Bityr: Type Inference on Binaries by Modeling Data Flows as Graph Neural Networks

ABSTRACT

We propose Bityr, a framework for inferring fine-grained type information from binaries. Bityr employs a machine learning based approach, and is meant to be usable by security analysts. To support different architectures and compiler optimizations in a general and precise manner, Bityr leverages an architecture-agnostic data-flow analysis to extract a graph-based intra-procedural representation of data-flow information. To scale to large binaries and cope with missing inter-procedural information, Bityr encodes the representation using a graph neural network model. We have implemented Bityr atop the ANGR framework which targets VEX IR. We evaluated Bityr on a suite of 13,326 binaries ranging over 33 open-source software projects compiled for x64, x86, ARM, and MIPS architectures with different optimization levels. Bityr achieves precision of 92.3%, 90.7%, 80.1%, and 78.3%, respectively, significantly out performing the state-of-the-art approaches.

1 INTRODUCTION

Binary type inference is a core research challenge in binary analysis. It concerns identifying the data types of registers and memory values in a stripped executable (or object file). However, automated and precise binary type inference is a hard problem [31, 67] due to the lack of high-level abstractions and the rich variety and sophistication of compiler optimizations, hardware architectures, and adversarial obfuscation methods [53].

A predominant use case of type information, and even more so for binaries, is program understanding. Many important security applications are interactive [55, 57]. For instance, a security analyst uses an interactive disassembler (e.g., IDA [3] or GHIDRA [2]) to decipher semantic information from binaries (e.g., types of certain variables). These tools may not infer all the required information since the general problem is intractable. Consequently, the analyst uses the tool's information and their expertise to assist (e.g., provide the type of a variable) or correct (e.g., change the type of a variable) the tool's output. The tool uses the newly provided information to refine and improve the results. This interactive process continues until the analyst is satisfied with the understanding gained from the semantic information.

Exiting binary type inference solutions can be broadly classified into three categories: Rule- and heuristic-based solutions (e.g., type inference in IDA and Ghidra), constraint-solving-based solutions (e.g., TIE [36], RETYPD [42], and OSPREY [69]), and machine-learning-based solutions (e.g., DEBIN [32], STATEFORMER [44], TYPEMINER [38], and DIRTY [12]). Our work is motivated by the following three challenges faced by the state of the art tools: (1) *Low accuracy in inferred types on stripped binaries.* Even the best solution only achieves an average F1 score of around 78% during evaluation. (2) *Limited architectural support.* Many solutions, especially heuristic-based and constraint-based ones, only support binaries on one or a limited number of architectures because it is difficult to generalize rules, heuristics, and constraint-solving methods to a wide range of architectures. (3) *Feedback-insensitive.* While machine-learning solutions generally offer higher accuracy, none of them can incorporate type information that analysts provide during manual reverse engineering (or hints from other tools), which significantly hinders their applicability in interactive settings.

In this paper, we present Bityr, a machine-learning-based binary type inference technique that aims at assisting security analysts in interactive reverse engineering sessions with highly-accurate inferred types for registers and memory locations. Bityr lifts binary code into VEX IR [41], a uniform intermediate representation that supports many architectures and abstracts away hardware-specific details, runs a light-weight data-flow analysis on each function to collect information about how each variable is accessed, generates novel, graph-based representation of data-flow information, and finally trains a model based on Graph Neural Networks (GNN) [48] for type inference.

The use of VEX IR is instrumental to the performance of Bityr. On the one hand, VEX IR allows Bityr to support a wide range of architectures and easy to extend to new architectures [1, 6, 33]. On the other hand, VEX IR is suitable for performing data-flow analysis, allowing Bityr to extract data-flow information that is highly relevant for type inference.

To strike a balance between scalability and accuracy, Bityr employs a novel graph-based intra-procedural representation of the data-flow information. Because the representation is constructed on a per-function basis, Bityr scales to large, real-world binaries. This representation is acceptable to GNNs, a deep neural network model that is well-suited for predicting rich properties of graph-structured data [60]. To the best of our knowledge, Bityr is the first to demonstrate the effective application of GNNs to the problem of binary type inference. Particularly, we demonstrate that they can adequately tolerate missing inter-procedural information, thereby allowing data-flow analysis to scale.

We implement Bityr using the ANGR binary analysis framework [53] and evaluate it on a suite of 13,326 binaries ranging over 33 open-source software projects compiled for four architectures (x64, x86, ARM, and MIPS) with four optimization levels, O0, O1, O2, and O3. On which Bityr achieves an overall type inference precision of 92.3%, 90.7%, 80.1%, and 78.3%, respectively.

Contributions. This paper makes the following contributions:

- We propose a novel graph-based representation of data-flow information that enables to synergistically combine a data-flow analysis and a graph neural network model to balance scalability and accuracy.
- We implement Bityr, a system that uses above GNN model trained on a large dataset of non-stripped binaries to recover types of high-level program variables from new, unseen stripped binaries.

- We demonstrate the effectiveness of Bityr by extensively evaluating it on a large corpus of 13,326 binaries and show that it achieves precision ranging 92.3% - 78.3%, significantly outperforming the state-of-the-art approaches.

In the spirit of open and reproducible science, we will open source Bityr and all evaluation artifacts upon the acceptance of this paper. For reviewing, we made an anonymized pre-release of the code base of Bityr at https://anonymous.4open.science/r/bityr-review/.

2 BACKGROUND

Before diving into the technical details of Bityr, we will first present necessary background knowledge for readers in this section, namely binary reverse engineering, binary type inference, and GNN.

2.1 Binary Reverse Engineering

Binary reverse engineering is the process of understanding a program without access or only having limited access to its source code. Security analysts reverse engineer binaries to understand the behaviors or provenance of malware [21, 64], discover vulnerabilities in binaries [39], and mitigate defects in legacy software [51]. In most cases, debug symbols are not available to security analysts, and they are forced to manually recover lost semantic information, such as variable locations, names, and types, during reverse engineering.

2.2 Type Inference on Binaries

Binary type inference is the automated process of reconstructing source-level type information, e.g., types of local variables and function arguments, from untyped byte-addressed memory and registers. It is challenging because most information is discarded during compiling unless debug symbols are preserved. As shown in Table 1, exiting binary type inference solutions can be broadly classified into three categories based on their core techniques: Rule- and heuristic-based type inference solutions, constraint-solving based solutions, and machine-learning-based solutions.

A key difference between these solutions is if they support type inference of structs and struct members (or struct layouts). Inferring struct members and their types requires complex and accurate reasoning and fine-grained flow information [10], which is hard to gain during static analysis. Most non-constraint-based inference techniques (i.e., top and bottom column groups) do not predict struct member types. Rewards [37] and Howard [54], which do predict struct member types, use dynamic traces to get precise offset information. However, as with any dynamic techniques, they suffer from low completeness: Their only support assembly code that is reachable during execution. Therefore, we make a design decision for Bityr to not use dynamic traces and not infer struct members. As we will show in Section 6.2, Bityr outperforms existing state-of-the-art binary type inference techniques.

2.3 Graph Neural Networks

Graph Neural Network (GNN) is a deep neural network architecture that is well-suited for predicting rich properties of graph-structured data [60], through a procedure called *message passing*. GNNs have been used for many program understanding and analysis tasks, such as identifying variable misuses in C# programs [5], localizing and repairing bugs in JavaScript code [20], predicting types in Python programs [4], and detecting code clones [61]. To the best of our knowledge, Bityr is the first to demonstrate the effective application of GNNs to the problem of binary type inference.

3 OVERVIEW

In this section, we set out by presenting the binary type inference problem. We then provide an overview of Bityr's architecture, highlighting key design choices that enable it to achieve the aforementioned criteria.

3.1 Binary Type Inference

We consider the problem of mapping binary-level variables to source-level types. Specifically, we focus on parameters and local variables in functions, which are crucial for understanding the behavior and intent of the function—and are thus of interest to reverse engineering. We illustrate our objective with an example in Figure 1, which shows a C function and its assembly code extracted from the x64 binary compiled with GCC using optimization level O0. In practice, only the binary is available, but inferring types for data that directly correspond to source-level variables is helpful for understanding the intent of the function, and perhaps even extracting a faithful decompilation. At the binary level, local variables are typically stored at stack offsets. For instance, the stack offset `-0x30(%rbp)` corresponds to `name_len` and `-0x38(%rbp)` corresponds to `ext_len`.

Our goal is to predict fine-grained type information in the form of C types such as `int32`, `uint64`, `struct*`, and `char**`, which are familiar to users with experience in popular tools for interactively reverse-engineering binaries such as IDA and Ghidra. In particular, we treat type inference as a classification problem, meaning that the variety of types we can predict is finite. While C types may be arbitrarily complicated, we found that our finite subset is still sufficiently expressive to cover nearly all cases that arise in practice. We describe our type system in more detail in Section 4.3 and document the frequency of types within our dataset in Section 5.

3.2 Architecture of Bityr

We next provide an overview of Bityr's pipeline, which is shown in Figure 2. We motivate the design decisions for each part of the pipeline, and illustrate how they work together to yield an effective framework for binary type inference.

Although the example binary discussed above for Figure 1 is x64, a key goal of Bityr is architecture independence over the input binary: Bityr should not only support a wide range of architectures, but also be easily extensible to new ones. To achieve this objective, Bityr targets VEX IR [41], which is an architecture-agnostic representation for a number of different target machine languages. Moreover, VEX IR is designed to make program analysis easier, which enables us to leverage off-the-shelf program analysis tools. In particular, Bityr builds upon the angr binary analysis framework [53] to obtain data-flow information, as described in Section 4.

Data-flow information is highly relevant for type inference. This is evident in traditional constraint-based techniques wherein the typing constraints essentially encode such information [10, 36, 42].

Table 1: A qualitative comparison among existing binary type inference techniques. All techniques support inferencing primitive types, which are omitted in this table. "Struct," "Struct Ptrs," and "Struct Members" refer to whether each technique can *automatically* infer such information. IDA and Ghidra only support manually specifying struct types to variables and perform extremely limited automated type inference of structs. TypeMiner does not attempt to recover the complete struct layout or types of all struct members. DIRTY only predicts types in its vocabulary, which means it does not support predicting structs that did not appear in its training set.

