Bityr: A Pluggable Framework for Learning Types from Binaries

Abstract—**We propose Bityr, a framework for inferring fine-grained type information from binaries. Bityr aims to be usable for security analysts by employing a machine learning based approach. 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 which is resource-efficient and capable of incorporating feedback in the form of type hints. We have implemented Bityr atop the ANGR framework which targets VEX IR. On a suite of 33 open-source software projects compiled for x64, x86, ARM, and MIPS architectures with different optimization levels, Bityr achieves accuracy of 77%, 77%, 66%, and 61% respectively. Moreover, accuracy progressively increases even further as type hints are provided.**

I. Introduction

Binary type inference is one of the core computational challenges in binary analysis. It concerns identifying the data types of registers and memory values in a stripped executable (or object file). The resulting information is useful in reconstructing source-level abstractions that greatly benefit many applications such as reverse engineering, malware analysis, vulnerability detection [12], patching, re-hosting, and other security-critical applications [10, 21, 47]. However, automated and precise binary type inference is a hard problem [32, 66] due to the lack of high-level abstractions and the rich variety and sophistication of compiler optimizations, hardware architectures, and adversarial obfuscation methods [54].

A predominant use-case of type information, and even more so for binaries, is program understanding. Many important security applications are necessarily interactive [55, 57]. For instance, a reverse engineer or security analyst uses an interactive decompilation tool such as 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 process of interaction continues until the analyst is satisfied with the understanding gained from the semantic information.

In this work, we aim to develop a *pluggable* framework for binary type inference—i.e., a framework that is usable in an interactive manner by security analysts. We identify four key criteria that such a framework must satisfy in order to be effective in practice:

- ***Accurate.*** It should have reasonably high prediction accuracy, *especially for entities that are relevant to the analyst*, e.g., types of source-level variables instead of compiler-introduced temporaries.
- ***Adaptive to Feedback.*** It should be able to adapt to type information provided by the analyst or other tools. This is important to ensure that it is usable in interactive settings.
- ***Architecture-Agnostic.*** It should be architecture-agnostic in order to support different prevalent architectures *as well as ease of extending to new ones*.
- ***Resource-Efficient.*** It should be usable on regular machines without requiring special hardware, e.g., abundant memory, powerful GPUs, or cloud resources. Furthermore, it should scale well with the input binary size in both time and memory to support interactivity and responsiveness.

As shown in Table I, there is significant room for improving upon state-of-the-art binary type inference techniques in terms of the above criteria. Techniques based on constraint solving such as TIE [37] and RETYPD [43] yield sound and reasonably accurate types. These techniques can be easily modified by adding new constraint rules to adapt to feedback [40]. However, constraint solving techniques are not architecture-agnostic since the constraint generation phase is architecture-dependent. Furthermore, the scalability of these techniques is limited by the constraint solving phase, and consequently, inference fails for any reasonably sized real-world binary [10].

Popular interactive decompilation tools such as IDA [3] and GHIDRA [2] commonly employ heuristic-based typing methods and have low accuracy. The heuristics are necessarily architecture-dependent and identifying these type heuristics for a new architecture requires considerable effort [30] and thus cannot be easily extended to a new architecture.

Recent tools such as DEBIN [33], STATEFORMER [45], and TYPEMINER [39] employ machine learning techniques for type inference. These techniques have improved accuracy and do not rely on constraint solving nor heuristics. However, they perform type inference in a batch mode, making them harder to adapt to feedback and hindering their applicability in interactive settings. Furthermore, these techniques require non-trivial customization and substantial new training data to extend to a new architecture. For instance, STATEFORMER uses over 600 unique tokens to encode the instruction set of its supported architectures.

In this paper, we present Bityr, a learning-based binary type inference framework that satisfies all of the above criteria. Bityr targets VEX IR [42], a uniform high-level representation that abstracts away different machine architectures. On one hand, this representation enables Bityr to be architecture-agnostic and easy to extend to new architectures [1, 6, 34]. At the same time, the representation is suitable for data-flow analysis, allowing Bityr to extract data-flow information which is highly relevant for type inference.

TABLE I: Comparison of Bityr to existing approaches. For each criterion, we indicate whether the system fully (✓), partially (❍), or does not (✗) satisfy it.

