Type Recovery from Binary Executables: A Comprehensive Survey

Raisul Arefin[1*] and Samuel Mulder[2]

[1*]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.
[2]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.

*Corresponding author(s). E-mail(s): ran0013@auburn.edu;
Contributing authors: szm0211@auburn.edu;

Abstract

The abstract

Keywords: Type Inference, Static Analysis, Dynamic Analysis, Reverse Engineering

1 Introduction

Software development is hard. A major goal of software engineering is to make the process easier, correct, manageable, and efficient. To achieve these, most software development techniques seek refuge in some sort of abstraction and divide-and-conquer. They might seem overhead on the surface and there are ways to bypass them. However, almost everyone uses some variations of these techniques. These techniques are supposed to make the lives of the programmers better. But often they make the life of a reverse engineer worse as these create extra complexities at the low level for him to debunk. Data structures, objects, classes, and even atomic types like integers are some kind of abstractions. Programmers who work with very low-level programming, might not use object-oriented programming but often they can not avoid using structures. However, in the realm of machine code, there are no objects or integers. To understand the true meaning of the intent of the programmer, it is very important to understand the abstractions the programmer used. For example, if the reverse engineer knows that the source of a distance calculation program has two variables, it does

not mean much. Once he knows the program has two integers, it has some meaning. Later, if he knows that the two integers belong to a pair, this gives a better idea, such as that can be a cartesian point. We can see that, the more higher the level of abstraction we can infer, the more useful it becomes to understand the intent of the program. The lifting also gets harder and more convoluted as the higher we want to lift.

Variables are logical storage in high level language. In binaries, there are no variables or high-level types associated with data similar to high level source code. A source variable has some programming language requirements that must be met by source code for it to be successfully compiled, such as variable name and type. An executable is run as a computer process. A process has physical storage for use, which consists of both CPU registers, and addressable memory. The physical storage has no names or types, scope, lifetime, however, hold values. The compiler maps logical variables into the executable's machine code and its physical storage which might include the CPU registers, stack, heap or the data section. 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 numbers. In cases the CPU have 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 calle 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 maybe unused and that can result in its complete absence in the executable. As we can see, it is a very complex method in which the data are mapped.

A lot of work has been done on binary-type information prediction. Most software reverse engineering tools also have type inference features. Besides, there are many other tools that have been solely developed for type prediction. We attempted 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 we got is that recent tools prefer to use other methods than dynamic analysis. Some tools, especially the proprietary reverse engineering tools such as IDA, and Binary Ninja do not describe their type inference method in detail. We studied blogs and books to find out their method. We have categorized and compared their methodology in detail in this paper. We also discuss how different tools produce different levels data of abstraction. For example, some tools just produce primitive types and some tools try to reconstruct C++ class definition. 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 representation. 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.

2 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 problem of type inference is recovering the types without the stored information as most commercial executables are delivered as tripped of those.

The process of recovering types from executables encompasses several tasks. Initially, the executable must be parsed to identify its various components, followed by the disassembly of its instruction sections. However, 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.

3 Scope

A significant amount of work has been done in the field of type inference. There is another survey done on the same topic which was published in 2016 titled "Type Inference on Executables" [1]. This work discussed in detail the work that has been done at that point. However, it is almost a decade old now and a lot of high-quality work has been done afterwards. There is also a lot of work which are not particularly type inference focus, rather they use type inference as a part of their method. To keep our scope of work manageable, we focused on the tools which are either type inference focused, or type inference is a significant part of their method. There are some popular tools that are not open-source or there is no public documentation about their method. We examined their website, other papers that discuss those, tutorials, and other public forums to learn about their method. In total, we have studied 40 tools.

4 Application

Binary type inference is one of the fundamental tasks of binary program understanding. It is crucial to comprehend the types of data the instructions are manipulating 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. Executables containing function signatures, and data structures known to be used by known malware can be looked out for [9]. Besides, type information is essential for binary rewriting which can be required for vulnerability prevention, binary code reuse, and rewriting [10]. 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. Besides, types have 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].

5 Background

5.1 Debug Information

Debug information is the redundant data that is included in a compiled executable which assist in understanding the behaviour of the program during execution. As the name suggests, the intention behind debugging information is debugging a program, ie, setting a breakpoint at the source code and examining the variable values when the program stops there. To achieve this, a lot of crucial information about the source is stored in the debug information. Debug info helps to reverse much of the compiler's carefully crafted transformations, converting the program's data and state back into the 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 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 and PDB. Debug information is crucial for creating the ground truth for type prediction task.

5.2 IR

Generally, Intermediate Representation is a representation of the source code used by compilers. Compilers convert source codes into IRs and then perform a variety of transformations on them. In this way, compilers can use the same transformations on source codes from different languages. However, in the context of reverse engineering, IR serves a similar purpose. Executables from different architectures are converted to IRs and the RE tools can apply the same analysis on the IR regardless of their architecture. However, like any other reverse engineering task, IR has weaknesses. A few examples are P-Code by Ghidra, VEX-IR by Angr, and Microcode by IDA Pro.

5.3 Binary Variants

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 raw 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, or the removal of

unused code. While these changes improve runtime efficiency, they also increase complexity, making the recovery of the original program structure and type information significantly harder.

Target Architecture (e.g., x86, ARM, RISC-V) The choice of architecture determines the instruction set, endianness, and calling conventions, among other factors. Moreover, some architectures may employ unique or proprietary instruction sets, further complicating the analysis process.

Compiler Tool (e.g., GCC, Clang, MSVC) Compilers translate source code into machine code differently, 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.

6 Methods

We have seen a wide variety of methods used in this field. Our analysis also showed how the methods used by tools have gradually changed and adopted over time. The tools can be divided into two primary categories on the basis of their procedure. These are dynamic analysis and static analysis. A common trend that can be seen is that dynamic analysis was very popular in the beginning but later static analysis caught up. And for tools that primarily rely on static analysis, we can find increasing use of deep learning methods for type inference. But surprisingly, none of the popular re tools seems to trust machine learning methods. Rather they rely on more principled methods. The rest of this section categorizes and summarizes the tools under survey according to their methodology of producing type information.