Category	Technique	Input	Completeness	Struct Types Inference Support	Multi-arch. Support
Rules and Heuristics	IDA [3]	Binary	High	N/A	✓
	Ghidra [2]	Binary	High	N/A	✓
	Rewards [37]	Dyn. Traces	Low	Struct, Struct Ptrs, Struct Members	✗
	Howard [54]	Dyn. Traces	Low	Struct, Struct Ptrs, Struct Members	✗
Type Constraint Solving	TIE [36]	Binary	High	Struct, Struct Ptrs, Struct Members	✗
	Retypd [42]	Binary	High	Struct, Struct Ptrs, Struct Members	✗
	OSPREY [69]	Binary	High	Struct, Struct Ptrs, Struct Members	✗
Machine Learning	Debin [32]	Disassembly	High	Struct	✗
	TypeMiner [38]	Dyn. Traces	Low	Struct, Struct Ptrs	✗
	StateFormer [44]	Runtime Values	High	Struct, Struct Ptrs	✓
	DIRTY [12]	Decompilation	High	Struct, Struct Ptrs	✗
	Bityr	Binary	High	Struct, Struct Ptrs	✓

```
int file_has_ext(char* file_name, char* file_ext) {
  char* ext = file_ext;
  if (*file_name) {
    while (*ext) {
      int name_len = strlen(file_name);
      int ext_len = strlen(ext);
      if (name_len >= ext_len) {
        char* a = file_name + name_len - ext_len;
        char* b = ext;
        while (*a && toupper(*a++) == toupper(*b++));
        if (!*a) return 1;
      }
      ext += ext_len + 1;
    }
  }
  return 0;
}
```

```
...
53:  mov    -0x30(%rbp),%eax
56:  movslq %eax,%rdx
59:  mov    -0x2c(%rbp),%eax
5c:  cltq
5e:  sub    %rax,%rdx
61:  mov    -0x38(%rbp),%rax
65:  add    %rdx,%rax
68:  mov    %rax,-0x20(%rbp)
6c:  mov    -0x28(%rbp),%rax
70:  mov    %rax,-0x18(%rbp)
74:  nop
...
```

```
-0x18 (%rbp): char*
-0x20 (%rbp): char*
-0x28 (%rbp): char*
-0x2c (%rbp): int32
-0x30 (%rbp): int32
-0x38 (%rbp): char*
-0x40 (%rbp): char*
```

Figure 1: Left: A C function that checks file extensions. Middle: The disassembly abstract of the function in compiled x64 binary. Right: Type predictions for variables at their corresponding stack offsets.

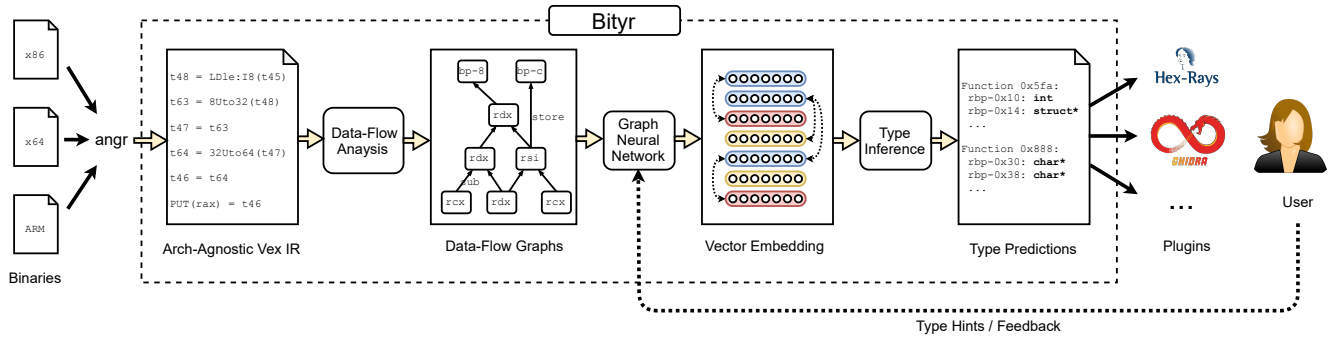

Figure 2: Bityr's pipeline. Binaries are first converted to VEX IR followed by data-flow analysis to yield a data-flow graph for each function. Each such graph is then passed to a graph neural network which yields a continuous representation that aims to capture the semantic information for its function. This representation is then used to predict a type for each variable present.

However, these methods are often limited by the constraint-solving step, which prevents them from effectively scaling to large binary applications. An attractive work-around is to employ machine learning. Although binaries lack sophisticated abstractions, they are in

fact rich in patterns and conventions, making them amenable to statistical data-driven methods. Bityr thereby uses a model to learn the data-flow patterns in binaries and output typing predictions accordingly.

To integrate classic data-flow analysis and modern machine learning, we must design a representation for typing information that is simultaneously easy to extract while being suitable for machine learning. Our key insight is to design a graph-based intra-procedural representation of data-flow information. First, constraint-encoded data-flow information is also naturally modeled through graphs, and in fact light-weight data-flow graphs are easy and efficient to acquire using ANGR. Moreover, modern graph neural networks (GNNs) are remarkably well-suited to learning and approximating the latent semantics of graph-structured data. This motivates the central data structure of Bityr, which is an efficiently constructed and information-rich graph that explicitly marks the derivation, usage, and location of data-flow throughout program execution. In short, we use ANGR to generate function-level data-flow graphs that are fed to a graph neural network. The graph neural network then generates a continuous embedding of the data-flow graphs that approximate the underlying typing semantics.

Type inference is then cast as a classification problem [46], in which we output from a range of finite C-level types. Although the possible types are in principle arbitrarily many, we observe that selecting a much smaller range of commonly seen types already encompasses a large portion of those that exist in the wild. Therefore, rather than incorporating the full complexity of structured prediction, the formulation of type inference as a classification problem suffices for binaries.

Bityr's output is a mapping of binary-level variables to their respective C-level types. This is easily interpretable as this closely matches the type systems of popular tools like IDA and GHIDRA, and is therefore also in a format that is easy to integrate with existing analysis loops. Furthermore, feedback in the form of type hints, provided by the user or another tool, can be used to refine and improve typing prediction: if a variable type is known, this amounts to simple feature engineering in the graph neural network.

4 METHODOLOGY

In this section we give a technical presentation of Bityr's approach to binary type inference as a machine learning problem. We first present our choice to target VEX IR in order to achieve architecture independence. Next, we describe our light-weight data-flow analysis procedure for generating information-rich data-flow graphs. Then, we present the design and rationale of our graph neural network architecture, as well as the distributed, vectorized embedding that it outputs. Finally, we show how Bityr treats type inference as a classification problem.

4.1 The VEX IR

Bityr first converts its input binaries into VEX IR via the ANGR binary analysis framework [53]. By targeting an IR rather than a specific architecture, Bityr accepts binaries that target a wide range of hosts. In particular, we chose the VEX IR because it backs a wide array of architectures, is designed with program analysis in mind, and has existing tooling support [1, 35].

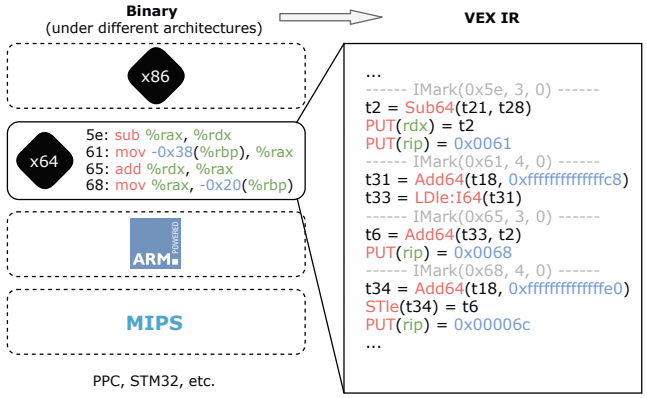

Figure 3: Binary to VEX IR conversion for lines 0x5e - 0x68 of Figure 1. Note that all architecture-specific side effects (e.g., changing `rip` in x64) are explicitly encoded in VEX IR.

As an example of this binary-to-VEX conversion, Figure 3 shows how a few lines of x64 binary are converted into their corresponding VEX snippet. VEX is a minimalistic IR in which the core semantics center around register, memory, and temporary variable read-writes. Temporary variables (`t2`, `t6`, etc) are indexed by non-negative integers and are a VEX-specific convention to enforce that valid VEX programs must be in static-single assignment form [17], which is a significant simplifying assumption for many program analysis techniques. Additionally, VEX encodes for operations such as 32-bit arithmetic, bit-wise logic, and floating-point arithmetic, whose semantics are appropriately implemented in ANGR.

4.2 Data-Flow Analysis

In this part we present how Bityr uses data-flow analysis to yield graphs that capture the relevant information for type inference. As an example, consider the function `foo` and its control-flow graph shown in Figure 4. Depending on the parameter `a`, either path P_1 or P_2 will be taken, which will appropriately modify the values of the two local stack variables `b` and `p`.

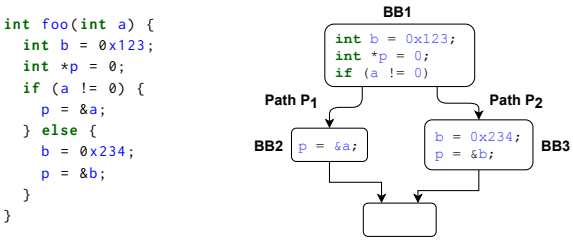

Figure 4: Example for illustrating our data-flow analysis. (Left) A simple function with two possible paths. (Right) The control-flow graph of the function.

