning / Logic Inference**: Applies inference rules over extracted facts using logical reasoning (e.g., *OOAnalyzer*).
- **Constraint Solving**: Extracts type constraints from code and solves them using SMT or custom solvers (e.g., *TIE*, *Retypd*, *TRex*, *BinSub*, *Manta*).
- **Statistical Models**: Uses probabilistic reasoning to infer types from heuristics and data patterns (e.g., *OSPREY*, *Stride*).
- **Type Propagation**: Propagates known type information through call graphs or data flows (e.g., *NTFUZZ*).

C. LEARNING-BASED METHODS

Recent tools increasingly adopt machine learning for type inference. These include traditional classifiers (e.g., *TypeMiner*, *DEBIN*, *BITY*, *Escalada et al.*), CNN-based models (e.g., *Cati*), sequence models using RNNs or Transformers (e.g., *EKLAVYA*, *DIRTY*, *StateFormer*, *SnowWhite*, *ReSym*, *IDIOMS*), and graph neural networks (e.g., *TIARA*, *TYGR*, *DRAGON*, *ByteTR*).

D. HYBRID METHODS

Hybrid tools combine dynamic and static techniques to leverage the strengths of both. Examples include *TypeSqueezer*, *HyRES*, and *TypeForge*.

VIII. DYNAMIC ANALYSIS TECHNIQUES FOR BINARY TYPE INFERENCE

Dynamic analysis refers to analyzing the behavior ofsoftware while it is running. It can run in a physical environment or a virtual or emulated environment. Dynamic analysis can be more precise because the program is actually run and some of its true behavior can be observed. Dynamic analysis can mitigate many of the uncertainties regarding memory addresses, operand values, and control-flow targets as they are known during execution. However dynamic analysis has a few serious flaws. One of the biggest limitations of dynamic analysis is incomplete coverage of the code. This is because only the code that gets executed is analyzed, which may leave out a significant portion of the program. Moreover, many malicious programs employ evasion tactics to identify that they are executing in a sandbox and will stop showing their true behavior. This could worsen the situation leaving out the most important chunk of the code being analyzed. Code coverage is not a type inference task, but having good coverage of disassembly is a prerequisite to any binary analysis task. One reason dynamic approaches are used despite their limited coverage could be that many applications only require partial typing, focusing on specific data structures or functions of interest rather than entire programs. A lot of type inference tools, most of which came out a decade ago or earlier used dynamic analysis, such as LEGO [13], Mempick [14], ARTISTE [15], Howard [16], TIE [17], Rewards [18], DDT [19], Dc [20], Laika [9], DynCompB [21]. These dynamic analysis tools can be further categorized based on their specific methodologies as:

A. 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 [22] is a dynamic analysis tool that uses memory images and probabilistic methods to predict object types. It uses dynamic analysis to figure out 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 is likely a pointer. 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, and the authors use it as a malware detection tool.

B. FLOW-BASED TYPE INFERENCE: TRACKING RUNTIME DATA FLOWS TO INFER TYPES

This method assigns types to variables based on how the program uses them during execution.

Rewards [23] uses type propagation in a dynamic setting. During execution, it looks for operations that reveal type information of variables. The actions they look for are 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 [21] is a dynamic analysis tool that infers abstract types. Abstract types are a semantic abstraction over more typically source level types. Basically, they identify 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.

C. 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 [16] identifies data structures by looking at the pattern of memory access during runtime. How memory is accessed 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 [24] is a binary extension of the former work DSI [25] 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.

D. 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 [19] dynamically monitors the organization of data in the memory of an application 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 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. This state is used to determine the invariants of the function calls [26]. Function invariants are program properties that are unchanged throughout the execution of an application and can be used as a signature of a function. These three things (memory graph, interacting functions, and invariants) are then matched against a library of known data structures to identify the data structure under examination.

MemPick [14] uses dynamic analysis to generate a memory graph by first organizing the heap buffer allocation and internal linkages. The graph illustrates how links between heap objects evolve during the execution of an application. MemPick detects the collections of similar objects in the memory graph. For example, if the memory graph contains a tree which has 5 nodes, then there should be 5 similar objects in the graph, which are the nodes. 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. The links between the nodes in a binary tree will be different than the links between a doubly linked list.

ARTISTE [15] recovers 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 uses multiple executions to generate the memory graph. It performs analysis on the memory graph and with the recovered primitive types, tries to recover the data structures containing them. This work is unique from other dynamic analysis-based type tools is that it tries to generate both high-level data structures and also primitive constituent types.