7 Dynamic Analysis Tools

Dynamic analysis refers to analysing the behaviour of a software while it is running. It can run in a real environment or a simulated environment. Dynamic analysis is pricise because the program is actually run and its true behaviour can be observed. It eliminates much of the uncertainties that come with static analysis. Dynamic analysis can mitigate much 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 flaws of dynamic analysis is incomplete coverage of the code. This is because, the code that are executed are only analysed which might leave out a large chunk of code. Moreover, many malicious programs emply evasion tactics to identify that they are executing in a sandbox and will stop showing their true behavior. This would 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 the 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 the entire program. A lot of type inference tools, mostly 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:

7.1 Value-Based Type Inference

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, it is likely a pointer.

Laika is a dynamic analysis tool that uses memory images and probabilistic methods to predict object types. They use 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. Later 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. Then they argue that this technique can be used to find similarities between two programs by looking at their memory image, thus using it as a malware detection tool.

7.2 Flow-Based Type Inference

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

Rewards 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, that knowledge is used to produce the types of the variable used as parameters. Type-revealing instructions are instructions that reveal type information about its operands. For example, floating-point instructions 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 the abstract types from binaries. Abstract types are different than usual source code types or abstractions. Rather this is a kind of semantic abstraction. Basically, they find the 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. They use flow-based analysis to find interaction between variables and decide 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.

7.3 Memory Access Pattern

Howard identifies data structure 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. Similarly, if X is a function frame pointer then, *(X+4) likely is a parameter and *(X-8) is likely a local variable. Howard analyzes both stack and heap for the data structures. It tries to identify the individual fields of a structure but members never accessed during the dynamic run can never be recovered.

DSIBin [22] is a binary extension of the former work DSI [23] which operates on C source code. It analyzes binaries by integrating with another type inferring tool 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. Then It refines the type information obtained by leveraging the core algorithm of DSI.

7.4 Analysing Memory Graph

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 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 on the function calls [24]. 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 uses dynamic analysis to generate a memory graph by first organizing the heap buffer allocation and their links between them. The graph illustrates how links between heap objects evolve during the execution of an application. Then 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 detect and isolate different data structures. To identify the type of the data structures, MemPick has a hand-crafted decision tree. What the decision tree does is it looks at the property of

the data structure and classify 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. The decision tree basically classifies the data-structures based on those properties.

ARTISTE recovers data structures along with the primitive types constituting the data structure. They also perform 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 x86 should be of type `num32`. Function type sinks are the functions whose signature and return type are publicly known, such as function calls. ARTISTE also generates the memory graph from multiple executions. It performs analysis on the memory graph and with the recovered primitive types, it tries to recover the data structures containing the primitive types. This work is unique from the other dynamic analysis-based type tools is that it tries to generate both high-level data structures and also primitive types that constitute them.

7.5 Other Methods

LEGO uses dynamic analysis to recover the class hierarchies and Composition Relationships of the program. This tool is related to type inference but it has no feature to identify object. But this can be useful if used with other object detecting tools.

8 Static Analysis

Static program analysis is a methodology for automatically predicting the behavior of programs without running them [43]. Static analyses use a general theoretical framework named abstract interpretation [44, 45]. Abstract interpretation is a method where a concrete program state is approximated with an abstract domain, and a program is analyzed with abstract semantics, which subsumes the concrete semantics of the program [37].

8.1 Value Set Analysis

Value-set analysis (VSA) was first introduced by [46, 47] 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 Retyped.

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 [25]	✗	✗	✗
Binary Ninja [26]	VSA	✗	✗
angr [27]	VSA, Symbolic Execution	✗	✗
TIE [17]	VSA	✗	✗
TypeMiner [28]	✗	✗	Random Forest classifier
OSPREY [29] ✓		probabilistic constraints	✗
EKLAVYA [30]	✗	✗	Skip-gram, RNN
DEBIN [5]	✓	✓	Randomized Tree Classification
DIRTY [7]	✗	✗	Transformer
StateFormer [31]	✗	✗	Transformer
CATI [32]	✗	✓	CNN
BITY [33]	VSA	✓	SVM
TypeSqueezer [34]	Dynamic Analysis and Heuristics	✗	✗
SnowWhite [35]	✗	✗	Bidirectional LSTM
Retyped [36]	VSA, Symbolic Variables		✗
NTFUZZ [37]	Data Flow Analysis	✗	✗
STRIDE [6]	✗	N-gram	✓
Escalada et al. [8]	✗	✓	Classifier Models
TIARA [38]	Program SLicing	✗	Graph Convolutional Network
OOAnalyzer [39]	Symbolic Analysis, Forward Reasoning	✗	✗
OBJDIGGER [40]	Symbolic Execution	✗	✗
TYGR [41]	✗	✗	Graph Neural Network
BinSub [42]	VSA, Symbolic Variables	✗	✗
SeeType []	Program Slicing	✗	BERT

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

8.2 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[48] for solving these constraints through formal methods.

Angr [27] is a binary analysis infrastructure built on VSA and symbolic execution [49–51]. In this Angr paper [27], 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 [52]. The variable recovery API [53] 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) [54]. For type information recovery, Angr currently uses a system that simplifies and solves type constraints [55]. Their type constraints are largely an implementation of the Retyped [36] tool.

For the recovery of object structures and relationships, **Objdigger** [40] 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 [56] to track individual ThisPtr propagation and usage between and within functions.

8.3 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 [39] 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.

8.4 Constraint 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. These constraints are then solved using algorithms to find the best fit that satisfies all constraints.

Lee et al. presented a principled type inference tool that can work with both static and dynamic analysis called **TIE** [17]. It is built on BAP [25]. Initially, the binary code is lifted to Binary Intermediate Language (BIL), the intermediate representation of BAP. BIL provides the initial low-level type information on which TIE builds on with further analysis. For example, if a variable is loaded, BIL will say if it is loaded from memory or register; and in the later case, how many bits the register has. TIE recovers the high-level variables by using VSA and analyzing access patterns in memory. TIE generates constraints on the type variables based upon how the variable is used, e.g., if a variable is used as part of signed division, they add the constraint that it must be a signed type. Then TIE tries to solve the constraints to get the types of each variable. The ideology of TIE is they want to infer as much as possible given the hints but not guess anything. If they can come to the conclusion that a variable is an integer but they are not sure if it is signed or not, they will infer it as a custom type number, which can be either signed or unsigned int. So, their output can be considered as a range of possible types a variable can have. It would be the responsibility of the reverse engineer to find the actual type given the range of possibilities.