Our objective is to infer the types of `a`, `b` and `p`, and to do this we aim to generate information-rich data-flow graphs such as those in Figure 5. These graphs are intended to convey how variables derive and use data during execution, and aim to capture sufficient

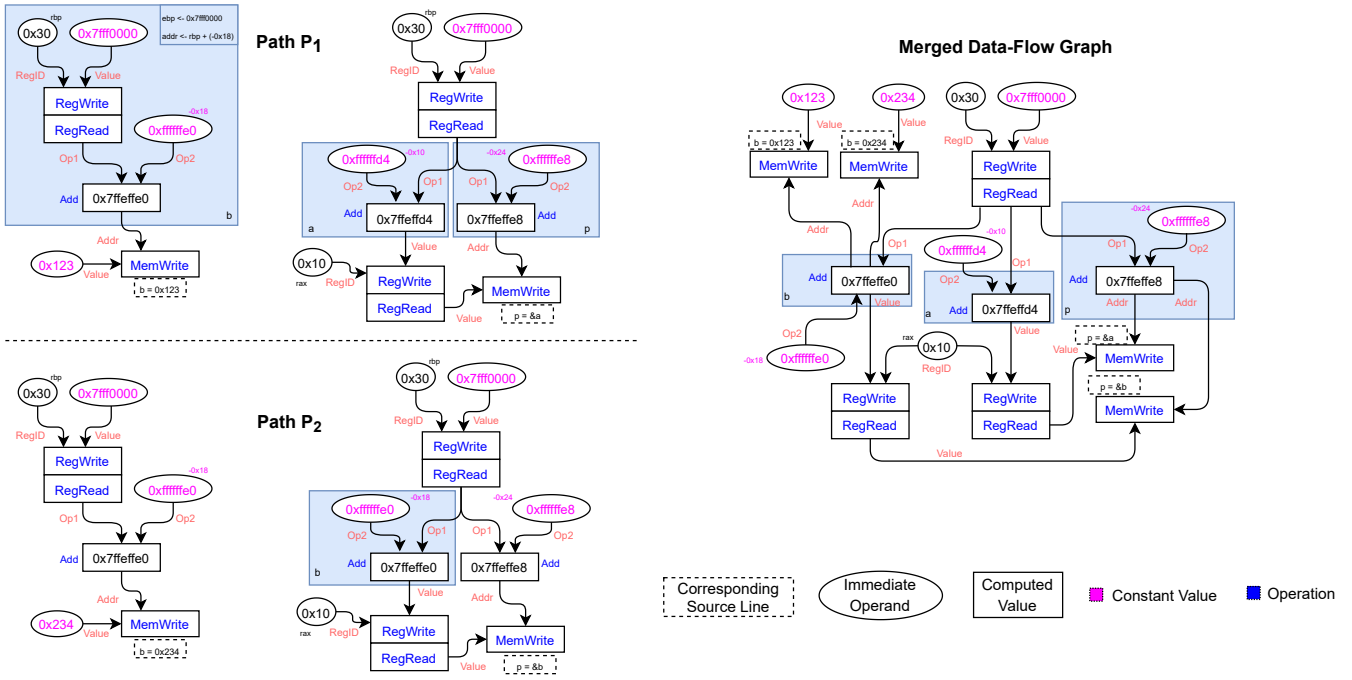

Figure 5: Data-flow graphs for the function `foo` in Figure 4. (Left-Top) variable-level data-flow graphs for `b` and `p` along P_1. (Left-Bottom) variable-level data-flow graphs for `b` and `p` along P_2. Note that because `b` was over-written, it has the value `0x234` rather than `0x123`. (Right) aggregation of all data-flow graphs. Above is a simplified view; data-flow graphs track the operands, operators, bitsizes, and locations of data derivation and usage.

information for a GNN to perform accurate type inference. The choice to target data-flow information is not arbitrary, and is in fact inspired by classical constraint-based type inference schemes, wherein constraints effectively encode these data-flow properties. As a high-level overview, our strategy for data-flow analysis is two-part, as follows:

1. Perform program execution along different non-cyclic paths in the function's control-flow graph to generate a data-flow graph for each variable along each path. Each path is then associated with a collection of variable-level data-flow graphs (Figure 5, left).
2. Aggregate the variable-level data-flow graphs of each path together into one big function-level data-flow graph (Figure 5, right). This in turn is passed to the graph neural network stage of Bityr's pipeline.

In the remainder of this part, we first discuss the relevant details and features of program execution in ANGR. Then, we show how these features are used over a single function in order to derive a *state cache*. Finally, we show how this state cache is used to extract variable-level data-flow graphs at the final state of each path, which are then aggregated into a single function-level data-flow graph.

4.2.1 Program Execution in ANGR. Because we use ANGR's execution engine to track data-flow facts, it is important to understand some of its core mechanics. First, ANGR's execution engine works at the granularity of bitvectors, and has the ability to track *bitvector expressions* throughout program execution. Execution centers around modifying a *state*, which maps locations to bitvector expressions, or more formally:

$$state : loc \rightarrow bvexpr, \qquad loc := reg \mid bvexpr$$

where *loc* may be a register (e.g. `rax`, `rbx`) or a bitvector expression for an address (e.g. `rbp - 0x20`).

As a concrete example, we consider executing a sequence of VEX instructions corresponding to the x64 assembly `add %rax, -0x20(%rbp)`, with the execution trace shown in Figure 6.

VEX IR Statement	State	Properties
		`st0[rbp] == w`
	`st0`	`st0[rax] == y`
`t0 = Get(rbp)`	`st1`	`t0 == w`
`t1 = Sub64(t0, 0x20)`	`st2`	`t1 == w - 0x20`
`t2 = Load32(t1)`	`st3`	`t2 == st2[w - 0x20]`
`t3 = Get(rax)`	`st4`	`t3 == y`
`t4 = Add64(t2, t3)`	`st5`	`t4 == st2[w - 0x20] + y`
`Store(t1) = t4`	`st6`	`st6[w - 0x20]` `== st2[w - 0x20] + y`

Figure 6: An example of execution in ANGR. Temporary variables are given by `t0`, `t1`, etc. and are in SSA form. (Left) The VEX instructions corresponding to `add %rax, -0x20(%rbp)`. (Middle) The states induced by each successive instruction, i.e. `st4 == st3.step(t3 = Get(rax))`. (Right) The relevant properties of each state.

We assume that the initial state `st0` has registers `rbp` and `rax` registers containing the bitvectors `w` and `y`, respectively. Executing this sequence of instructions ultimately stores a bitvector expression `st2[w - 0x20] + y` at the memory address `w - 0x20`, which is itself a bitvector expression.

Bitvector expressions are useful because their syntactic structure documents the derivation process of the resulting value, in particular allowing us to inspect what operators were used. In addition, ANGR allows arbitrary Python objects to annotate the nodes of a bitvector expression tree, meaning that we can easily track where data is loaded from and written to—this is especially helpful when attempting to align data with DWARF [15] debug information.

4.2.2 Exploring the Control-Flow Graph. We now discuss our strategy for using ANGR's execution on the control-flow graph in order to generate data-flow graphs. The high-level idea is to execute along different path of the control-flow graph and then, by inspecting the read-from and written-to locations of each respective state, we can extract *variable-level data-flow graphs* that provide the derivation of a particular bitvector expression at a particular location. Because our objective is to perform intra-procedural data-flow analysis, we need a careful strategy when exploring the control-flow graph in order to efficiently achieve good coverage.

One key insight is that it suffices to evaluate the basic block at each node only *once*. This is because within a basic block the same sequence of instructions will always be run regardless of *how* (i.e., path) the execution reaches it, and we want to capture the flow of data through a basic block without considering how execution reaches the basic block.

In addition, we may completely disregard path feasibility. This is because we are interested in how data may be used on *both* sides of a conditional branch. Even if a particular path is not feasible, any usage of program variables is still valuable information for type inference.

Our technique is shown in Algorithm 1. The idea is to execute the basic blocks in a particular sequence where initialization before each call to `execOneBlock` is determined by a *state cache*, which is a mapping of *node* $\rightarrow$ *state*. This cache is important because it stores the final states of all the basic blocks that we have explored, meaning that the read-from or written-to locations of each of its states contains information rich bitvector expressions, which we later use to derive variable-level data-flow graphs.

Our particular sequence of node execution is determined using notions of pre- and post-dominance [17] on the control-flow graph. In particular, we define a partial order on the nodes depending on which nodes *must* precede each other. As ANGR ensures that all functions have a unique entry block, this node ordering scheme within *worklist* ensures that we never cache miss when calling `initState`. In addition, because who explores a block is irrelevant, any final state of an explored predecessor suffices for initialization.

The sequence of nodes visited by Algorithm 1 induces a set of simple paths. The *final state* of each execution path correspond to the states in the cache which were *not* used to for initialization. As a consequence, this means that each final state has traversed a basic block that no other final state has.

4.2.3 Data-Flow Graphs. We derive data-flow graphs from the set of final states in the state cache.

Algorithm 1: Execution of a single function.

Input: The function's *entry* node and a *worklist* of nodes to explore in sequence
Output: A populated *cache* : *node* $\rightarrow$ *state*

```
1  Function execOneBlock(state, node)
2      for instr ∈ node.block.instructions do
3          state ← state.step(instr)
4      return state

5  Function initState(node, cache)
6      if there is a CFG predecessor pred of node such that
         pred ≠ node and pred ∈ cache then
7          state_0 ← cache[pred]
8          state_0[ip ↦ node.addr]
9          return state_0, cache
10     return ⊥, cache

11 state_0 ← freshZeroState()
12 state_0[ip ↦ entry.addr,  ip ↦ HIGH_ADDR]
13 state_f ← execOneBlock(state_0, entry)
14 cache[entry ↦ state_f]
15 for node ∈ worklist do
16     state_0, cache ← initState(node, cache)
17     if state_0 ≠ ⊥ then
18         state_f ← execOneBlock(state_0, node)
19         cache[node ↦ state_f]
```

By examining the read-from or written-to locations of each path's final state, we obtain a set of bitvector expressions that are used to derive the variable-level data-flow graphs shown in Figure 5 (left). The nodes of these data-flow graphs are indexed by ANGR's immutable bitvector expressions, and correspond to either *immediate operands* (e.g., constants, register offsets) that are the argument to some *operation*, or the *computed value* (e.g., to-be-written values) that result from said operation. Operations include categories such as 32-bit addition and memory-write, and together with incoming *edge labels* denote how a particular node—which corresponds to a bitvector expression—is derived.