E. OTHER METHODS

LEGO [13] 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 derives a fingerprint from the destructor sequence of each object-trace, which encodes inheritance depth and ancestor cleanup order. These fingerprints are organized into a trie structure and refined with method-call information, allowing Lego to infer class hierarchies, identify member functions, and detect composition relationships.

IX. STATIC ANALYSIS TECHNIQUES FOR TYPE INFERENCE

Static analysis methods try to understand how a program behaves without running it [53]. 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 [54], [55], which builds a simpler version of the program that still keeps important information. In abstract interpretation, the program's actual behavior is approximated using an abstract model, which helps analyze it safely [43]. Based on this model, static analysis 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.

A. VALUE SET ANALYSIS

Value-set analysis (VSA) was first introduced by [56], [57] which is used for analyzing the contents of the memory. It introduces the concepts of memory regions, which are 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. Abstract interpretation, in turn, is a method of program analysis that creates a simplified model of the program to predict its behavior without executing it.

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 aggrigate structures as well. VSA is used by tools like Ghidra, Binary Ninja, angr, TIE, BITY, and Retypd.

B. SYMBOLIC EXECUTION-BASED METHODS

Symbolic execution is a technique for analyzing programs by simulating their execution using symbolic inputs instead of concrete values. It explores how a program behaves for a range of inputs that share a specific execution path. Instead of running the program with actual values, symbolic execution uses symbolic variables to represent input values and produces outputs or constraints expressed in terms of these symbolic variables.

How Symbolic Execution Works: Let's consider the example function listed below.

```
1  def example(a, b):
2      if a > 0:
3          c = b + 1
4      else:
5          c = b - 1
6      assert c != 2
```

Inputs as Symbols: Instead of giving specific values to a and b (like `a=1, b=2`), symbolic execution treats them as symbolic variables, such as : `a = x , b = y.`

Tracking Execution Paths: Symbolic execution tracks the program's behavior based on symbolic values. If a branch condition ever depends on unknown symbolic values, the symbolic execution engine simply chooses one branch to take, recording the condition on the symbolic values that would lead to that branch. After a given symbolic execution is complete, the engine may go back to the branches taken and explore other paths through the program.

Path 1: If `a > 0` (i.e., `x > 0`), then `c = y + 1`.
Path 2: If `a <= 0` (i.e., `x <= 0`), then `c = y - 1`.

Checking Assertions: The program has an assertion: `c!=2`. With symbolic execution, we can check this assertion for each path:

Path 1 `(x > 0)`:
- `c = y + 1`
- The assertion `c != 2` becomes `y+1 != 2, or y != 1.`

Path 2 `(x <= 0)`:
- `c = y - 1`
- The assertion `c != 2` becomes `y-1 != 2, or y != 3.`

Findings: Symbolic execution determines the conditions under which the assertion is violated:

- If Path 1 is taken and y = 1, the assertion fails.
- If Path 2 is taken and y = 3, the assertion fails.

We have solved the symbolic variables manually here but there are methods such as SMT solvers [58] for solving these constraints through formal methods.

OBJDIGGER [45] employs symbolic execution to recover object-oriented structures from C++ binaries compiled with MSVC. For the recovery of object structures and relationships, they developed an approach that leverages the use of the ThisPtr, a reference assigned to each unique object instance. When objects are created, the compiler allocates space in memory for each class instance. The amount of space

allocated is based on the number and size of data members and possibly padding for alignment. Every instantiated object is referenced by a pointer to its start in memory; this reference is commonly referred to as the ThisPtr. It essentially points to the memory address where an object's data begins. In some compilers, when methods associated with an object are called, ThisPtr is typically passed as a hidden argument to these methods, allowing them to access and manipulate the object's member data and functions. The paper describes how ThisPtr can be tracked through the flow of a program to deduce relationships between methods and the structure of the object, such as identifying constructors, methods, data members, and inheritance hierarchies. They use symbolic execution and static inter-procedural dataflow analysis [59] to track individual ThisPtr propagation and usage between and within functions. It uses symbolic execution to understand how object fields and class relationships are structured in binary code. It observes how ThisPtrs are used to access memory, allowing it to infer the layout of data members within a class . It also tracks how virtual function tables are assigned and used, enabling recovery of virtual methods and their positions. Additionally, by following how ThisPtrs are passed and modified across constructor calls, OBJDIGGER can detect inheritance relationships between classes.