Retyped [36] developed by Noonan et al. applied principled static type-inference algorithms for type inference including structure types and polymorphic types [57]. It operates on an intermediate representation generated by GrammaTech's static analysis tool CodeSurfer [58]. Retyped 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, Retyped generates type constraints from a TSL-based abstract interpreter [59]. They also include the pre-computed type information obtained from externally linked functions in this stage. Retyped then uses a constraint solver algorithm called the Simplification Algorithm based on [60] and later processes the output to infer C-like type information.

BinSub [42] is another principled type inference tool that reformulated Retyped with algebraic subtyping [61]. Retyped has a subtyping system that allows its type

inference algorithm to be polymorphic and flexible. However, their tool has a worst-case execution time of $O(n^3)$. Binsub reformulated the subtyping scheme of Retyped with algebraic subtyping which they show is faster. Initially, BinSub lifts the disassembly to IR and generates type constraints. The type constraints are then converted to subtyping constraints. Then the constraints are solved to infer types. The major change it has with Retyped is in the constraint-solving scheme, which uses algebraic subtyping. BinSub is implemented with Angr framework and for comparing their accuracy with Retyped they evaluated their tool and Retyped with Average Type Distance.

8.5 Statistical Approach

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 working with the source of a whole function, it devised a method to capture the usage signature of the variable under inspection by adopting the concept of n-gram [62]. An n-gram is a sequence of 'n' items (like words or letters) taken from a larger text. For example, in decompiled source code a sequence of 5 tokens will be called a 5-gram. As the usage signature of a variable, Stride captures the n-grams before and after a variable. Before doing that, it normalizes the decompiled source for more generalization. It created a large database of these n-grams and their corresponding types. The concept is if the usage signature of a variable from an unseen source matches the usage signature of a variable stored in the database, the type of the unseen variable should be the same/similar to the saved one in the database. It generates a large collection of such usage signatures and matches unseen variable's usage signatures with the database. Depending on the count of matches and the length of the n-gram, it tries to predict the type.

8.6 Type Propagation (Type Sources and Sinks)

NTFUZZ [37] presented a static analysis method to infer system call signatures on the Windows operating system. NTFUZZ is a type-aware kernel fuzzing [63] 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 whole program in a bottom-up style to infer the types of the system calls.

8.7 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, 64] 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.

8.8 Type Inference in Different RE Tools

Ghidra is one of the most popular reverse-engineering tools providing features such as disassembly, control flow graph generation, decompilation, recovering type information, etc. Firstly, it performs Value Set Analysis for variable recovery [65] on its Intermediate Representation called Pcode. Ghidra employs multiple methods for type information recovery. For library function calls, Ghidra often knows the prototype of the function. Ghidra stores common library function signatures and uses them to match calls to determine their prototypes [4]. Then the recovered information can be propagated to identify types of more variables. Ghidra also employs a rule-based approach, where how memory is accessed in the instructions helps predict the data type being accessed [4, 29]. This method tries to identify common code patterns for using different data structures and predicts the data type based on that. For example, the pattern for accessing a globar integer array is different than a heap-allocated integer array.

IDA Pro is another popular proprietary reverse engineering tool that is closed source. Initially, within each identified function, IDA conducts a thorough analysis of the stack pointer register's behavior to identify accesses to variables stored in the stack and to comprehend the function's stack frame layoutc̃iteidabook. IDA has a set of pattern-matching rules to identify the data structure fields. For instance, they assume that field accesses occur by first loading the base address of the data structure into a

register, followed by adding the field offset to the register [29]. For type recovery, IDA uses various static analysis methods. Similar to Ghidra, IDA also stores and matches signatures of library functions to detect them in the disassembly [66]. Then the type information recovered from those library functions is propagated to annotate more variablesc̃iteidabook.

RetDec [2, 67], an open-sourced reverse engineering toolchain developed by Avast is a retargetable machine-code decompiler based on LLVM [68]. 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 [69]. 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 [70]. 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 [71]. 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)** [25] is another popular suite of utilities and libraries that enables binary program analysis. However, BAP does not support type inference [72]. **Ddisasm** uses the type recovery algorithm of retyped [73].

9 Machine Learning Tools

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) [74], 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. Because, 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.

9.1 Classifier Models

Classification models are the simplest kind of machine learning models. These models divide data points into predefined groups called classes. Classifiers learn class characteristics or features from input data and learn to assign possible classes to unseen data based on their features.

TypeMiner is one of the first tools that used machine learning techniques for type prediction in binaries. They used a combination of Random Forest classifiers [75] and Linear Support Vector Machine [76] 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's [33, 77]program analysis method starts with Variable discovery and related instruction set extraction for discovered variables. They used a technique similar to Value-set analysis [47] for variable discovery identifying variables from functions analyzing global data and heap memory usage. Afterward, Bity analyzes the dependency relation between instructions and tries to find instructions related to a discovered variable. This set of instructions is supposed to represent how a variable is stored, interpreted, and manipulated which should provide important information for type recovery. They perform more analysis on the selected instruction to generate a feature vector. For example, they use Term Frequency-Inverse Document Frequency (TF-IDF) [78] to choose which instructions to keep and which to discard according to indicative quality for type prediction. For the type prediction model, they trained a classifier trained by Support Vector Machine(SVM) with a linear kernel and Random Forest.

DEBIN [5] by He et al. augments stripped binaries by predicting DWARF information. DWARF data contains high-level variable and type information useful for decompilation and program understanding. Initially, given the binary code, DEBIN lifts it to BAP-IR [25]. Then they break down the BAP-IR into smaller units which they call program elements, such as Rigesiter Variables, Memory Offset Variables, Flags, Constants, Locations, etc. Some of the elements might look out of place, but it is because they are parts of BAP-IR, not raw assembly. Then DEBIN analyses the lifted IR and classifies the program elements into items related to a source variable or not, using an Extremely randomized Tree[79] classification model. The reason for this step is to separate elements that were used to represent a variable from those that were used for other purposes, such as temporary usage for optimization. Afterward, 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) [80], 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]used statistical and machine learning methods to predict the return types of functions. A function may have one or multiple return statements. When they compiled the C programs for their dataset generation, they modified the source functions by adding non-operational code and labels to all of the return statements. These instrumentations helped them to break down the function binary into multiple chunks where each chunk is associated with each function invocation and return statements. However, using all the instructions of a chunk as model input might not always work for the maximum input length limitation. They experimented to choose the best set of instructions (features) as input to predict the return type with feature selection methods. For the final classification task, they experimented with 14 different classifiers such as AdaBoost, Bernoulli naive Bayes, Decision tree, Perceptron, Logistic regression, Linear support vector machine, and Support vector machine.