The variable-level data-flow graphs describe only how a particular bitvector expression at a particular location in a particular path's final state is derived. Individually, they do not convey how a variable's data, or even partially-derived values, are used by other variables throughout the function. Indeed, different variable-level data-flow graphs may share identical sub-graphs, and inter-variable data-flow gives additional hints about what a bitvector expression's type might be. This motivates aggregating all the variable-level data-flow graphs into a single function-level data-flow graph, as shown in Figure 5 (right). Since we use ANGR's bitvector expressions to index nodes, this is simply a graph union.

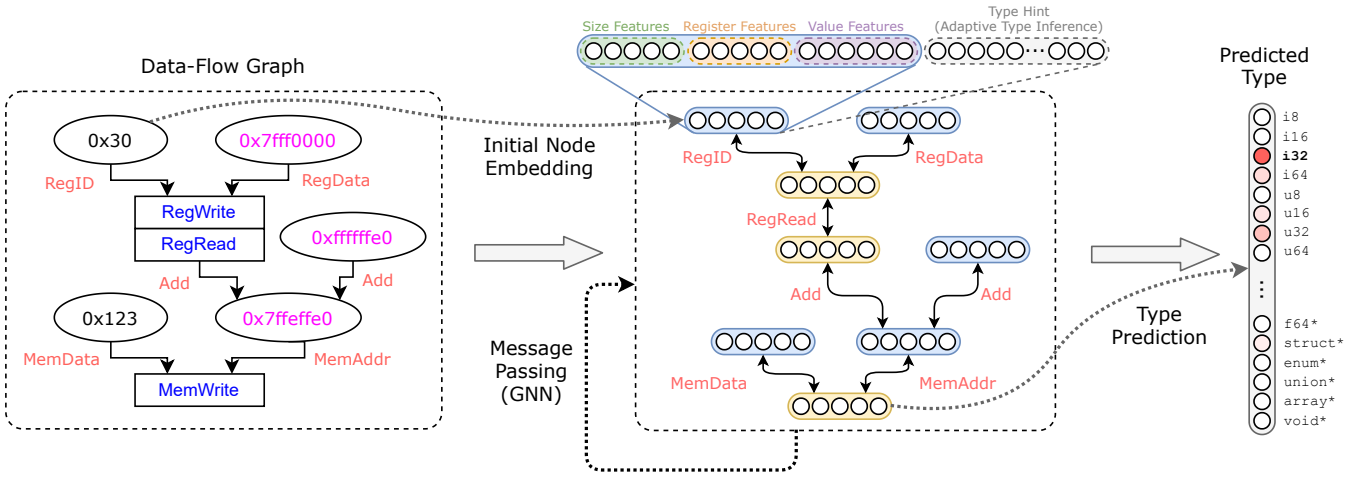

Figure 7: Architecture of graph neural network in Bityr. The data-flow graph (left) is converted into the vector representation (middle). The nodes in the data-flow graph is transformed into node embedding using Node Embedding Initialization. The directed edges are augmented to allow message passing on both forward and backward directions. After message passing, the node embedding is passed to the type prediction layer to produce the type vector (right).

4.3 Type Inference with Graph Neural Networks

Given a graph $G = (V, E)$ that contains a set of nodes V and edges E, a graph neural network (GNN) f in our context would embed the graph into a set of vectors (or embeddings), i.e., $f(G) : \mathcal{G} \mapsto \mathbb{R}^{|V| \times d}$. Here $\mathcal{G}$ represents the space of the graphs, while d specifies the dimensionality of the embedding per each node. As shown in Figure 7, the GNN encodes the graph in an iterative fashion, where each iteration or layer of GNN propagates the information from nodes to their direct neighbors. We next elaborate the specific design choices of f.

Node Embedding Initialization. The first layer of the GNN starts with the initial embedding representation of each node $h_v^{(0)}, \forall v \in V$. In our setting, we represent the node with the following simple features (Figure 7):

- Bitvector size: one-hot encoding of the size of the node bitvector value, from the set of possible sizes $\{1, 8, 16, 32, 64, 128, \texttt{others}\}$. Note that it is possible to have an irregular sized bitvector, usually due to `SHIFT` operations. In such cases, we use the encoding for `others`.
- 5 register related features, including *is_register*, *is_arg_register*, and *is_ret_register*.
- 11 value features related to the concrete node value, such as *is_bool*, *is_float*, *close_to_stack_pointer*, *is_zero*, *is_negative*, and *is_one*.

We denote the above features as $x_v \in \mathbb{R}^D$ where D is the dimension of the features. Then, the initial embedding is $h_v^{(0)} = W_0 x_v + b_0$ where $W_0 \in \mathbb{R}^{d \times D}$ and $b_0 \in \mathbb{R}^d$ are learnable parameters.

Edge Type. In our setting, each edge $e \in E$ is a triplet $e = (u, r, v)$ that represents a directional edge of type r from node u to v. The type of the edge represents the data-flow, control-flow, and other operational meanings in pre-defined types $\mathcal{R}$. Note that for a GNN to function properly, one would need to create a backward edge for any forward edge in the original data-flow graph. The backward edge type must be different than the forward edge type. Therefore for any edge type r, we have an edge type $\text{rev}(r) \in \mathcal{R}$ representing its backward edge type. As the total number of possible types $|\mathcal{R}|$ is known beforehand, we can design the message passing operator based on the edge types, as described next.

Message Passing Layer. Each layer of the GNN f performs a "message passing" operation that propagates the information from the nodes to their direct neighbors. We denote the embedding of node v at layer l as $h_v^{(l)}$, with the boundary case of $h_v^{(0)}$ defined above, and the update formula defined recursively as follows:

$$h_v^{(l)} = \sigma\Big(\text{AGGREGATE}(\{g(h_u^{(l-1)}, r, h_v^{(l-1)})\}_{e=(u,r,v) \in \mathcal{N}_v})\Big) \quad (1)$$

Here σ is an activation function such as ReLU or Sigmoid. AGGRE--GATE is a pooling function that aggregates the set of embeddings into a single vector. $\mathcal{N}_v$ denotes all incoming edges to node v, while the function $g(u, r, v)$ is the message function that produces an embedding. We adopt the design choice from RGCN [49], and realize Eq 1 as follows:

$$h_v^{(l)} = \text{ReLU}\Big(\sum_{r \in \mathcal{R}} \text{MEAN}(\{W_r^{(l)} h_u^{(l-1)} + W_0^{(l)} h_v^{(l-1)}\}_{e \in \mathcal{N}_v^r})\Big) \quad (2)$$

where $W_r^{(l)} \in \mathbb{R}^{d \times d}$ are weights that depend on layer index l and edge type r, and $W_0^{(l)} \in \mathbb{R}^{d \times d}$. $\mathcal{N}_v^r \subseteq \mathcal{N}_v$ denotes the incoming edges to node v with edge type r.

After L layers, we use the output of the last layer as the vector representation for each node, $h_v = h_v^L$, and this vector is used for label prediction, as described next.

Type Prediction. After obtaining the embedding h_v for a particular node v, we use a multi-layer perceptron (MLP) to classify h_v

into the node label, which is the type corresponding to the node. Given a set of types T, our type prediction layer produces a vector $t_v \in \mathbb{R}^{|T|}$ for the node v, as shown as the right most vector in Figure 7. During training, our predicted type vector t_v is then compared with the ground truth type vector $\hat{t}_v \in \mathbb{R}^{|T|}$, the one-hot encoding of the ground truth type under the set of types T. In this work, we apply cross entropy loss function

$$\mathcal{L}(y, \hat{y}) = -\sum_i \hat{y}_i \log(y_i) + (1 - \hat{y}_i) \log(1 - y_i)$$

to compute the loss $l = \mathcal{L}(t_v, \hat{t}_v)$. The loss l is then back-propagated to update the learnable parameters. During testing and prediction phases, on the other hand, we apply argmax on t_v to obtain the type that is predicted to have the highest probability. Note that we predict types for all nodes in the graph. During both training and testing phases, since we are aware of the mapping from source-level variables to graph nodes, we only compare to ground truth the predicted types of the nodes that correspond to source-level variables.

5 IMPLEMENTATION AND SETUP

Bityr comprises 7k lines of Python code. The data-flow analysis module is based on the ANGR framework [53]. The learning module is written using PyTorch Geometric library of PyTorch 1.8.1. All the experiments are run on a Linux server with Ubuntu 20.04, Intel Xeon Gold 5218 at 2.30GHz with 64 cores, 251GB of RAM, and two NVIDIA GeForce RTX 3090-Ti GPUs.

Dataset. We use the set of 33 programs provided by STATEFORMER [44] as our binary dataset. The dataset contains well-known projects like `coreutils`, `openssl`, and `sqlite`, cross-compiled into 4 architectures (x64, x86, ARM, and MIPS) and 4 optimization levels (O0, O1, O2, and O3). Our data generation pipeline takes in non-stripped binaries in this binary dataset and performs data-flow analysis on each function to obtain data-flow graphs. Using DWARF information, we label graph nodes that correspond to source-level variables with their actual source-level types. We omitted a few binaries and functions from the original STATEFORMER dataset for the following reasons. (i) Due to higher optimization levels abstracting away variable definitions, our data generation pipeline might not detect all source-level variables, resulting in discrepancies in the number of such variables – we omitted those functions. (ii) We also omit functions for which no local variables are detected and binaries and functions on which ANGR failed. Table 2 provides detailed statistics across all four architectures. We apply an 8:1:1 training, validation, and testing split for each architecture-optimization combination.