angr [29] is a binary analysis infrastructure built on VSA and symbolic execution [60]–[62]. In this angr paper [29], it is stated that angr uses VSA for variable recovery and their implementation is similar to the variable recovery in TIE [17]. However, their current implementation has a new approach for variable recovery [63]. The variable recovery API [64] provided by angr takes a function as input and performs data-flow analysis. Then it runs concrete execution on every statement and tracks all register/memory accesses to discover all places that are accessing variables. This analysis follows Static Single Assignment (SSA) [65]. For type information recovery, angr currently uses a system that simplifies and solves type constraints [66]. Their type constraints are largely an implementation of the Retypd [41] tool.

C. 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 applies inference rules to derive new facts. Each inference rule consists of a set of **preconditions** and a **conclusion**. The process works as follows:

1) **Initialization**: Start with the given facts and inference rules.
2) **Rule Application**: Check whether the preconditions of any inference rule are satisfied by the current set of facts. If so, add the rule's conclusion to the fact set.
3) **Iteration**: Repeat the process, applying inference rules to the updated fact set to derive additional facts.
4) **Termination**: Continue until no new facts can be generated from the current facts and rules.

This iterative reasoning approach enables logical deduction by systematically expanding the known facts using the inference rules.

OOAnalyzer [44] is an approach to recover C++ classes and methods from stripped executables using Logic Programming. OOAnalyzer uses a lightweight symbolic analysis to efficiently generate an initial set of low-level facts that form the basis of their reasoning called fact-base. With the low-level facts, they perform forward reasoning; reasoning about the program by matching a built-in set of rules over facts in the retrieved low-level facts. However, sometimes OOAnalyzer cannot reach new forward reasoning conclusions before important properties of the program are resolved. To continue making progress in these scenarios, OOAnalyzer identifies an ambiguous property and makes an educated guess about it, which is called hypothetical reasoning. During this, whenever OOAnalyzer detects an inconsistency in the current fact base it backtracks and systematically revisits the earlier guesses that have been made, starting with the most recent one. Consistency checks are implemented by a special set of rules that detect contradictions instead of asserting new facts. When all facts are consistent, the model then generates and provides the final output to the user.

D. TYPES INFERENCE VIA CONSTRAINT GENERATION AND SOLVING

Type constraints for type inference are generated by analyzing how variables and functions interact within a program. The idea is that those interactions will reveal some properties about those variables which will be stored as constraints. For example, if a variable is used in arithmetic operations, constraints will specify that it must be a numeric type. Solving these constraints yields a set of type assignments that best satisfy all observed behaviors.

TIE [17] by Lee et al. is a principled type inference tool that integrates both static and dynamic analysis. It is built on the BAP framework [27], 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 [41], developed by Noonan et al., applies a static, constraint-based type inference approach capable of handling structure and polymorphic types [67]. It operates on an intermediate representation (IR) produced by GrammaTech's CodeSurfer [68]. 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 [69]. They also include the pre-computed type information obtained from externally linked functions in this stage. The constraints are resolved using a simplification algorithm based on [70], after which the inferred type information is translated into C-like types.

BinSub [52] builds on Retypd by replacing its subtyping mechanism with algebraic subtyping [71], 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 [42] 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. Their 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 for recursive and nested structures.

Manta [46] pointed out 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.

E. STATISTICAL MODELS FOR RECOVERING TYPE INFORMATION

OSPREY is a probabilistic variable and data structure recovery technique. Compilation is a lossy process and formulating general rules to recover variables is very difficult. It tried to address the uncertainty of variable information recovery from binaries using random variables and probabilistic constraints. It defined different hints that can help predicting the variable information, such as data-flow hints that takes data-dependency into consideration. It also defined hints for memory access patters that considers the fact that fields of the same data structures would be accessed in a unified manner. Another kind of hint it came up with is points-to hint which relates two variables by the pointers they are pointed to. Afterwards, it constructs probabilistic constraints where the predicates describe the structural and type properties of memory chunks, which are denoted by random variables. Then it solves these probabilistic constraints to predict the variable information.

Stride [6] 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 [72], 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.

F. TYPE PROPAGATION (TYPE SOURCES AND SINKS)

NTFUZZ [43] presented a static analysis method to infer system call signatures on the Windows operating system. NTFUZZ is a type-aware kernel fuzzing [73] 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.