9.2 CNN Tools

CNNs [74] offer an optimal architecture for identifying and learning essential features in images, audio, signal, and sequential data. The instructions are represented as a sequence and feed to the CNN model. Though CNNs are not well-known for working with sequential text data, they can do a decent job of text classification, especially when combined with pre-trained embeddings like Word2Vec.

Cati [32] developed by Chen et al. is a type prediction tool that relies on CNN machine learning technologies. It uses different preprocessing methods on the input instructions. For example, it replaces all the addresses, function names, and numerical values with their corresponding special tokens. For selecting candidate instructions for type prediction, it only considered the memory access instructions and dereference instructions. It considered the backward 10 and forward 10 instructions to the target instruction as the context. They performed statistical analysis to show that spatially local instructions are probabilistically more correlated. The sequence of instructions was then encoded to a vector representation using WordToVec [81] to feed to a

multi-stage classifier model. The model is a set of multiple CNN, whereas most type-prediction tools use Natural Language Processing (NLP) models. The multi-stage classifier makes predictions on multiple stages. For example, in the first step, it predicts if the variable is a pointer or not. If it is a pointer the next step is to predict what kind of pointer it is (struct, void, or other basic type), and if it is not a pointer then what kind of basic type it was.

9.3 Natural Language Processing(NLP) Models

NLP models are a natural fit for type prediction tasks. NLP models are tailored to excel at sequence processing tasks. Executable programs are represented as sequences of assembly instructions. Though assembly is not a natural language, they have syntax and semantic meaning like other high-level languages. Models such as transformers or Large Language Models (LLMs) consider the context of a word within a sentence. In case of instructions, context is critical for understanding how instructions interact and contribute to the overall program behavior. NLP models can learn these relationships effectively. Additionally, models like transformers and RNNs are capable of generating embeddings from instructions that capture semantic relationships. Recent advances in LLMs like ChatGPT and LLAMA have shown significant promise in binary analysis tasks. These models leverage their strong contextual understanding and transfer learning capabilities to infer the semantics of binary code effectively. Though we have not seen these latest LLMs used for type prediction tasks yet, that remains a lucrative scope for future research.

EKLAVYA is one of the first tools to use a deep learning model tailored for natural language processing (NLP) for recovering type information. They used a Recurrent Neural Network [82] to predict function signature. As the model input, they decided to use the raw bytes of the functions rather than the text representation. However, they used Skip-gram negative sampling [83] 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 would be to find the most related instructions for a given instruction. This method is supposed to generate 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 by Chen et al. is a Transformer-based model that performs both type prediction and variable name prediction. The reason for them to perform both tasks together is they think these tasks are supplementary to each other. For example, programmers usually name integer variables differently than chars. Dirty takes decompiled source code from IDA Pro as input instead of machine instructions. IDA Pro generates some interim analysis results on the binary. DIRTY also includes some of those analysis results about the data-layout of the program with the input as well, such as size, storage location(e.g., register or stack offset), size, nested data types (e.g., if the variable is a struct), and offsets of its members, if any (e.g., offsets in an array or of fields in a struct). DIRTY uses this information to learn a soft mask which influences the outcome of the Trasnformer model. For example, if the initial prediction by the decoder turns out to be incompatible with the data-layout information, the mask is

supposed to reduce the probability of the prediction. DIRTY depends on the analysis of IDA Pro. They claim any decompilers can be used instead of IDA Pro providing that tool provides both decompilation and data-layout information.

SnowWhite by Lehmann et al. [35, 84] is another NLP based approach that tries 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 [85]. Code written in languages such as C/CPP, C#, GO, or Rust can be compiled into WebAssemblies. Another advantage of WebAssemblies is 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. So, instead of tailoring for multiple languages, they tailored their model for the common format. SnowWhite uses bidirectional LSTM [86] with global attention as the machine learning model to predict the types. They use two models to predict return type and parameter types of functions.

StateFormer uses Roberta [87], a trasnformer model for predicting the types from disassembled instructions. They used a Transformer[88] Model which is tailored for Natural Language Processing. So they pre-trained their transformer model with a related task which is supposed to make the model learn the assembly language before starting on the final task of type prediction. They pretrain StateFormer with Generative State Modeling(GSM). In short, GSM means trying to simulate the behavior of instructions and their effect on register values. For example, GSM should teach the model that `add ecx , 3 ;` should increase the value of `ecx` by 3 or `add ecx ;` should increment `ecx` value by 1. After pretraining they transfer StateFormer's learned knowledge by finetuning it for type inference.

SeeType is another transformer-based tool for type prediction. SeeType initially pre-trains a BERT [89] model on masked language modeling and data-dependency prediction tasks. The pretraining is supposed to familiarize the BERT model with assembly language which was originally pre-trained on natural languages. To formulate the input for the model for type prediction task, it tries to capture the context of a target instruction. Sometimes a function can be large and the whole function might not fit into the model as input. So, to solve that problem, they used program slicing [] methods. It utilized both forward and backward slicing to formulate the input rich in context for the target instruction. For training SeeType they also generated a large dataset of open-source programs. They also provided a method for generating ground truth by using both debug-information and the source code. It also reflects on the importance of tokenization and the importance of numerical values in machine code and proposes a method to handle numerical values in a better way.

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 [90]. 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[91, 92]. Different tools have tried various methods for addressing this issue, as numerical values are a significant part of machine code. 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 [31] used a different method to handle numeric values. They included the Neural Arithmetic Unit (NAU) [93], which creates embeddings meant to represent the meaning of numbers used in arithmetic operations. SeeType also addressed this issue. They tried to train their tokenizer in a way to set the vocabulary optimized for handling numeric values.