System	Technique	Accurate	Adaptive to Feedback	Resource Efficient	Architecture Independent
TIE [37]	Constraint Solving	✓	❍	✗	✗
RETYPD [43]		✓	❍	✗	✗
IDA [3]	Heuristics	✗	✓	✓	✗
GHIDRA [2]		✗	✓	✓	✗
DEBIN [33]	Probabilistic Graphical Models	✓	✗	✓	❍
STATEFORMER [45]	Pretrained models and Transformers	✓	✗	❍	❍
TYPEMINER [39]	Random Forest and Support Vector Machine	✓	✗	✓	❍
Bityr	Data-Flow Analysis and Graph Neural Networks	✓	✓	✓	✓

To strike a balance between scalability and accuracy, Bityr employs a novel graph-based intra-procedural representation of the data-flow information. Since the representation is constructed on a per-function basis, Bityr scales to large real-world binaries. At the same time, this representation is acceptable input to Graph Neural Networks (GNNs) [50], a deep neural network model that is well-suited for predicting rich properties of graph-structured data [60]. In our setting, the graphs encode data-flow information obtained by a light-weight analysis over VEX IR, and the properties concern type information of registers and memory values.

GNNs have been used for many program understanding tasks such as identifying variable misuses in C# programs [5], localizing and repairing bugs in JavaScript programs [20], predicting types in Python programs [4], and detecting code clones in cross-platform binaries [61]. To our knowledge, our work is the first to demonstrate the effective application of GNNs to the problem of binary type inference. In particular, we demonstrate that they can adequately cope with missing inter-procedural information, thereby allowing the data-flow analysis to scale. They can also seamlessly incorporate type hints, rendering Bityr usable in interactive settings. Finally, they enable Bityr to be resource-efficient, both during offline training and online inference.

We have implemented Bityr atop the ANGR binary analysis framework [54] and evaluate it on a suite of 33 open-source software projects compiled for x64, x86, ARM, and MIPS architectures with different optimization levels. Bityr achieves accuracy of 77%, 77%, 66%, and 61% respectively. Moreover, the accuracy increases even further progressively as type hints are provided. We also observe that Bityr does not require intensive system resources and runs in time almost linear to program size in batch mode.

In summary, this work makes the following contributions:

- We present Bityr, an architecture-agnostic framework for binary type inference that is capable of incorporating feedback in the form of type hints.
- 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 have implemented Bityr as a framework that can be readily integrated into interactive environments and easily extended to newer architectures.
- We demonstrate the effectiveness of Bityr by extensively evaluating it on a large corpus of binaries using different architectures and compiler optimizations.
- We have made the source code of Bityr publicly available at https://github.com/bityr-sp22/bityr-sp22.

II. 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.

A. Binary Type Inference

We consider the problem of mapping binary-level variables to source-level types. Specifically, we focus on local variables in functions, which are crucial for understanding the behavior and intent of the function—and are thus of interest to a reverse engineer. We illustrate our objective with an example in Figure 1, which shows a C function and its 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 `-0x30(%rbp)` corresponds to `file_name`.

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

```c
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*
```

Fig. 1: (Left) A C function that checks file extensions. (Middle) The compiled x64 binary. (Right) Type predictions for variables at the corresponding stack offsets.

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 III-C and document the frequency of types within our dataset in Section IV.

B. *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 [42], 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 [54] to obtain data-flow information, as described in Section III.

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, 37, 43]. 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.

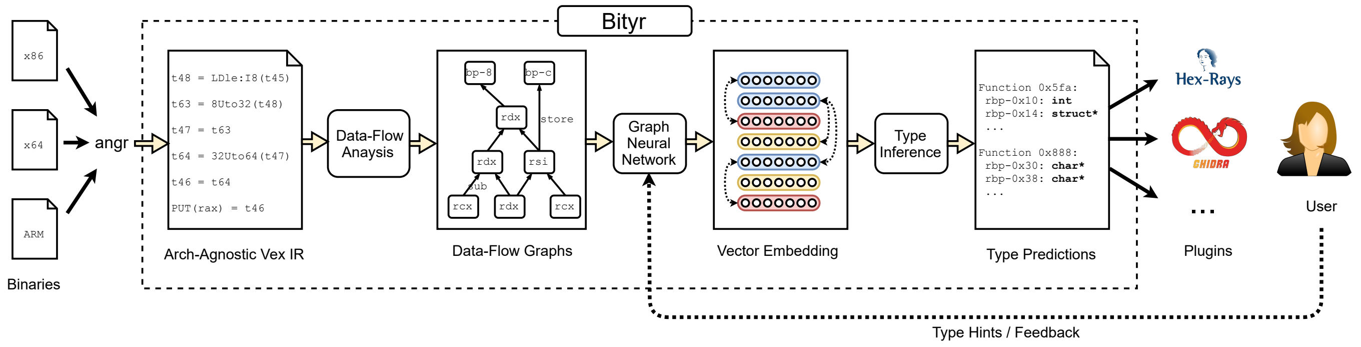
In order to integrate both classical data-flow analysis and modern machine learning, we must design a representation for typing information that is simultaneously easy to extract while being suitable for machine learning. Our key insight is to design a graph-based intra-procedural representation of data-flow 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 [48], 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.