G. HYBRID TOOLS: COMBINING STATIC AND DYNAMIC TECHNIQUES

Lin et al. presented **TypeSqueezer** which 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. In other words, TypeSqueezer inspects 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 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. They call their type recovery method value-based type inference. It does not care much about fine grained types but mostly distinguishes between values, references, references to references, etc. TypeSqueezer utilizes the recovered type information for devising a method for improving Control-Flow Integrity [10], [74] to prevent forward-edege attacks [11]. 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, as it was not their primary task. TypeArmor [75] is another similar tool that infers function signatures for Control-Flow Integrity.

H. 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, IDA Pro, RetDec, Radare2, Binary Ninja, and CMU BAP.

Ghidra 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 [76] 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 [4]. 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 [4], [31]. For example, the pattern for accessing a globar integer array is different than a heap-allocated integer array.

IDA Pro 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 [31]. For type recovery, IDA combines static analysis with a signature matching mechanism (FLIRT) to identify known library functions [77]. Once identified, the type information of these functions is propagated to other variables in the program [3].

RetDec [2], [78], an open-sourced reverse engineering toolchain developed by Avast is a retargetable machine-code decompiler based on LLVM [79]. Given a binary, 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 [80]. Afterward, the obtained IR would be refined and optimized in multiple passes. During those steps, variables are identified with their types. The final IR is converted to high-level C/C++ code including variables and types.

Radare2 has variable detection and type prediction features. However, their predicted types are not the same as the C standard. For example, they will not predict char or short but they will predict int8_t or uint64_t. Radare2's type inference feature relies heavily on matching preloaded function prototypes and calling conventions [81]. It also supports basic Run-Time Type Information(RTTI) recovery based on virtual table parsing.

Binary Ninja initially transforms the instructions to Binary Ninja Intermediate Language (BNIL). Then they perform value set analysis within BNIL for variable discovery [82]. 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)** [27] is another popular suite of utilities and libraries that enables binary program analysis. However, BAP does not support type inference [83]. **Ddisasm** has a type recovery implementation based on Retypd [84].

X. LEARNING-BASED TYPE INFERENCE

Many of the latest type predicting tools have used various kinds of machine learning methods, such as Random Forest Classifiers, Support Vector Machine (SVM), Convolutional Neural Network (CNN) [85], Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), Graph Neural Networks and Transformers. 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 lots of benefits and limitations.

One of the reasons why machine learning tools are a good fit for binary analysis tasks as they can easily generalize, given enough samples. For example, many non-machine learning tools have to cover a lot of 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 of using machine learning models, without a large amount of samples, the models will not work well. There are architectures for which collecting a large number of source code is not always possible. Collecting a large number of binaries and performing analysis on them can be also a resource-intensive task.

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

A. 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), Random Forests, Decision Trees, 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.

TypeMiner is one of the first tools that used machine learning techniques for type prediction. They used a combination of Random Forest classifiers [86] and Linear Support Vector Machine [87] for the prediction task. First, they perform trivial static analysis on the binary and generate 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 they think are 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 operand consists of two parts: the type of the operand and the width of the operand's value. They use Data Dependency Graphs 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 may decide what type it is if it's not a pointer. Then another final classifier may determine if it's signed or not.

BITY [38], [88] also uses classification to recover types, starting with variable discovery using a value-set analysis–inspired method [57]. 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 should provide important information for type recovery. These instruction sets are processed into feature vectors, with Term Frequency-Inverse Document Frequency (TF-IDF) [89] used to retain relevant features. The extracted features are then passed to a classification model combining SVM and Random Forest to infer variable types.

DEBIN [5] 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 [27] and classifies program elements (e.g., registers, memory offsets) using an Extremely Randomized Trees [90] 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) [91], which allows predictions considering the relationship of a node with its neighbors, in this case, context. Then 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. [8] focused on predicting function return types using statistical and machine learning classifiers. During dataset creation, they modified C source functions to insert instrumentation that helps link return statements to function calls. These instrumentations break 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 applied feature selection to extract the most relevant instructions. For classification, they experimented with 14 models, including AdaBoost, Naive Bayes, Decision Trees, Perceptron, Logistic Regression, and multiple SVM variants.