9.4 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks utilizes a deep learning architecture and specializes in processing and analyzing data represented as graphs. Unlike other traditional neural networks that operate on images, audio, or text, GCNs directly take graph-structured data as their input. Graph structured data consists of nodes (entities) and edges (relationships). Binary programs are often analyzed and represented as graph representations, such as control-flow graphs or data-flow graphs. GCNs offer an opportunity to automatically analyze those popular graph representations and infer type information from them.

TIARA [38] recovers STL container classes [94] in C++ binaries using both principled and machine learning methods. They use the idea of binary program slicing [95–97] to obtain a type-relevant inter-procedural forward slice of instructions to summarize how the variable is used in the binary. The forward slice includes all the instructions that are affected by the target instruction. The backward slice contains all the instructions that affect the target instructions in any way. However, they did not use backward slicing to create their variable context like other related works [32, 98]. TIARA generates a Control Flow Graph with the selected instructions from their program slicing algorithm TSlice. Then they train a GCN (Graph Convolutional Network) [99, 100] to predict the container type of a given variable.

TYGR[41] is another Graph Neural Network [101] based tool for inferring both basic and struct variable types in stripped binary programs. TYGR is architecture agnostic as it uses VEX IR [102] which is an architecture-agnostic IR for many different architectures. Initially, they generate the control flow graph of a function. Then they perform program execution along different non-cyclic paths in the CFG to generate data-flow graphs for each variable along each possible path. Then they aggregate all the data flow graphs to get the function level data-flow graph. Then they encode the aggregated dataflow graph into an embedding and train a classifier model to predict types. They also created a new dataset TYDA on which they evaluated their model and compared it with baseline methods.

.

10 Tool Input

In this section, we are going to discuss the initial representation of the programs the tools used for their method to predict the types. In general, the input of all the type inference is initially the binary file itself. Because everything the tools use for this task are produced from the binary. For example, the disassembly, data-flow graph, IR, etc all comes from the binary.

ML Tools Input. A variety of input is used by the tools depending on which input representation suits their type inference method. For example, most of the recent tools that use some kind of NLP model use some form of sequential representation of the binary. Sequential representation includes disassembled instructions (StateFormer, SeeType, CATI, SnowWhite, Escalada et al.), decompiled source code (DIRTY, STRIDE) or Intermediate Representation (DEBIN). These are sequential representations because the order in which they are present in the program has valuable meaning. For example, if you change the order of the instructions in a basic block, the function of that basic block might change. This property is the same in source code. NLP models are known for their ability to understand the dependencies in a sequence. This is why Transformer, RNN, or LSTM models mostly rely on the sequential representation of the input. One exception in this scheme is EKLAVYA, which uses raw function bytes as its input even though it uses an RNN model for type inference. Even though they are bytes, they actually represent the instructions but are not disassembled. TYGR takes a function's data-flow graph as its input and it uses a Graph Neural Network as it's model. TypeMiner uses a random forest classifier model and uses an unordered representation of the instructions as their input.

Use of Intermediate Representation Tools that are general reverse engineering tools usually use a variety of techniques. They perform a multitude of different kinds of analysis which often contributes to their type prediction. However, one common trait they all share is they operate on IR rather than instructions. Using IR lets them support a variety of architectures. With IR, they do not have to handle each architecture, but just transform each architecture code to IR. If the analysis is done on the IR, the same analysis can be done on binaries from multiple architectures. This way, focus can be given on improving the analysis leaving the platform dependent constraints to IR. If IR is not used, the same analysis would have to be modified for applying on binaries from different architecture. Besides, IRs are often of higher abstraction than machine code, which can make type prediction task easier. Some popular RE tools and the IR they use are Ghidra's P-code, IDA Pro's Microcode, Angr's Vex, BAP's BIL. Binary Ninja employs a series of IRs, Low-Level IL (LLIL) which is closest to assembly, Mid-Level IL (MLIL) and High-level IL(HILL). Besides, DEBIN and TIE also uses BAP's BIL. Retyped uses the IR from CodeSurfer[58]. There are also some limitations of IR as well. The process of producing IR is not always perfect and can introduce error in the IR. Besides, Some architecture-specific details may be abstracted away in IR, leading to challenges in reconstructing the original behavior accurately.

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 (Retyped, BinSub, Osprey) to inferred types from another tool as in DSIbin.

Dynamic Tools. The dynamic analysis tools main input is the executable itself as that needs to be run. Other than that some methods look at the memory image of the program by producing memory graphs, such as DSIBin[22], Artiste[15] ,MemPick [14], and DDT[19]. Besides, Rewards[103] performs type propagation so it needs to know the signatures of library functions as well.

11 Use of Context

In programming, "program context" refers to the surrounding information, data, and environment that influences how a program runs and interprets its instructions. Often some binary functions can be huge which demands high computational resources. This can cause issues, especially with machine learning type inference tools. NLP models often struggle with handling long inputs and maintaining context over extended sequences due to their architecture and computational limitations. Traditional RNNs struggle with long-range dependencies due to the vanishing gradient problem, while transformers handle context better but face efficiency limits from quadratic complexity in sequence length [88, 104]. Context is critical for binary-type inference tasks as it provides the surrounding instructions or program behavior necessary to infer the role and relationships of variables and functions. The naive method to include the context of an instruction is to consider the containing function as the context. Long functions can cause problems in this regard. Different NLP-based tools handle this limitation in different ways:

Program Window Cati and SnowWhite use a very similar method for extracting the context of the target instruction that corresponds to variables. They take 10 instructions backward and 10 instructions forward. So, the context is a total of 21 instructions including the target instruction.

Stride based their core type prediction method based on context similarity. They say if the surrounding code near the target variable is similar in two cases, then they are likely of the same type. To match the surroundings they matched n-grams of different lengths.

program slicing Binary program slicing is another technique that is used to extract the context of an instruction. Program slicing techniques extract relevant portions of a program's code by analyzing its data dependencies and ignoring unrelated parts [95]. It can be performed forward (tracing how a value influences subsequent instructions) or backward (tracing how a value is computed or used). TIARA uses forward slicing and SeeType uses both forward and backward slicing to generate their context.