III. 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.



```
t48 = LDle:I8(t45)
t63 = 8Uto32(t48)
t47 = t63
t64 = 32Uto64(t47)
t46 = t64
PUT(rax) = t46
```

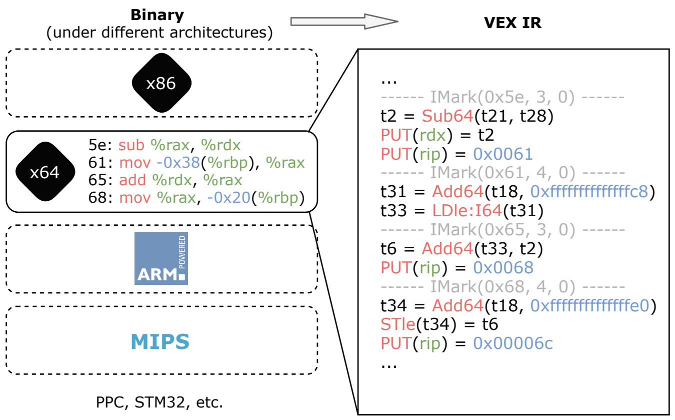
```
Function 0x5fa:
 rbp-0x10: int
 rbp-0x14: struct*
 ...

Function 0x888:
 rbp-0x30: char*
 rbp-0x38: char*
 ...
```

Fig. 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.

Finally, we show how Bityr treats type inference as a classification problem.

A. The VEX IR



```
5e: sub %rax, %rdx
61: mov -0x38(%rbp), %rax
65: add %rdx, %rax
68: mov %rax, -0x20(%rbp)
```

```
...
------ IMark(0x5e, 3, 0) ------
t2 = Sub64(t21, t28)
PUT(rdx) = t2
PUT(rip) = 0x0061
------ IMark(0x61, 4, 0) ------
t31 = Add64(t18, 0xffffffffffffffc8)
t33 = LDle:I64(t31)
------ IMark(0x65, 3, 0) ------
t6 = Add64(t33, t2)
PUT(rip) = 0x0068
------ IMark(0x68, 4, 0) ------
t34 = Add64(t18, 0xffffffffffffffe0)
STle(t34) = t6
PUT(rip) = 0x00006c
...
```

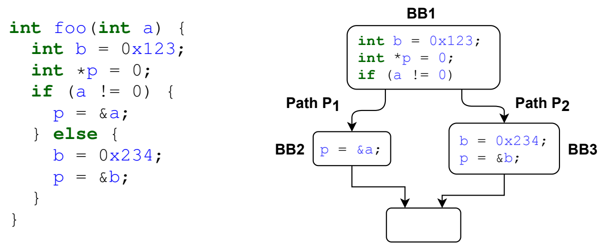
Fig. 3: Binary to VEX IR conversion for lines 0x5e - 0x68 of Figure 1. Note that all architecture-specific side effects (e.g., changing `rip` in x64) are explicitly encoded in VEX IR.

Bityr first converts its input binaries into VEX IR via the ANGR binary analysis framework [54]. 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, 36].

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.

B. 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`.



```
int foo(int a) {
  int b = 0x123;
  int *p = 0;
  if (a != 0) {
    p = &a;
  } else {
    b = 0x234;
    p = &b;
  }
}
```

```
int b = 0x123;
int *p = 0;
if (a != 0)
```

```
p = &a;
```

```
b = 0x234;
p = &b;
```

Fig. 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 `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 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. Function calls are skipped by zeroing the return register. Each path is then associated with a collection of variable-level data-flow graphs (Figure 5, left).