B. CNN-BASED MODELS FOR TYPE PREDICTION

Convolutional Neural Networks (CNNs) [85] 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 [36], 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 their statistical analysis showing that spatially local instructions tend to be more informative. The preprocessed instruction sequence is encoded using Word2Vec [92] 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. They suggested that instructions close to one another usually operate on variables of similar types and propagating type information as a cluster would be beneficial. **Cati++** [37] is an extension of Cati that uses a different embedding algorithm.

C. 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 to models that are built for understanding 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 is one of the first tools to use a deep learning NLP model for recovering type information. They used a Recurrent Neural Network [93] to predict function signature. Unlike most tools that use disassembled instructions or decompiled code, EKLAVYA takes raw bytes of functions as input. However, they used Skip-gram negative sampling [94] method 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, it 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 their model is the count of function arguments and the corresponding types of those arguments.

DIRTY, proposed 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, the authors note that it can work with any decompiler that provides both decompiled code and rich data-layout information.

Building on DIRTY, **Cao and Leach, 2023** [33] 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], [95], 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 [96]. 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 it predicts the type information in a similar way of how type information is presented in 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 [97], and it uses two separate models to predict return types and parameter types of functions, respectively.

StateFormer uses RoBERTa [98], a transformer model originally developed for natural language processing, to predict variable types from disassembled instructions. Since transformer models are not inherently familiar with 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.

ReSym 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 is a neural decompiler that jointly predicts code and user-defined types (UDTs) using a large language model. It leverages neighboring functions from the call graph to provide richer context for type prediction. The model is finetuned with decompiled code plus call graph context as input, and 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.

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 [99]. 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 [100], [101]. 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 [8] also performed normalization on the numeric values. They replaced numeric values with special tokens. Stride [6] also replaced the numbers over 100 into several tokens depending on the magnitude. TypeMiner gets rid of any numeric values during normalization. StateFormer [35] used a different method to handle numeric values. They included the Neural Arithmetic Unit (NAU) [102], which creates embeddings meant to represent the meaning of numbers used in arithmetic operations.

D. GRAPH-BASED NEURAL MODELS FOR TYPE INFERENCE

Graph Convolutional Networks (GCNs) [103], [104] 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 [48] targets the recovery of STL container classes [105] in C++ binaries using a combination of principled and learning-based techniques. It employs inter-procedural forward slicing [106]–[108] 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 [47] is a Graph Neural Network [109]-based tool designed to infer both basic and struct types in stripped binaries. It is architecture-agnostic and operates on VEX IR [110], 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 [49] 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.

E. HYBRID METHODS

ByteTR [50] initiates by performing SSA, Def-Use Analysis, Intra-procedural, 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 Networks (GGNN) for type prediciton.

TypeForge introduce a novel way of type prediciton with a two-stage pipeline: Synthesize multiple candidate composite type declarations using a novel graph abstraction. Select the best-fit type declaration by evaluating how well each improves the readability of decompiled code. TypeForge initates 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. Next Stage 2 of TypeForge selects the best-fit composite type from the candidate pool by evaluating how each affects decompiled code readability. An LLM-assisted double-elimination process performs pairwise comparisons of generated code variants to determine which version is more readable.

HyRES 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 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.

XI. INPUT REPRESENTATIONS USED IN TYPE INFERENCE TOOLS

This section 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, CATI, SnowWhite, Escalada et al.)
- **Decompiled source code:** (e.g., DIRTY, Cao and Leach, STRIDE, DRAGON, RESYM, IDIOMS, TypeForge)
- **Intermediate representations (IR):** (e.g., DEBIN)

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, which uses raw function bytes as input for an RNN model. Although it does not disassemble the bytes, they still contain program semantics.

Other models use non-sequential representations. For example, TYGR constructs a data-flow graph for each function and uses a Graph Neural Network (GNN) to infer types. TypeMiner, 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:** P-code
- **IDA Pro:** Microcode
- **Angr:** VEX
- **BAP:** BIL
- **Binary Ninja:** Low-Level IL (LLIL), Mid-Level IL (MLIL), and High-Level IL (HLIL)
- **DEBIN and TIE:** BAP's BIL
- **Retypd:** CodeSurfer IR [68]

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, Osprey) and program constraints (Retypd, BinSub, Osprey) to inferred types from another tool as in DSIbin.

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** [24]
- **ARTISTE** [15]
- **MemPick** [14]
- **DDT** [19]

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

- **REWARDS** [23] propagates type information based on calls to known library functions.