12 Languages Supported

Most type recovery tools we surveyed primarily focus on executables compiled from C/C++ programs. Among them, tools that only focus on C binaries are Binsub, Escalada et. al., Bity, Cati, Stateformer, Dirty, Debin, SeeType, Eklavya, Osprey, TIE, and Howard. Tools targeting both C and C++ binaries are DSIBIN, Stride, Retyped,

and TypeSquezzer. Whereas Tiara, Objdigger, and OOanalyzr focus on C++ binaries. Snowwhite works on WebAssemblies which can be produced from C, C++, Rust, or Go.

13 Type Coverage

The type tools cover different types of type analysis tasks. 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 Types: Primitive or atomic types are the most basic data types, such as integers (signed, unsigned, short, long, etc), floats, characters, booleans, pointers, and arrays. Predicting primitive types in binary analysis is advantageous for understanding the fundamental building blocks of a program's operations. By accurately identifying primitive types from binary code, analysts can infer the structure of data and better understand how a program behaves. Moreover, predicting primitive types is crucial for reconstructing higher-level abstractions because they serve as the fundamental building blocks for more complex data structures. Tools like Binsub, TYGR, Escalada et. al., Bity, Cati, Stateformer, DEBIN, Eklavya, TypeMiner, and SeeType primarily focus on producing primitive types.

Structure Identification: Some tools try to identify structures and classes used in the executables. Tiara tries to identify popular C++ classes like STL container classes. Similarly, OOanalyzer tries to identify C++ classes and methods in those classes. DSIBIN tries to identify frequently used data structures like Soubly Linked Lists, Singly Linked Listks, Skip Lists, Nested Structures, etc. Mempick Can identify linked lists, binary-tree, n-array, trees, etc. DDT and DSIBIN also perform data structure identification.

Structure/Class Reconstruction: One difference between structure reconstructing tools from structure identifying tools is that identification means confirming a structure under inspection is from a pre-defined set of structures. Structure reconstruction focuses on producing the members of the structure that constitute the structure itself. So, even if the structure was not known before, this kind of analysis should be able to produce some insights about the novel structure. Retyped, Osprey, Rewards, ARTISTE, TIE, OOAnalyzer, Objdigger, and Howard fall in this category.

Others: Some tools produce other kinds of type information. Lego can analyze and produce class hierarchy performing dynamic analysis. Objdigger can also find class hierarchy, using static analysis. DynCompB predicts if two variables have the same high-level purpose. Laika tries to find similar Objects. Additionally, Binsub and Typesquezzer also focus on recovering function signatures.

14 Pre-processing

14.1 Handling Numerical Values

Addresses and other numerical values are useful because they carry information about control flow and the storage locations of source-level variables [90]. 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[91, 92]. Different tools have tried various methods for addressing this issue, as numerical values are a significant part of machine code. 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 [31] used a different method to handle numeric values. They included the Neural Arithmetic Unit (NAU) [93], which creates embeddings meant to represent the meaning of numbers used in arithmetic operations. SeeType also addressed this issue. They tried to train their tokenizer in a way to set the vocabulary optimized for handling numeric values.

15 Evaluation

Different type tools produce different kinds of type information which requires different kinds of evaluation. For example, evaluation metrics for primitive type prediction are not suitable for evaluating tools that produce definitions for structures or classes.

15.1 Perfect Match Accuracy:

This kind of metric assesses how precisely a tool's predictions align exactly with the actual values or labels. It typically measures cases where an output must match the ground truth completely to be considered correct. This type of accuracy is stringent, emphasizing the importance of exactness in model performance. The majority of the type prediction tools use this kind of evaluation, such as accuracy, precision, recall, and f-1/. Tygr, SeeType, StateFormer, Tiara, Rewards, Cati, Dirty, Debin, EKLAVYA, TypeMiner, and OSPREY evaluated their method using a variant of this metric.

15.2 Fine-graned Accuracy:

Fine-grained accuracy involves measuring the relative closeness or similarity between predicted values and ground truth. This method allows for a degree of error while measuring correctness. For example, if a tool tags a short integer as an integer, perfect match accuracy would say it is totally wrong. A fine-grained method would consider the fact that it has identified the variable as an integer but failed to tag it as short and score it as partially correct. These kinds of metrics are more customized for the problem and might provide some interesting insights.

Type Distance Metric: The Type Distance Metric evaluates the accuracy of type inference systems by measuring the average distance between inferred and ground-truth types using a type lattice. So, the closer the predicted type is to the actual type, the more correct this is. TIE, Binsub, Retyped, Bity, and ARTISTE use some version of type lattice to evaluate their performance. In the type lattice of TIE in Figure 1, we can see that how types are presented in a type lattice. The intuition is, that if the tool can not generate the exact type, doing something close has some value.

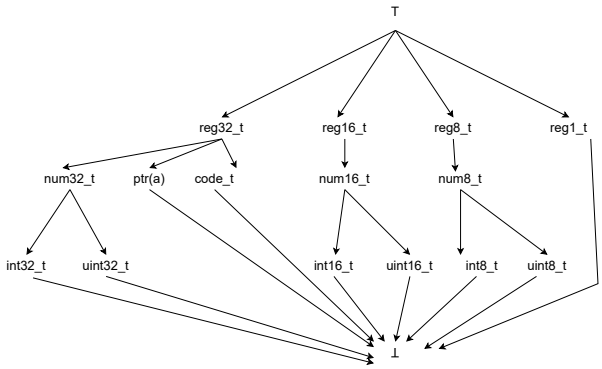

Fig. 1

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.

Edit distance: This is a measure used to evaluate the quality of C++ class abstractions. This metric calculates the number of modifications needed to transform the set of classes recovered by an analysis tool into the correct set of classes. The modifications include moving methods between classes, adding missing methods, removing extraneous methods, and merging or splitting classes. This metric provides a way to assess how closely the recovered class structures resemble the ground truth. OOAnalyzer uses this metric to measure their performance in reconstruction C++ classes.

16 Discussion

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

Dynamic Typing and Modern Programming Constructs: Languages with dynamic typing such as Python and JavaScript pose particular challenges for type

inference tools. The flexibility of these languages allows on-the-fly type changes and polymorphism, which can be very complex to analyze. To the best of our knowledge, none of the type inference tools are specialized in binaries generated from dynamically typed languages. As programming languages evolve, new constructs and paradigms (like generics in Java) are becoming popular. Future tools will need to evolve to understand and predict these to be useful.