Typing Statistics. While a program may in principle contain infinitely many types, we found that in practice a handful of types are significantly over-represented. Figure 8 shows a breakdown of the type composition for stack variables found in the DWARF debug information of our dataset. In particular, types such as struct pointers, `u64`, `char*`, `i32`, and `u32` are relatively common. The statisics suggest that it is practical for a machine learning approach to treat type inference as *classification* rather than *structured prediction*. Indeed, our choice of output types, as shown in Figure 9, precisely covers 97.1% of all observed types in the dataset. For the types that are not exactly matching any type in our output domain, we simplify it to the closest type. For example, `struct***` is casted into `void*`.

Arch.	Opt. Level	# Binaries	# Functions	# Variables
x64	O0	1,088	226,055	937,174
	O1	1,019	202,220	799,132
	O2	997	204,056	816,838
	O3	994	203,091	813,677
x86	O0	1,044	182,243	708,040
	O1	1,036	176,415	647,490
	O2	1,063	176,441	659,920
	O3	1,062	175,351	655,734
ARM	O0	633	76,141	361,296
	O1	633	41,548	159,568
	O2	630	40,123	163,752
	O3	628	39,911	168,525
MIPS	O0	620	56,584	266,596
	O1	624	26,356	102,151
	O2	628	25,798	104,515
	O3	627	25,724	108,278

Table 2: Statistics of our dataset.

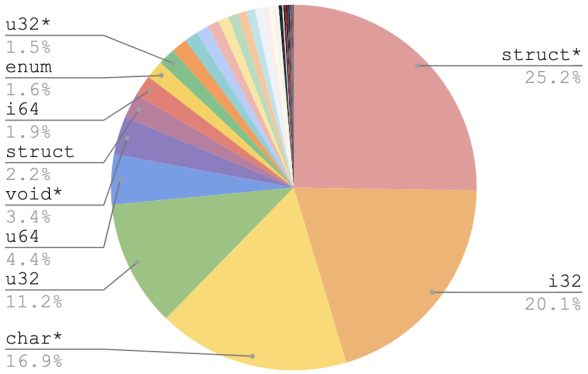

Figure 8: Breakdown of types in our dataset.

base type ::= `i8` | `i16` | `i32` | `i64` | `i128` |
`u8` | `u16` | `u32` | `u64` | `u128` |
`bool` | `char` |
`struct` | `union` | `enum` | `array`

output type ::= *base type* | *base type*`*` | *base type*`**` | `void*`

Figure 9: Description of output types.

Learning. In all our experiments, we use the Adam optimizer with initial learning rate 10^{-3} and batch size 32. We train our model end-to-end with 50 epochs and pick the model with the lowest validation loss. We use ReLU as the activation function during message passing and the type prediction. Finally, our GNN is configured to

Arch.	Opt. Level	Precision	Recall	F1 score
x64	O0	91.5	91.0	91.3
	O1	93.4	93.1	93.2
	O2	92.1	91.8	91.9
	O3	92.0	91.7	91.9
x86	O0	89.7	89.1	89.4
	O1	91.6	91.3	91.4
	O2	90.6	90.2	90.4
	O3	90.7	90.4	90.5
ARM	O0	85.5	85.0	85.3
	O1	79.2	78.4	78.8
	O2	77.2	76.5	76.9
	O3	78.3	77.4	77.9
MIPS	O0	80.5	79.9	80.2
	O1	76.8	76.2	76.5
	O2	79.4	78.8	79.1
	O3	76.4	76.0	76.2

Table 3: Bityr's precision, recall, F1 score results for each architecture and optimization.

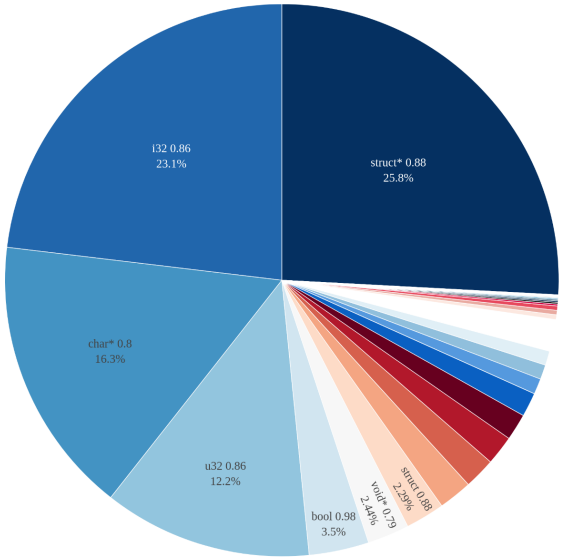

Figure 10: Inference precision for arm O0.

have eight ($L = 8$) message passing layers, with latent dimension $d = 64$.

6 EVALUATION

Our evaluation aims to investigate the effectiveness of our approach by answering the following questions:

RQ1 How accurate is Bityr's type inference on real-world binaries?

RQ2 How does Bityr compare to existing binary type inference techniques?

RQ3 How much does accurate data-flow information improve the performance of Bityr?

RQ4 How efficient is Bityr's type inference engine?

6.1 RQ1: Type Inference Accuracy of Bityr

We first investigate the performance of Bityr on various architectures and optimization levels. For each architecture, we first trained on all the available data (e.g. x64-OAll) before testing on individual optimization levels (e.g. x64-O1). Table 3 shows the precision, recall, and F1 scores. Bityr achieves average F1 scores of 92.1% on x64, 90.4% on x86, 79.7% on ARM, and 78.0% on MIPS. The F1 scores on ARM and MIPS are lower than those on x64 and x86. We believe that this gap is due to different numbers of functions in training sets for each architecture. As shown in Table 2, there are significantly more binaries and functions in x64 and x86 than in ARM and MIPS in the dataset. We also perform a manual inspection of a randomly sampled set of generated data-flow graphs for each architecture, and find that they are similar. This means more data is very likely to improve the performance of Bityr on ARM and MIPS binaries. We verify this experimentally and provide the details in Appendix A.1.

6.1.1 Accuracy across Types. Here, we present the accuracy of Bityr w.r.t types, specifically, how the accuracy varies across different types. Figure 10 provides detailed inference results for ARM-O0. As expected, the accuracy positively correlates with the size, i.e., the number of occurrences (Figure 8), of the corresponding type in the dataset. For instance, the precision for `struct*` is 0.88, and it represents 25.8% of the entire dataset. However, an intriguing observation is that although `bool` (boolean) type occupies a very small portion of the dataset, 3.5% in this case, it achieves high precision. This is because boolean variables are usually as flags for branches, and consequently, the same bool variable may occur multiple times for different program paths. This results in more than twice the number of location nodes and edges, such as `RegRead` node and `Value` edge in Figure 5, with respect to other variable nodes. Furthermore, these bool nodes are found to be connected to a greater number of edges with labels related to comparison operations such as `__eq__`. The trend is the same for other architectures and optimization combinations. We present the complete results in Table 7 in Appendix.

6.2 RQ2: Comparison Against Baselines

There are a number of published techniques that focus on binary type inference, such as TIE [36], TypeMiner [38], and Retypd [42]. Unfortunately, many of them are not publicly available. We compare against the performance of several most recent techniques that are publicly accessible: DIRTY [12], STATEFORMER [44] , which use transformer neural network for type inference; DEBIN, which uses probabilistic decision trees; and OSPREY [69], which uses a probabilistic type constraint solving for type recovery. We omit comparisons against commercial tools (Ghidra and IDA) because OSPREY outperforms both of them.

6.2.1 Comparison with DEBIN and STATEFORMER. We run Bityr directly on the dataset provided by STATEFORMER and derive results

Arch.	Opt.	Bityr	StateFormer	Debin
x64	O0	91.3 (+9.9)	81.4	73.8
	O1	93.2 (+18.3)	74.9	
	O2	91.9 (+21.8)	70.1	
	O3	91.9 (+20.6)	71.3	
x86	O0	89.4 (+4.9)	84.5	63.7
	O1	91.4 (+19.8)	71.6	
	O2	90.4 (+17.6)	72.8	
	O3	90.5 (+8.9)	81.6	
ARM	O0	85.3 (+7.2)	78.1	67.4
	O1	78.8 (+1.7)	77.1	
	O2	76.9 (+1.5)	75.4	
	O3	77.9 (-12.5)	90.4	
MIPS	O0	80.2 (-15.0)	95.2	–
	O1	76.5 (+0.8)	75.7	
	O2	79.1 (+5.7)	73.4	
	O3	76.2 (+0.4)	75.8	

Table 4: Comparison on F1 scores among Debin [32, Table 3], StateFormer [44, Table 2], and Bityr. Both StateFormer and Bityr are evaluated on the same data set, while the Debin results were reported by their authors on their own data set.

for comparison with Stateformer. Debin takes stripped binaries as input and outputs binaries augmented with DWARF debug information. However, Debin performs both type inference *and* source-level variable recovery, which means that the amount of variable-type pairs present was often small. We could not run Debin to only conduct type inference. Hence, we directly take the numbers reported by Debin [32, Table 3]. For StateFormer, we also directly compared Bityr with the results in their paper because we are using the same dataset. Both Debin and StateFormer use precision (P), recall (R), and F_1, where

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad R = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad F_1 = \frac{2 \cdot P \cdot R}{P + R}$$

and TP, FP, and FN denote true-positive, false-positive, and false-negative respectively. We follow the same metric for comparison purposes. As F_1 is a mixture of precision and recall, we chose to compare F_1 scores among Bityr, Debin and StateFormer. We are also use micro-averages for F1 scores, as all prior work like is using this metric and the result is showed in Table 4.

StateFormer achieves an average 78.1% F1 score across all architecture, optimization while Bityr achieves an average 85.1% F1 score and outperforms StateFormer by 7.0%. To be noticed, StateFormer does not predict double pointer type, while Bityr is capable of predicting it.