XII. 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 [111], 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 [112]. 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** and **SnowWhite** 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** 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 [106], [113]. This helps reduce the input size while retaining the essential context. **TIARA** uses *forward slicing* to include all instructions influenced by the target instruction.

VARIABLE-CENTERED GRAPHS WITH K-HOP CONTEXT (DRAGON)

DRAGON 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.

TABLE 2. Languages Supported by Type Inference Tools

Tool	C	C++	WebAssembly
BinSub	✓		
Escalada et al.	✓		
BITY	✓		
CATI	✓		
StateFormer	✓		
DIRTY	✓		
DEBIN	✓		
EKLAVYA	✓		
OSPREY	✓		
TIE	✓		
Howard	✓		
DRAGON	✓		
TRex	✓		
ByteTR	✓		
IDIOMS	✓		
DSIBin	✓	✓	
Stride	✓	✓	
Retypd	✓	✓	
TypeSqueezer	✓	✓	
RESYM	✓	✓	
Manta	✓	✓	
TIARA		✓	
ObjDigger		✓	
OOAnalyzer		✓	
SnowWhite			✓

XIII. 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, the majority of tools focus on C, with several extending support to C++. Tools like TIARA, ObjDigger, and OOAnalyzer are specifically tailored for C++ binaries, while SnowWhite targets WebAssembly binaries that may originate from C, C++, Rust, or Go.

XIV. TYPE COVERAGE: FROM PRIMITIVE TYPES TO RECONSTRUCTED ABSTRACTIONS

Binary type analysis tools serve a range of purposes, covering diverse aspects of type inference. This section 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, TYGR, Escalada et al., Bity, Cati, Stateformer, DEBIN, Eklavya, DRAGON, Manta and TypeMiner 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 focuses on detecting common C++ classes, including those from the Standard Template Library (STL). Similarly, OOAnalyzer identifies C++ classes and their associated methods. DSIBIN specializes in recognizing frequently used data structures such as doubly and singly linked lists, skip lists, and nested structures. Mempick is capable of detecting a variety of dynamic structures, including linked lists, binary trees, and n-ary trees. Additionally, both DDT and DSIBIN contribute to structure identification.

Structure/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 it has 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, Osprey, Rewards, ARTISTE, TIE, OOAnalyzer, Objdigger, TRex, RESYM and Howard 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 recovers similar hierarchies through static analysis. DynCompB predicts whether two variables serve the same high-level purpose based on their usage patterns. Laika attempts to identify similar objects by comparing their behavioral and memory characteristics. Additionally, tools like Binsub and TypeSqueezer focus on recovering function signatures, which can aid in understanding calling conventions and interface reconstruction.

Table 3 summarizes the categorization of these tools based on the type inference functionalities described above.

XV. 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 recover complex structures or class definitions. In this section, we first highlight empirical findings from ByteTR. 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.

A. EMPIRICAL FINDINGS OF BYTETR'S TYPE RECOVERY STUDY

ByteTR [50] performed an empirical study on different properties of type usage over a large set of binaries. They utilized the TYDA 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.

B. 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, StateFormer, Tiara, Rewards, Cati, DIRTY, DEBIN, EKLAVYA, TypeMiner, and OSPREY have reported results using this approach.

C. FINE-GRANED 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. In the following, we discuss several fine-grained accuracy metrics that have been employed by recent type inference tools.

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, BinSub, Retypd, BITY, and ARTISTE—employ variants of this metric to assess their performance. Figure 1 illustrates an example type lattice used by TIE, 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.

Type Prefix Score: It 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 uses this metric to measure their accuracy.

Decision Tree: TRex [42] 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 adopts this metric to evaluate its effectiveness in recovering C++ class structures.

XVI. 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 be a significant bottleneck that limits 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, Ghidra, and Angr, 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.

XVII. CONCLUSION

Type inference from stripped binaries is a foundational problem in reverse engineering, with broad applications in decompilation, vulnerability detection, binary rewriting, and software comprehension. This survey reviewed 39 type inference tools, categorizing them by the types they recover, the methodologies they employ, and the evaluation metrics they