Scalability: This limitation is true for almost all binary analysis tasks. With the increase in the size of the binary, the computational resources required for analyzing them increase exponentially. Some of the tools try to perform an analysis of a piece of the program rather than the whole program to mitigate this issue. This area needs more attention as it hinders almost all binary analysis tasks, including type inference.

User Experience and Tool Integration: Except for the general purpose reverse engineering software like IDA Pro, Ghidra, and Angr; most tools do not have a user-friendly interface developed for using their tool. Many of the time, the tools are not open-sourced. In the case of the open-sourced ones, there is a lack of documentation. Releasing the tools as a plugin for popular tools like IDA or ghidra can be a solution to this.

Lack of Standardization in Output Formats Complicates Tool Comparison: A problem in evaluating and comparing type inference tools is the lack of a standardized output format across different systems. Each tool tends to produce outputs that reflect its unique approach to type inference, which can vary widely in terms of detail, format, and the kinds of type information provided. This diversity in outputs makes it challenging to directly compare the effectiveness, accuracy, and comprehensiveness of different tools.

17 Conclusion

References

[1] Caballero, J., Lin, Z.: Type inference on executables. ACM Comput. Surv. **48**(4) (2016) https://doi.org/10.1145/2896499

[2] Retdec, A.: Retdec Github Repository. GitHub

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

[4] Eagle, C., Nance, K.: The Ghidra Book: The Definitive Guide. No Starch Press, ??? (2020)

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

[6] Green, H., Schwartz, E.J., Goues, C.L., Vasilescu, B.: Stride: Simple type

recognition in decompiled executables. arXiv preprint arXiv:2407.02733 (2024)

[7] Chen, Q., Lacomis, J., Schwartz, E.J., Goues, C.L., Neubig, G., Vasilescu, B.: Augmenting Decompiler Output with Learned Variable Names and Types (2021)

[8] Escalada, J., Scully, T., Ortin, F.: Improving type information inferred by decompilers with supervised machine learning. ArXiv **abs/2101.08116** (2021)

[9] Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: OSDI, vol. 8, pp. 255–266 (2008)

[10] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1) (2009) https://doi.org/10.1145/1609956.1609960

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

[12] Eager, M.J.: Introduction to the dwarf debugging format (2012)

[13] Srinivasan, V., Reps, T.: Recovery of class hierarchies and composition relationships from machine code. In: International Conference on Compiler Construction, pp. 61–84 (2014). Springer

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

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

[16] Slowinska, A., Stancescu, T., Bos, H.: Howard: A dynamic excavator for reverse engineering data structures. In: Network and Distributed System Security Symposium (2011). https://api.semanticscholar.org/CorpusID:43281

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

[18] Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 11th Annual Information Security Symposium, pp. 1–1 (2010)

[19] Jung, C., Clark, N.: 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, pp. 56–66 (2009)

[20] Dolgova, K., Chernov, A.: Automatic type reconstruction in disassembled c programs. In: 2008 15th Working Conference on Reverse Engineering, pp. 202–206 (2008). IEEE

[21] Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 255–265 (2006)

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

[23] White, D.H., Rupprecht, T., Lüttgen, G.: 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, pp. 259–269 (2016)

[24] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of computer programming **69**(1-3), 35–45 (2007)

[25] Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, pp. 463–469 (2011). Springer

[26] 35, V.: Binary Ninja. GitHub

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

[28] Maier, A., Gascon, H., Wressnegger, C., Rieck, K.: Typeminer: Recovering types in binary programs using machine learning. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 288–308. Springer, Cham (2019)

[29] Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.-c., Kwon, Y., Aafer, Y., Zhang, X.: Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 813–832 (2021). https://doi.org/10.1109/SP40001.2021.00051

[30] Chua, Z.L., Shen, S., Saxena, P., Liang, Z.: Neural nets can learn function type signatures from binaries. In: Proceedings of the 26th USENIX Conference on Security Symposium. SEC'17, pp. 99–116. USENIX Association, USA (2017)

[31] Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., Jana, S.: 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. ESEC/FSE 2021, pp. 690–702. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3468607 . https://doi.org/10.1145/3468264.3468607

[32] Chen, L., He, Z., Mao, B.: Cati: Context-assisted type inference from stripped binaries. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 88–98 (2020). https://doi.org/10.1109/DSN48063.2020.00028

[33] Xu, Z., Wen, C., Qin, S.: Type learning for binaries and its applications. IEEE Transactions on Reliability **68**(3), 893–912 (2019) https://doi.org/10.1109/TR.2018.2884143

[34] Lin, Z., Li, J., Li, B., Ma, H., Gao, D., Ma, J.: 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. CCS '23, pp. 2725–2739. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623214 . https://doi.org/10.1145/3576915.3623214

[35] Lehmann, D., Pradel, M.: Finding the dwarf: Recovering precise types from webassembly binaries. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 410–425. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523449 . https://doi.org/10.1145/3519939.3523449

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

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

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

[39] Schwartz, E.J., Cohen, C.F., Duggan, M., Gennari, J., Havrilla, J.S., Hines, C.: 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, pp. 426–441 (2018)

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

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

[42] Smith, I.: Binsub: The simple essence of polymorphic type inference for machine code. arXiv preprint arXiv:2409.01841 (2024)

[43] Rival, X., Yi, K.: Introduction to Static Analysis: an Abstract Interpretation Perspective. Mit Press, ??? (2020)

[44] Cousot, P., Cousot, R.: 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, pp. 238–252 (1977)

[45] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of logic and computation **2**(4), 511–547 (1992)

[46] Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) Compiler Construction, pp. 5–23. Springer, Berlin, Heidelberg (2004)

[47] Balakrishnan, G., Reps, T.: Wysinwyx: What you see is not what you execute. ACM Transactions on Programming Languages and Systems (TOPLAS) **32**(6), 1–84 (2010)

[48] De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008). Springer

[49] Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy, pp. 380–394 (2012). IEEE

[50] Ramos, D.A., Engler, D.: {Under-Constrained} symbolic execution: Correctness

checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 49–64 (2015)

[51] Schwartz, E.J., Avgerinos, T., Brumley, D.: 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, pp. 317–331 (2010). IEEE

[52] Angr: Angr Github Repository. GitHub

[53] Angr: Angr Github Repository. GitHub

[54] Van Emmerik, M.J.: Static Single Assignment for Decompilation. University of Queensland, ??? (2007)