Debin is evaluated on a private dataset that includes x64, x86, and ARM binaries, but not MIPS. Since their evaluation does not distinguish between different optimization levels, we group them together in our comparison. Moreover, Debin predicts a strictly smaller set of types than Bityr. It does not differentiate among

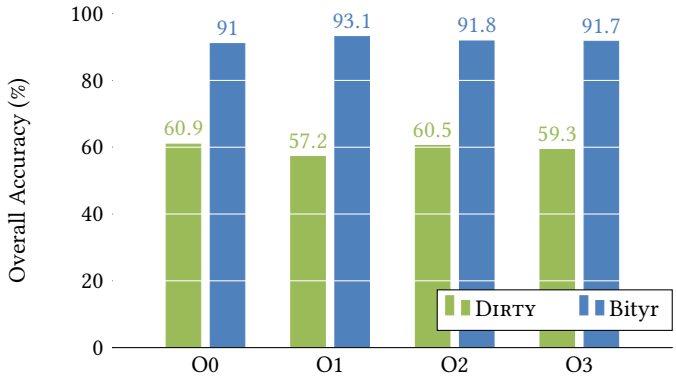

Figure 11: Overall accuracy of Bityr and Dirty on the StateFormer x64 dataset.

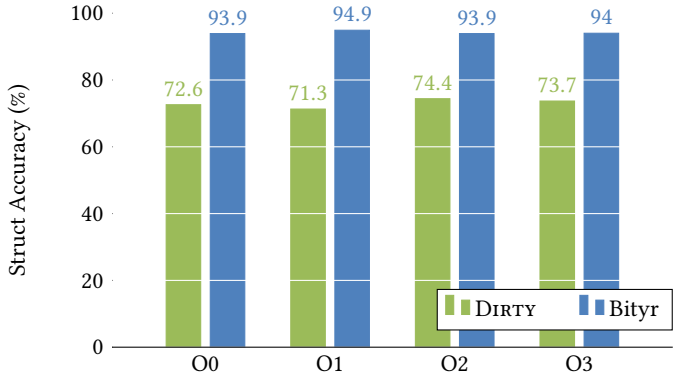

Figure 12: Accuracy of predicting `struct` types for Bityr and Dirty on the StateFormer x64 dataset.

pointer types: Pointers to structs (`struct*`) and pointers to chars (`char*`) are uniformly labeled as `pointer`.

StateFormer is evaluated on the same binaries as Bityr, but employs a different method for identifying binary-level variables. In particular, StateFormer considers *every* token in a stream of assembly as candidate for type inference—it is the model's job to predict that certain instructions, such as `nop` have the sentinel type of *no-access*. This level of granularity means that there is significant redundancy in how StateFormer counts variables [44, Table 1], and also means that many of StateFormer's type predictions are not useful for an end-user without additional post-processing. Additionally, at StateFormer's token-level granularity many types can be known from local instructions, e.g., it should be clear that one of the operands to the ARM instruction `ldr` is a pointer. Furthermore, because StateFormer relies on a neural architecture to *learn* the semantics of program execution, it is not clear that StateFormer can accurately track long-range data and typing-dependencies.

6.2.2 Comparison with Dirty and OSPrey. To ensure a fair evaluation, we only compare against those configurations that are used in the evaluation of the corresponding tools. Dirty is trained only on x64 binaries, hence we only test Dirty on x64 binaries

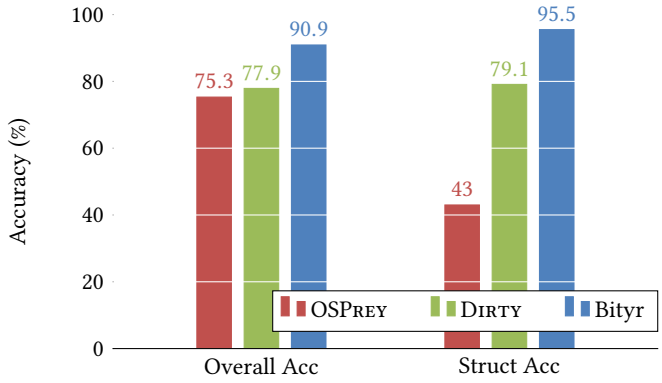

Figure 13: Accuracy results on GNU coreutils O0 executables

from StateFormer dataset. Since Dirty can predict up to 48,888 different types, we convert its predictions in a post-hoc manner to make it comparable with Bityr. For example, Dirty predicts struct's inner types, but Bityr does not; therefore, the structure type `struct S{int x; float y;}` is converted to `struct`. Also, Dirty is evaluated by accuracy, which we compare against Bityr's accuracy. Figure 11 shows the overall type prediction accuracy of Dirty in comparison with Bityr. Figure 12 shows the accuracy for predicting only struct types. These results show that Bityr identifies types with ~30% higher accuracy. Even for struct types (for which Dirty is specialized), Bityr outperforms Dirty by over 20%.

The implementation of OSPrey is not publicly accessible. However, the authors of the paper provided results of OSPrey on O0 binaries of GNU Coreutils. We removed Coreutils functions that are part of Bityr's training set to ensure a fair comparison, as including these functions might inflate Bityr's results. As Dirty was compared against OSPrey, we also evaluated Dirty on the same dataset (Coreutils O0 binaries). Because OSPrey only predicts several primitive and complex types, we convert both Dirty and Bityr's predicted types in a post-hoc manner to have comparable results. Specifically, type names and field names are discarded. For example, `bool` and `char` are both converted to `Primitive_1`, which stands for a primitive type occupying 1 byte of memory, and `const char *` and `char *` are converted to `Pointer`. Because Bityr does not predict types for members of structs, we also discarded such information from OSPrey and treated related types as struct type.

Figure 13 shows the the accuracy of both overall types and only struct types for all three tools. Bityr outperforms both OSPrey and Dirty. Specifically, Bityr is 15.6% more accurate than OSPrey in terms of overall type prediction, and more than 50.0% more accurate when predicting struct types. In conclusion, our results show that Bityr is more effective than the state-of-the-art machine-learning type inference techniques by outperforming them by a considerable margin.

6.3 RQ3: Impact of Accurate Data-Flow Information

To demonstrate the importance of using accurate data-flow information, we evaluated a version of Bityr that used a basic reach-def graph of VEX IR registers, which can be considered as *syntactic* flow representation. Note that, unlike our data-flow graph, this simple syntactic representation did not track memory dependencies. We evaluated the syntactic version of Bityr with the accurate data-flow version on a small percentage of the dataset and on X64, X86, and ARM architectures. Table 5 shows the results. As expected, the syntactic version of Bityr (without memory dependencies) performed poorly on RISC architectures (e.g., ARM) that make heavy use of loads, stores, and data moving between registers and memory. The results clearly demonstrate the necessity of using accurate data-flow information for accurate type inference on binary code.

	X64	X86	ARM
Syntactic (Top-3)	72.3 (88.6)	71.6 (87.5)	34.6 (59.3)
Accurate (Top-3)	77.2 (92.8)	77.1 (91.6)	66.1 (88.5)
Δ (Top-3)	+4.9 (+4.2)	+5.5 (+4.1)	+31.5 (+29.2)

Table 5: The impact of accurate data-flow analysis (Accurate) and light-weight syntactic data-flow analysis (Syntactic) on the accuracy of type inference (both top-1 and top-3 accuracy). Note that this experiment was performed on a smaller subset of GNU Coreutils binaries, hence the difference between numbers reported here and the ones reported in Table 3.

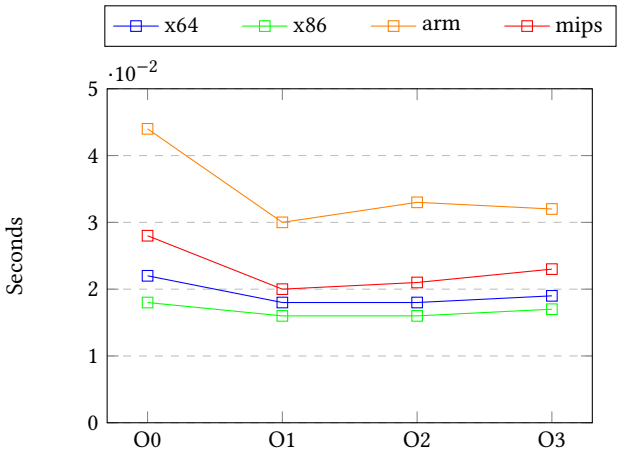

Figure 14: Inference speed per function

6.4 RQ4: Efficiency of Bityr

In this section, we measure the inference performance of Bityr. Specifically, we measure each function's inference time and memory consumption, and the average numbers for each architecture are shown in Figure 14 and 15, respectively.

6.4.1 Inference Time. The inference time per function is reasonable and ranges from 1.5 - 4.5 seconds. The average inference time (Figure 14) follows a similar trend for all architectures, i.e., slightly higher for O0, whereas similar for all other optimization levels. This is because O0 binaries (i.e., without optimization) have more variables than in higher optimization levels as the variables

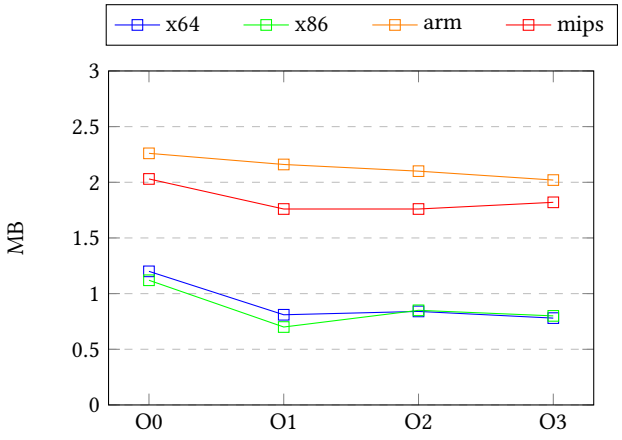

Figure 15: Inference memory cost per function

get optimized out. It is interesting to see that the average time for ARM is slightly higher than for other architectures. This is because our variable inference identified more variables than necessary and consequently performed more work than necessary by predicting types for non-existent variables. Nonetheless, the absolute time is relatively less at 4.5 seconds.

6.4.2 Memory Consumption during Inference. The Figure 15 shows the average memory consumption across all architectures. As expected, it follows a similar trend as the inference time. It is interesting to note that memory consumption is higher for RISC architectures, i.e., ARM and MIPS, compared to CISC architectures, i.e., x86 and x64. This is because of the additional load and store instructions in RISC, leading to more nodes in the data-flow graph and, consequently, higher memory consumption. Similar to inference time, the absolute memory consumption is reasonable at 2 MB.