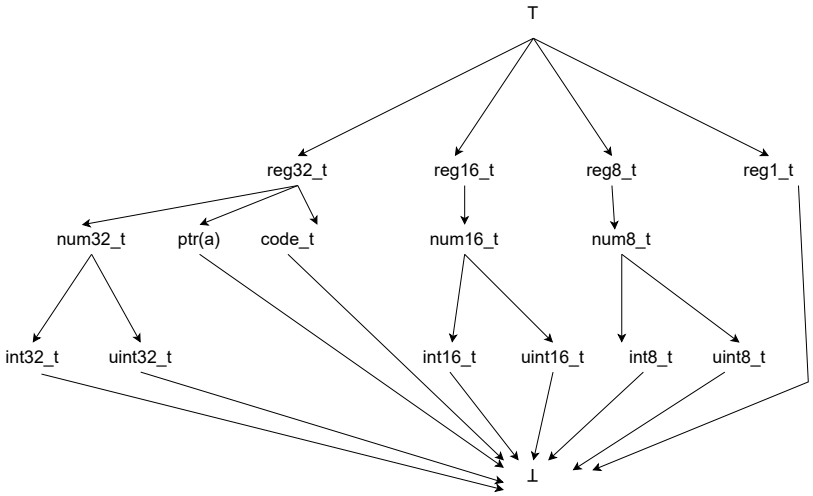

FIGURE 1. Type Distance Metric in Type Lattice: This figure illustrates how types are represented in the type lattice of TIE, showing the accuracy of type inference systems by measuring the distance between inferred and ground-truth types. The closer the predicted type is to the actual type, the more accurate the inference.

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.

REFERENCES

[1] 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

[2] A. Retdec, "Retdec github repository," https://github.com/avast/retdec.

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

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

[5] 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.

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

[7] 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.

[8] 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

[9] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures." in *OSDI*, vol. 8, 2008, pp. 255–266.

[10] 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

[11] 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

[12] M. J. Eager, "Introduction to the dwarf debugging format," 2012.

[13] 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.

[14] 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.

[15] 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

[16] 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

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

[18] 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*, 2010, pp. 1–1.

[19] 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.

[20] 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.

[21] 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.

[22] 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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Z. Sha, H. Shu, H. Wang, Z. Gao, Y. Lan, and C. Zhang, "Hyres: Recovering data structures in binaries via semantic enhanced hybrid reasoning," *ACM Transactions on Software Engineering and Methodology*, 2025.

[35] 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

[36] 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.

[37] L. Chen, Z. He, Y. Qian, and B. Mao, "Cati++: empirical study and evaluation for adjacent instruction enhanced type inference," *The Computer Journal*, p. bxaf004, 2025.

[38] 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.

[39] 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

[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] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 27–41.

[42] J. Bosamiya, M. Woo, B. Parno, and R. Model, "Trex: Practical type reconstruction for binary code," *Reconstruction*, vol. 3, pp. 2–3.

[43] 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.

[44] 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.

[45] 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

[46] 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.

[47] 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."

[48] 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.

[49] C. Stewart, R. K. Gaede, and J. H. Kulick, "Dragon: Predicting decompiled variable data types with learned confidence estimates."

[50] 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," *arXiv preprint arXiv:2503.07243*, 2025.

[51] 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.

[52] I. Smith, "Binsub: The simple essence ofnbsp;polymorphic type inference fornbsp;machine code," p. 425–450, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-74776-2_17

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

[54] 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.

[55] ——, "Abstract interpretation frameworks," *Journal of logic and computation*, vol. 2, no. 4, pp. 511–547, 1992.

[56] 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.

[57] ——, "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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Angr, "Angr github repository," https://github.com/angr/angr/issues/1965.

[64] ——, "Angr github repository," https://api.angr.io/en/v9.2.56/_modules/angr/analyses/variable_recovery/variable_recovery.html.

[65] M. J. Van Emmerik, *Static single assignment for decompilation*. University of Queensland, 2007.

[66] Angr, "Angr github repository," https://github.com/angr/angr/blob/20d3dd168247a3d96bd9651c6eff68b4aafc5f7c/angr/analyses/typehoon/typehoon.py.

[67] 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.

[68] 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.

[69] 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.

[70] M. Fahndrich and A. Aiken, *Making set-constraint program analyses scale*. University of California at Berkeley, 1996.

[71] 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.

[72] 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.

[73] 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.

[74] 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

[75] 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.

[76] Ghidra, "Ghidra github repository," https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Decompiler/src/decompile/cpp/varnode.hh.

[77] H. Rays, "Hex rays documentation," https://hex-rays.com/products/ida/tech/flirt/in_depth/.

[78] J. Křoustek, P. Matula, and P. Zemek, "Retdec: An open-source machine-code decompiler," in *July 2018*, 2017.

[79] 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.