[55] Angr: Angr Github Repository. GitHub

[56] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(1), 26–60 (1990)

[57] Rehof, J., Fähndrich, M.: Type-base flow analysis: from polymorphic subtyping to cfl-reachability. ACM SIGPLAN Notices **36**(3), 54–66 (2001)

[58] Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: Codesurfer/x86—a platform for analyzing x86 executables. In: International Conference on Compiler Construction, pp. 250–254 (2005). Springer

[59] Lim, J., Reps, T.: Tsl: A system for generating abstract interpreters and its application to machine-code analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **35**(1), 1–59 (2013)

[60] Fahndrich, M., Aiken, A.: Making Set-constraint Program Analyses Scale. University of California at Berkeley, ??? (1996)

[61] Dolan, S., Mycroft, A.: Polymorphism, subtyping, and type inference in mlsub. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 60–72 (2017)

[62] Brown, P.F., Della Pietra, V.J., Desouza, P.V., Lai, J.C., Mercer, R.L.: Class-based n-gram models of natural language. Computational linguistics **18**(4), 467–480 (1992)

[63] Jeong, D.R., Kim, K., Shivakumar, B., Lee, B., Shin, I.: Razzer: Finding kernel race bugs through fuzzing. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 754–768 (2019). IEEE

[64] Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer,

M.: Control-flow integrity: Precision, security, and performance. ACM Comput. Surv. **50**(1) (2017) https://doi.org/10.1145/3054924

[65] Ghidra: Ghidra Github Repository. GitHub

[66] Rays, H.: Hex Rays Documentation. Hex Rays

[67] Křoustek, J., Matula, P., Zemek, P.: Retdec: An open-source machine-code decompiler. In: July 2018 (2017)

[68] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86 (2004). IEEE

[69] Liu, Z., Yuan, Y., Wang, S., Bao, Y.: Sok: Demystifying binary lifters through the lens of downstream applications. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 1100–1119 (2022). IEEE

[70] Team, R.: The official radare 2 book, https://book.rada.re/ (2017)

[71] Potchik, B.: Architecture Agnostic Function Detection in Binaries. Binary Ninja Blog

[72] BAP, C.: BAP Github Repository. GitHub

[73] GrammaTech: ddisasm github repository. Github

[74] O'Shea, K., Nash, R.: An introduction to convolutional neural networks. ArXiv **abs/1511.08458** (2015)

[75] Parmar, A., Katariya, R., Patel, V.: A review on random forest: An ensemble classifier. In: International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICI) 2018, pp. 758–763 (2019). Springer

[76] Zhang, Y.: 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, pp. 179–186 (2012). Springer

[77] Xu, Z., Wen, C., Qin, S.: 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, pp. 430–446 (2017). Springer

[78] Salton, G.: Introduction to modern information retrieval. McGraw-Hill (1983)

[79] Geurts, P., Ernst, D., Wehenkel, L.: Extremely randomized trees. Machine learning **63**, 3–42 (2006)

[80] Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the Eighteenth International Conference on Machine Learning. ICML '01, pp. 282–289. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

[81] Mikolov, T., Chen, K., Corrado, G.S., Dean, J.: Efficient estimation of word representations in vector space. In: International Conference on Learning Representations (2013). https://api.semanticscholar.org/CorpusID:5959482

[82] Medsker, L.R., Jain, L., *et al.*: Recurrent neural networks. Design and Applications **5**(64-67), 2 (2001)

[83] Mikolov, T., Chen, K., Corrado, G.S., Dean, J.: Efficient estimation of word representations in vector space. In: International Conference on Learning Representations (2013). https://api.semanticscholar.org/CorpusID:5959482

[84] Luong, T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. In: Màrquez, L., Callison-Burch, C., Su, J. (eds.) Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1412–1421. Association for Computational Linguistics, Lisbon, Portugal (2015). https://doi.org/10.18653/v1/D15-1166 . https://aclanthology.org/D15-1166

[85] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. SIGPLAN Not. **52**(6), 185–200 (2017) https://doi.org/10.1145/3140587.3062363

[86] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. CoRR **abs/1409.0473** (2014)

[87] Zhuang, L., Wayne, L., Ya, S., Jun, Z.: A robustly optimized BERT pre-training approach with post-training. In: Li, S., Sun, M., Liu, Y., Wu, H., Liu, K., Che, W., He, S., Rao, G. (eds.) Proceedings of the 20th Chinese National Conference on Computational Linguistics, pp. 1218–1227. Chinese Information Processing Society of China, Huhhot, China (2021). https://aclanthology.org/2021.ccl-1.108

[88] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17, pp. 6000–6010. Curran Associates Inc., Red Hook, NY, USA (2017)

[89] Kenton, J.D.M.-W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of naacL-HLT, vol. 1, p. 2 (2019). Minneapolis, Minnesota

[90] Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: International Conference on Compiler Construction, pp. 5–23 (2004). Springer

[91] Thawani, A., Pujara, J., Szekely, P.A., Ilievski, F.: Representing numbers in NLP: a survey and a vision. CoRR **abs/2103.13136** (2021) 2103.13136

[92] Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., Janes, A.: Big code!= big vocabulary: Open-vocabulary models for source code. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1073–1085 (2020)

[93] Madsen, A., Johansen, A.R.: Neural arithmetic units. arXiv preprint arXiv:2001.05016 (2020)

[94] Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., ??? (1998)

[95] Weiser, M.: Program slicing. IEEE Transactions on software engineering (4), 352–357 (1984)

[96] Tian, J., Xing, W., Li, Z.: Bvdetector: A program slice-based binary code vulnerability intelligent detection system. Information and Software Technology **123**, 106289 (2020)

[97] Xue, H., Venkataramani, G., Lan, T.: Clone-slicer: Detecting domain specific binary code clones through program slicing. In: Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation, pp. 27–33 (2018)

[98] nahid: Seetype. who?, ??? (420)

[99] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

[100] Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)

[101] Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE transactions on neural networks **20**(1), 61–80 (2008)

[102] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices **42**(6), 89–100 (2007)

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

(2010)

[104] Hochreiter, S.: The vanishing gradient problem during learning recurrent neural nets and problem solutions. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems **6**(02), 107–116 (1998)