Project	# Variables	Runtime(CPU)			
		Bityr	Stateformer	Debin	Ghidra
ImageMagic	23727	208	187.8	N/A*	664.3
PuTTY	22429	143	146	5239.8	514.2
Findutils	6534	43	23.7	849	83.3
zlib	730	6	3.3	119	11.7

*Debin terminates abruptly after running one of the binaries for 138 minutes.

Table 6: Execution time on CPU (in seconds) of Bityr, STATEFORMER, Debin, and Ghidra on 4 software projects with different number of varibles.

We present the inference time on four software projects in comparison with StateFormer, Debin and Ghidra in Table 6. We chose these projects as they were used to measure the inference time in StateFormer. Bityr performs in par with StateFormer with slight difference. This shows that Bityr achieves higher precision with the same inference time.

6.4.3 Bityr Graph Building (or Training) Time. We also measured the time it takes to build data flow graphs necessary for training. Figure 16 shows a scatter plot of graph-building time on x64 Coreutils O0 binaries against functions according to the number of variables. A detailed distribution of graph-building time is presented in Appendix 17. It takes less than ten sections for 80.42% of functions and 93.37% of functions complete within 30 seconds. For most cases, as expected, the graph build time is proportional to the number of variables. However, there are some cases where this is not the case. For instance, it took 400 seconds to build the data flow graph for a complex function with few variables. This is because the graph-building time also depends on the complexity of the dataflows (e.g., nested loops and conditionals).

The trend is the same across other architectures and compilation options.

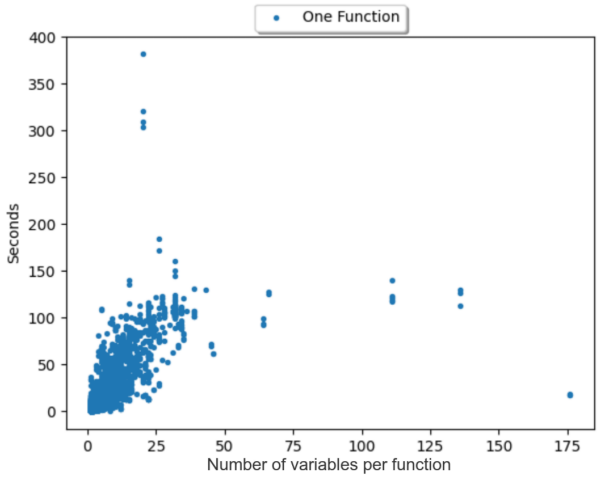

Figure 16: Graph building speed for x64 Coreutils O0.

7 DISCUSSION

Our experiments demonstrate that using graph neural networks to learn and apply data-flow patterns for type inference is competitive with other machine learning approaches as well as industry-standard tools like IDA. We now discuss the main limitations of Bityr and how to overcome or mitigate them.

- *Choice of Program Variables.* Bityr focuses on predicting the types of local variables of functions. It is easy to extend our implementation to handle global variables in a similar manner. In particular, it necessitates combining data-flow graphs from different functions that use the global variable.
- *Choice of Type Information.* Bityr assigns one of a finite number of types to each program variable. In particular, it does not support rich compound types; for instance, although it can identify types like `struct*`, it cannot infer the fields of a `struct`. Likewise, it cannot infer polymorphic types in the style of some constraint-based methods [42]. While our typing scheme already offers significant coverage, one could extend Bityr to predict richer types by using structured prediction instead of classification.
- *Scope of Data-Flow Analysis.* Our data-flow analysis only examines intra-procedural data-flow. We expect that an inter-procedural analysis will yield richer input data for the learning model, and

thus better performance. However, inter-procedural analysis is potentially expensive, and an excessive amount might offset the scalability benefit of using a machine learning approach. Nevertheless, we believe this to be a fruitful direction of investigation.

- *Type Hints of External Functions.* Many analysis tools (including ANGR) know the types of standard C library functions ahead of time. Our implementation does not take advantage of these types to improve prediction accuracy, but it would be straightforward to support since Bityr can incorporate type hints.
- *Indirect Jump Target Resolution.* Our analysis relies on ANGR to construct control-flow graphs. While ANGR can accurately compute the targets of direct jumps, estimating the targets of jumps that involve dynamic computation is much harder. As a result, we may end up never exploring certain parts of a function. Fortunately, the learning model can cope with such missing information, or fall back to the user for type hints.

8 RELATED WORK

This work primarily focuses on type inference applied to binaries, which are compiled object files such as ELF/PE files. Specifically, we focus on data type inference, i.e., recovering simple types for the identified variables.

Static analysis, specifically constraint solving, is one of the commonly used approaches. These techniques work by first introducing types based on specific rules and then propagate this seed information to different entities (variables/registers or memory objects) based on the program's data-flow [40]. Some works focus on inferring a limited set of types such as signed/unsigned integers [65], strings [13], `struct` types [56]. On the other hand, works such as TIE [36] and Retypd [42] attempt to infer a more comprehensive set of types specified using a type-lattice. These techniques seed their algorithms by assigning types based on certain base rules. For instance, an operand for `load` or `store` instruction should be of pointer type. They then use propagation techniques either based on Value Set Analysis [7] or constraint solving to propagate these seed types to all other entities.

On the other hand, best-effort techniques [22, 29] that are based on heuristics suffer from precision. Furthermore, most of these techniques are specific to each architecture, such as x64, x86, ARM, etc. Although TIE [36] and Retypd [42] try to be architecture-agnostic by using an IR such as BIL [9], not all architectures (e.g., MIPS) are supported by BIL. Finally, none of these techniques are available as open-source [10], which makes it hard to evaluate or extend them transparently. There are other techniques specific to C++ [26], where the main goal is to determine the classes and layout of objects. These techniques are not directly applicable as they mainly focus on recovering object-oriented features [66] such as class hierarchy [27, 50] and virtual table layout [19].

Some techniques use dynamic analysis [16, 30, 34, 68], wherein type propagation is usually done by taint tracking, and finally combine results from different executions to determine the type of a variable. However, the effectiveness of these techniques depends on the feasibility of executing the program and the availability of high coverage test cases, which is not easy, especially for libraries, embedded programs, and network-based programs.

Machine learning (ML) techniques have been explored in the context of binary analysis, popular applications being vulnerability detection [24, 24, 43, 59], function identification [8, 47, 52, 58], and code clone detection [18, 23, 25, 45, 61–63]. Most of these works use traditional ML models such as SVMs. However, recent works [52, 61] have started using Neural networks, especially Recurrent Neural Networks (RNNs). ML techniques are also used for semantic problems such as type inference. CATI [11] uses word2vec to predict types based on usage contexts. Similarly, EKLAVYA [14] uses RNNs to predict function signatures, including types of the arguments. DEBIN [32] uses probabilistic models to predict debug information (types and names of variables) in stripped binaries. They use a dependency graph to encode uses of identified variables and then convert them into feature vectors and then train a model based on Extremely Randomized Trees [28]. STATEFORMER [44] sidesteps the problem of feature selection by using transformers on micro execution traces to learn the instruction semantics as pre-trained models. These pre-trained models are further used to perform type prediction. Similarly, DIRTY [12] also uses a transformer model for type prediction. The recent technique, OSPREY [69] tries to combine both constraints solving and machine learning. Unfortunately, as shown in Table 1, none of these techniques have multi-architecture support and require considerable effort to extend to a new architecture. Finally, our comparative evaluation in Section 6.2 shows that Bityr outperforms all these techniques.

In contrast, Bityr uses data-flow analysis to precisely capture intra-procedural data-flow graphs and encodes them using graph neural networks, which in turn perform type inference via classification.

9 CONCLUSION

We presented Bityr, a pluggable framework for binary type inference. Bityr uses data-flow analysis to precisely track the data flow of program variables in an architecture-agnostic manner. The data-flow information is encoded using graph neural networks which then perform type inference as a classification task. We evaluated Bityr on 33 software projects, and demonstrated that it can predict fine-grained types for source-level program variables with reasonably high accuracy. Furthermore, Bityr can adapt effectively to feedback, is easy to extend to different architectures, and scales well to support interactive use-cases.

REFERENCES

[1] Adding new architecture to vex ir. http://angr.io/blog/throwing_a_tantrum_part_4/.
[2] Ghidra. https://ghidra-sre.org/.
[3] Ida pro. https://hex-rays.com/ida-pro/.
[4] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.
[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. 2018.
[6] Shushan Arakelyan, Sima Arasteh, Christophe Hauser, Erik Kline, and Aram Galstyan. Bin2vec: learning representations of binary executable programs for security tasks. *Cybersecurity*, 4(1):1–14, 2021.
[7] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.
[8] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[10] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35, 2016.

[11] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98. IEEE, 2020.

[12] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium*, Boston, MA, August 2022.

[13] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 88–95, 2005.

[14] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[15] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4, 2010.

[16] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[17] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[18] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[19] David Dewey and Jonathon T Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *NDSS*, 2012.

[20] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

[21] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.

[22] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 51–60, 2013.

[23] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[25] Qian Feng, Rundong Zhou, Yanhui Zhao, Jia Ma, Yifei Wang, Na Yu, Xudong Jin, Jian Wang, Ahmed Azab, and Peng Ning. Learning binary representation for automatic patch detection. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2019.

[26] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.

[27] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of class hierarchies for decompilation of c++ programs. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 240–243. IEEE, 2010.

[28] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[29] I Guilfanov. Simple type system for program reengineering. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 357–361. IEEE, 2001.

[30] Philip J Guo, Jeff H Perkins, Stephen McCamant, and Michael D Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 255–265, 2006.

[31] Christophe Hauser, Yan Shoshitaishvili, and Ruoyu Wang. Poster: Challenges and next steps in binary program analysis with angr.

[32] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[33] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 57–67. IEEE, 2016.