[80] 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.

[81] R. Team, "The official radare 2 book," 2017.

[82] B. Potchik, "Architecture agnostic function detection in binaries," https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html.

[83] C. BAP, "Bap github repository," https://github.com/BinaryAnalysisPlatform/bap/issues/1056.

[84] GrammaTech, "ddisasm github repository," https://github.com/GrammaTech/gtirb-ddisasm-retypd.

[85] 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

[86] 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.

[87] 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.

[88] 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.

[89] G. Salton, "Introduction to modern information retrieval," *McGraw-Hill*, 1983.

[90] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[91] 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.

[92] 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

[93] L. R. Medsker, L. Jain *et al.*, "Recurrent neural networks," *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.

[94] 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

[95] 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

[96] 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

[97] 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

[98] 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

[99] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[100] 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

[101] 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.

[102] A. Madsen and A. R. Johansen, "Neural arithmetic units," *arXiv preprint arXiv:2001.05016*, 2020.

[103] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[104] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[105] M. H. Austern, *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[106] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[107] 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.

[108] 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.

[109] 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.

[110] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[111] 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.

[112] 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.

[113] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *1997 Proceedings International Conference on Software Maintenance*, 1997, pp. 188–195.

Methods Used			
Tool	**Formal Methods**	**Statistical Analysis**	**Machine Learning**
IDA Pro [3]	Code Pattern Identification	✗	✗
Ghidra [4]	VSA, Code Pattern Identification	✗	✗
RetDec [2]	SSA	✗	✗
BAP [27]	✗	✗	✗
Binary Ninja [28]	VSA	✗	✗
angr [29]	VSA, Symbolic Execution	✗	✗
TIE [17]	VSA	✗	✗
TypeMiner [30]	✗	✗	Random Forest classifier
OSPREY [31] ✓		probabilistic constraints	✗
EKLAVYA [32]	✗	✗	Skip-gram, RNN
DEBIN [5]	✓	✓	Randomized Tree Classification
DIRTY [7]	✗	✗	Transformer
Cao and Leach, 2023 [33]	✗	✗	Transformer
ReSym [34]	Constraint Solving	✗	LLM
IDOMS [34]	Call graph	✗	LLM
StateFormer [35]	✗	✗	Transformer
CATI [36], CATI++ [37]	✗	✓	CNN
BITY [38]	VSA	✓	SVM
TypeSqueezer [39]	Dynamic Analysis and Heuristics	✗	✗
SnowWhite [40]	✗	✗	Bidirectional LSTM
Retypd [41]	VSA, Symbolic Variables		✗
TRex [42]	SSA, Constraint Solving		✗
NTFUZZ [43]	Data Flow Analysis	✗	✗
STRIDE [6]	✗	N-gram	✓
Escalada et al. [8]	✗	✓	Classifier Models
OOAnalyzer [44]	Symbolic Analysis, Forward Reasoning	✗	✗
OBJDIGGER [45]	Symbolic Execution	✗	✗
Manta [46]	SSA, Constraint Solving	✗	✗
TYGR [47]	✗	✗	Graph Neural Network
TIARA [48]	Program Slicing	✗	Graph Convolutional Network
DRAGON [49]	✗	✗	Graph Neural Network
ByteTR [50]	SSA, cross-function variable propagation	✗	Gated Graph Neural Network
TypeForge [51]	variable propagation	✗	LLM
HyRES [34]	Static Analysis, Constraint solving	✗	LLM, Transformer
BinSub [52]	VSA, Symbolic Variables	✗	✗

TABLE 1. Overview of Tool Classification by Methodology: This table categorizes tools based on their major methodologies.

TABLE 3. Categorization of Type Analysis Tools by Functionality

Category	Tools
Primitive Type Prediction	Binsub, TYGR, Escalada et al., Bity, Cati, Stateformer, DEBIN, Eklavya, TypeMiner, DRAGON, DIRTY, RESYM, ByteTR, Manta
Structure Identification	Tiara, OOAnalyzer, DSIBIN, Mempick, DDT, DIRTY
Structure/Class Reconstruction	Retypd, Osprey, Rewards, ARTISTE, TIE, OOAnalyzer, Objdigger, Howard, TRex, RESYM
Class Hierarchy Recovery	Lego (dynamic), Objdigger (static)
Variable Role Similarity	DynCompB
Object Similarity	Laika
Function Signature Recovery	Binsub, TypeSqueezer