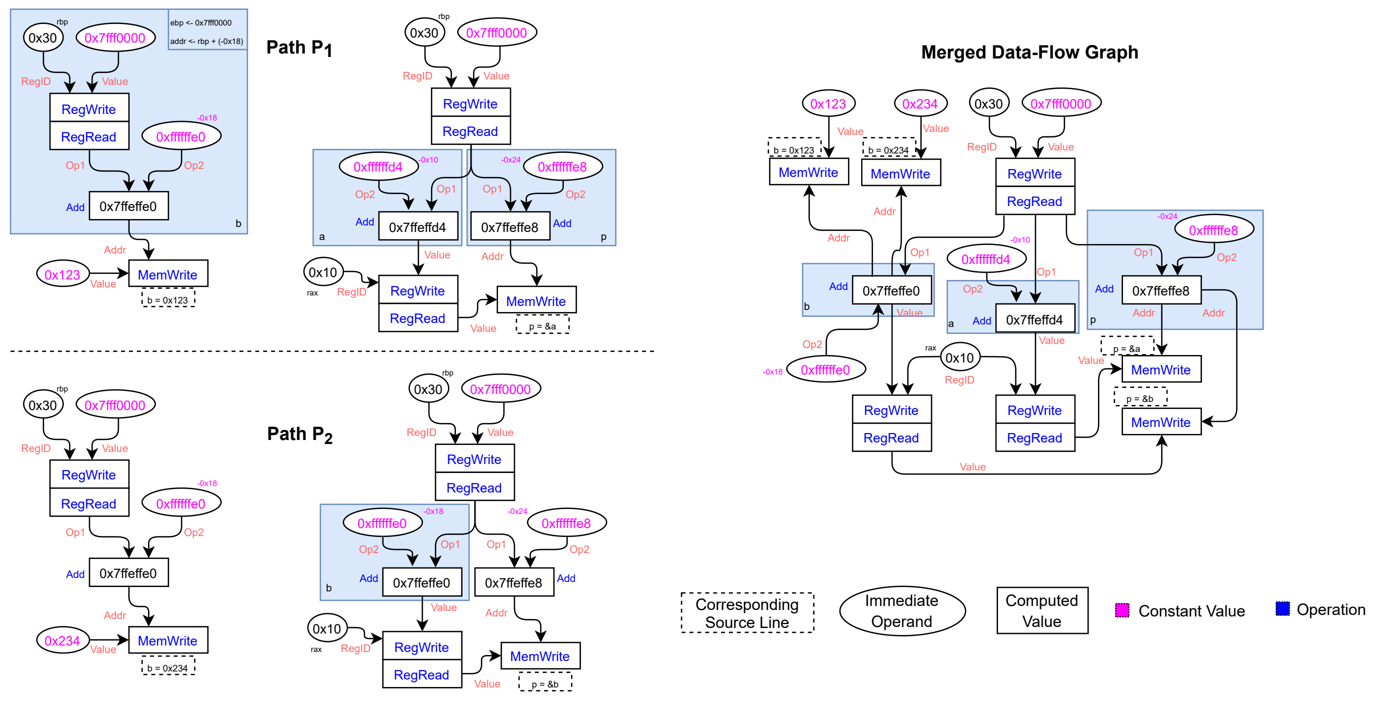


Fig. 5: Data-flow graphs for the function `foo` in Figure 4. (Left-Top) variable-level data-flow graphs for `b` and `p` along P_1. (Left-Bottom) variable-level data-flow graphs for `b` and `p` along P_2. Note that because `b` was over-written, it has the value `0x234` rather than `0x123`. (Right) aggregation 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.

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.

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:

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

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.

We assume that the initial state `st0` has registers `rbp` and `rax` registers containing the bitvectors `w` and `y`, respectively.

VEX IR Statement	State	Properties
	st0	`st0[rbp] == w` `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`

Fig. 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.