[34] Changhee Jung and Nathan Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–66, 2009.

[35] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364. IEEE, 2017.

[36] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

[37] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, page 18, San Diego, CA, Feb 2010.

[38] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 288–308. Springer, 2019.

[39] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS22)*, ASIACCS 22, June 2022.

[40] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[41] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[42] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–41, 2016.

[43] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 450–459. IEEE, 2015.

[44] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *IEEE S&P*, 2021.

[45] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[46] Sakthi Kumar Arul Prakash and Conrad S Tucker. Node classification using kernel propagation in graph neural networks. *Expert Systems with Applications*, 174:114655, 2021.

[47] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Machine learning-assisted binary code analysis. In *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security, Whistler, British Columbia, Canada, December*. Citeseer, 2007.

[48] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[49] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018.

[50] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441, 2018.

[51] Edward J Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, page 17. USENIX Association, 2013.

[52] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[53] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[54] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, Feb 2011.

[55] Yihao Sun, Jeffrey Ching, and Kristopher Micinski. Declarative demand-driven reverse engineering. *arXiv preprint arXiv:2101.04718*, 2021.

[56] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *2010 10th IEEE Working Conference on*

Source Code Analysis and Manipulation, pages 179–188. IEEE, 2010.
[57] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1875–1892, 2020.
[58] Shuai Wang, Pei Wang, and Dinghao Wu. Semantics-aware machine learning for function recognition in binary code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 388–398. IEEE, 2017.
[59] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.
[60] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.
[61] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.
[62] Hongfa Xue, Guru Venkataramani, and Tian Lan. Clone-hunter: accelerated bound checks elimination via binary code clone detection. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 11–19, 2018.
[63] Hongfa Xue, Guru Venkataramani, and Tian Lan. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, pages 27–33, 2018.
[64] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.
[65] Qiuchen Yan and Stephen McCamant. Conservative signed/unsigned type inference for binaries using minimum cut. 2014.
[66] Kyungjin Yoo and Rajeev Barua. Recovery of object oriented features from c++ binaries. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 231–238. IEEE, 2014.
[67] Bin Zeng. Static analysis on binary code. Technical report, Tech-report, 2012.
[68] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. 2012.
[69] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 813–832. IEEE, 2021.

A APPENDIX

Fig 17 shows the process time for graph building on x64 coreutils O0. 93.37% of functions finished graph building process within 30 seconds and 97.35% of functions complete the process within 60 seconds.

A.1 Impact of Data on Accuracy

To analyze the effect of data, we varied the size of the training data for arm O0 and measured the change in the performance of our models. Training data was subsampled at rates of 1%, 5%, 10%, and increased by 10% for the rest. The results of these experiments are shown in figure 18. The training data size is plotted on the x-axis, while accuracy is plotted on the y-axis. As we can see, increasing the data has a positive impact on the accuracy. This provides evidence for our observation that low accuracy on ARM and MIPS is due to a smaller dataset.

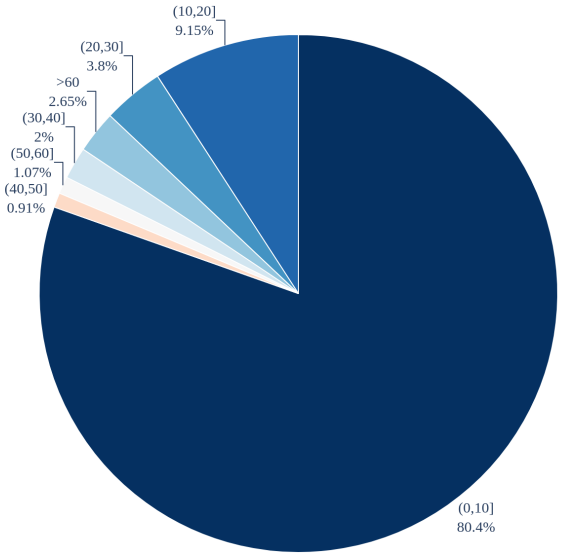

Figure 17: Process time distribution for graph building on x64 Coreutils O0.

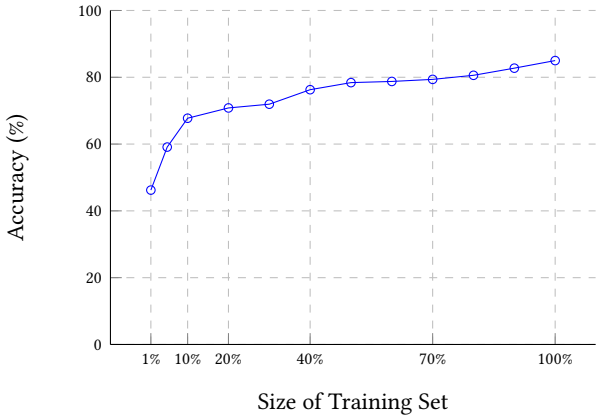

Figure 18: Accuracy of Bityr on arm O0

Table 7: Inference precision for each type on different optimization levels and architectures.

	Precision															
	X64				MIPS				X86				ARM			
Type	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**
struct*	0.94	0.95	0.94	0.94	0.88	0.77	0.78	0.76	0.93	0.93	0.93	0.93	0.88	0.84	0.83	0.81
char*	0.86	0.9	0.88	0.88	0.8	0.66	0.66	0.62	0.86	0.88	0.86	0.87	0.8	0.72	0.76	0.76
u32	0.92	0.93	0.93	0.94	0.8	0.75	0.73	0.75	0.92	0.93	0.93	0.93	0.86	0.83	0.82	0.79
u64	0.93	0.94	0.93	0.94	0.82	0.88	0.68	0.77	0.86	0.89	0.88	0.87	0.88	0.89	0.81	0.81
i32	0.91	0.82	0.78	0.76	0.81	0.63	0.63	0.61	0.82	0.8	0.79	0.78	0.86	0.67	0.7	0.69
array<void>	0.88	0.85	0.89	0.84	0.74	0.77	0.71	0.74	0.77	0.79	0.81	0.77	0.79	0.75	0.78	0.81
struct	0.95	0.95	0.94	0.96	0.8	0.76	0.78	0.81	0.96	0.95	0.95	0.96	0.88	0.81	0.82	0.82
i64	0.89	0.94	0.92	0.89	0.7	0.82	0.96	0.83	0.89	0.94	0.93	0.93	0.78	0.78	0.7	0.79
void*	0.64	0.7	0.66	0.65	0.78	0.72	0.61	0.63	0.63	0.84	0.83	0.75	0.79	0.51	0.51	0.58
bool	0.96	0.95	0.95	0.95	0.94	0.82	0.65	0.74	0.93	0.92	0.91	0.92	0.98	0.89	0.85	0.86
struct**	0.81	0.84	0.86	0.87	0.72	0.71	0.6	0.66	0.81	0.87	0.84	0.86	0.8	0.76	0.58	0.65
enum	0.85	0.96	0.88	0.9	0.74	0.67	0.63	0.81	0.78	0.94	0.9	0.91	0.75	0.63	0.65	0.87
u64*	0.84	0.86	0.8	0.81	0.82	0.78	0.68	0.81	0.62	0.78	0.79	0.8	0.93	0.98	0.69	0.85
char**	0.73	0.85	0.84	0.8	0.78	0.67	0.59	0.61	0.77	0.86	0.85	0.84	0.81	0.78	0.65	0.72
char	0.9	0.88	0.82	0.9	0.92	0.66	0.67	0.66	0.79	0.94	0.85	0.76	0.93	0.84	0.57	0.41
u32*	0.84	0.89	0.87	0.89	0.84	0.83	0.82	0.83	0.72	0.82	0.69	0.77	0.81	0.83	0.82	0.87
union*	0.76	0.84	0.9	0.84	0.81	0.52	0.65	0.63	0.82	0.92	0.94	0.79	0.74	0.76	0.59	0.62
i32*	0.67	0.58	0.71	0.55	0.68	0.61	0.32	0.45	0.57	0.7	0.43	0.41	0.72	0.54	0.45	0.4
f64	0.86	0.85	0.85	0.74	0.89	0.2	0.53	0	0.83	0.86	0.53	0.39	0.77	0.51	0.55	0.39
i64*	0.76	1	0.94	0.97	0.72	0.58	0.7	0.49	0.92	0.99	0.97	0.86	0.96	0.96	0.96	0.82
f32*	0.85	0.5	0.18	0.4	0.7	0	0.03	0.02	0.75	NA	NA	NA	0.85	0.41	0.46	0.11
u16	0.86	0.96	0.84	0.72	0.8	1	0	0	0.82	0.91	0.87	0.74	0.9	0.21	0	0.31
u16*	0.83	0.79	0.85	0.87	0.24	0.94	0.22	0.64	0.64	0.93	0.9	0.76	0.2	0.82	0.47	0
enum*	0.81	0.99	0.87	0.94	0.73	0	0	0	0.83	1	1	0.99	0.81	0.65	0.75	0.96
union	0.72	0.75	0.8	0.84	0.84	0.94	0.4	0.77	0.2	1	0.5	0.22	0.64	0.19	0.9	0.92
f32	1	0	0	0	0.96	0	NA	NA	0.58	NA	NA	NA	0.82	0	0	0.67
void**	0.99	1	0.97	0.86	0.91	0.81	1	0.66	0.5	1	0.38	0.72	0.8	0.79	0.33	1
i16*	0.99	0.9	0.97	1	0.86	0.8	0	NA	0.55	0.94	0.97	0.69	0.61	NA	0.89	0.95
union**	0.7	0.73	0.71	0.68	0	NA	NA	0	0.72	0.94	0.97	0.88	0	0	NA	0
f64*	0.77	0	0.76	0.9	0.42	1	0.53	0	0.79	0.09	0.14	0.2	0.6	0.47	0.96	0.9
i16	1	NA	0.95	0.98	0.82	0	0	0	0.94	0.7	0.83	0	0.92	0	0	0
i64**	0.98	1	1	1	NA	NA	NA	NA	NA	0	NA	NA	0	0	NA	NA

*NA suggests the variable type is not used in the given dataset.