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

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 $cache : node \to state$

1  Function execOneBlock($state, node$)
2      for $instr \in node.block.instructions$ do
3          $state \leftarrow state.step(instr)$
4      return $state$

5  Function initState($node, cache$)
6      if there is a CFG predecessor $pred$ of $node$ such
         that $pred \neq node$ and $pred \in cache$ then
7          $state_0 \leftarrow cache[pred]$
8          $state_0[ip \mapsto node.addr]$
9          return $state_0, cache$
10     return $\bot, cache$

11 $state_0 \leftarrow$ freshZeroState()
12 $state_0[ip \mapsto entry.addr,\ \ ip \mapsto$ HIGH_ADDR$]$
13 $state_f \leftarrow$ execOneBlock($state_0, entry$)
14 $cache[entry \mapsto state_f]$
15 for $node \in worklist$ do
16     $state_0, cache \leftarrow$ initState($node, cache$)
17     if $state_0 \neq \bot$ then
18         $state_f \leftarrow$ execOneBlock($state_0, node$)
19         $cache[node \mapsto state_f]$
```

DWARF [15] debug information.

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 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 $node \to state$. This cache is important because it stores the final states of all the basic blocks that we have explored, meaning that the 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.

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

By examining the 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.

C. 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

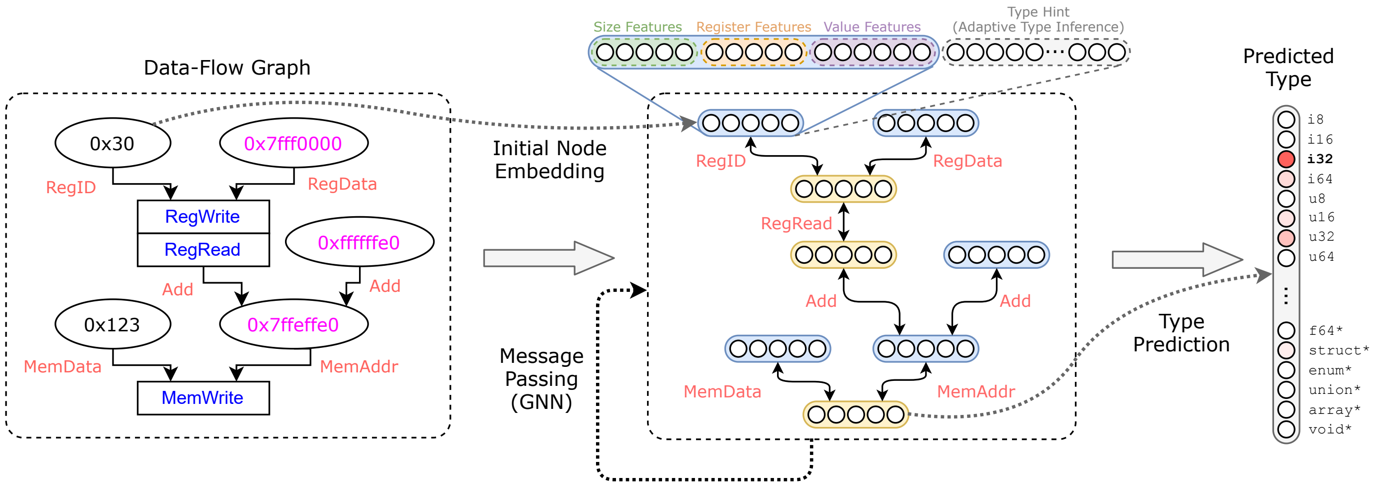


Fig. 7: Architecture of graph neural network in Bityr. The data-flow graph (left) is converted into the vector representation (middle). The nodes in the data-flow graph is transformed into node embedding using Node Embedding Initialization. The directed edges are augmented to allow message passing on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

GNN propagates the information from nodes to their direct neighbors. We next elaborate the specific design choices of f.

a) 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.

b) 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.

c) 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. AGGREGATE 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 [51], 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.

d) 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.

D. *Adaptive Type Inference*

While the above setting is already sufficient for predicting types, we can extend the initial embedding representation x_v of each node to make the whole system adaptive to external type knowledge. Along with the original features x_v, additional information about variable type is also added to the initial node embedding. We call such type information as *Type Hints*. Under a finite set of types T, for a given node v, we define its type hint $e_v \in \mathbb{R}^{|T|}$ as either

1) t, if the node v is annotated with a type and $t \in \mathbb{R}^{|T|}$ is the one-hot embedding of the type, or
2) $\vec{0}$, if the node v is not annotated with any type.

Then, we have our extended initial node embedding

$$x'_v = \begin{bmatrix} x_v \\ e_v \end{bmatrix},$$

as indicated. Despite the new dimension of the node features $D' = D + |T|$, the rest of the system stays exactly the same.

Note that this setup subsumes the previous setup, since it is possible that no type hint is provided and all the type hints in the initial node embeddings are zero. This generalized setup allows external knowledge to be incorporated, rendering Bityr adaptive to feedback. When a type hint is provided for a node, we simply encode the type as a vector, and set the corresponding features to be the type vector. This architecture is designed so that the type hint can be passed via messages through edges to help infer the types of other nodes.

IV. Implementation and Setup

a) Implementation: We implemented Bityr in 7K lines of Python code. The data-flow analysis module is based on the ANGR framework [54]. The learning module is written using PyTorch Geometric library of PyTorch. We have made our implementation available for anonymous review at https://github.com/bityr-sp22/bityr-sp22.

b) Dataset: We use the set of 33 programs provided by STATEFORMER [45] 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. 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 also omit functions for which no local variables are detected. Table II provides detailed statistics, including the average size of the graph, of the dataset for each architecture and optimization level. As can be seen from the table, there are on average 7. We apply an 8:1:1 training, validation, and testing split on each dataset. In the following sections, we use "OAll" to denote the combined dataset of O0-O3 under the same architecture.

Arch	Opt	#Bin	#Funcs	#Vars	#Nodes/Func
x64	O0	1046	38742	221914	116.9
	O1	978	17597	79718	31.1
	O2	957	18898	86986	35.5
	O3	954	18160	84151	36.4
x86	O0	982	23051	114485	67.8
	O1	939	13939	51648	34.9
	O2	978	14553	55648	38.8
	O3	979	14118	53862	38.9
ARM	O0	624	25492	155984	131.8
	O1	629	6900	30037	35.4
	O2	625	7057	33728	50.9
	O3	624	6695	32449	56.6
MIPS	O0	613	22869	136808	100.4
	O1	612	4258	19756	27.6
	O2	614	4075	19371	33.8
	O3	613	3863	18730	37.8

TABLE II: Statistics of our dataset.

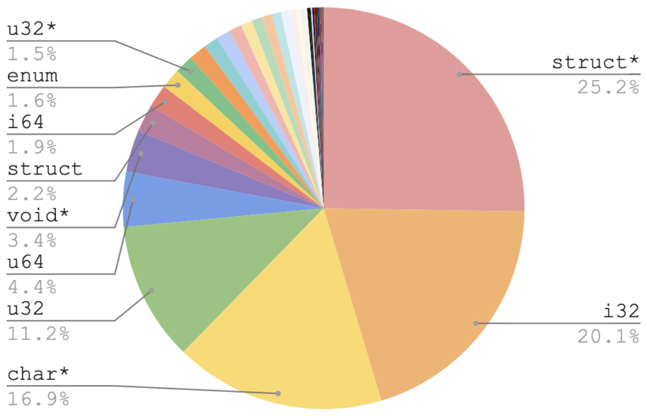


Fig. 8: Breakdown of types in our dataset.

c) 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

$$\begin{array}{rl} \textit{base type} ::= & \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{i128} \mid \\ & \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{u128} \mid \\ & \texttt{bool} \mid \texttt{char} \mid \\ & \texttt{struct} \mid \texttt{union} \mid \texttt{enum} \mid \texttt{array} \\ \textit{output type} ::= & \textit{base type} \mid \textit{base type}\star \mid \texttt{void}\star \end{array}$$

Fig. 9: Description of output types.

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*`.

d) 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 have the number of message passing layers $L = 8$, with latent dimension $d = 64$.

V. 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 approaches?

RQ3 How effective is providing type hints in improving the type inference accuracy?

RQ4 How effective is the use of data-flow analysis?

A. RQ1: Accuracy of Bityr

We first investigate the performance of Bityr on different 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). Our results are shown in Table III and include the accuracy for top-3 prediction, which checks if any of the top-3 predicted types are correct. Bityr consistently achieves 77.2% on x64 and 77.1% on x86, but has lower numbers on ARM (66.1%) and MIPS (60.9%). We believe that this gap is likely due to the amount of training data: we are able to extract significantly more variable-type pairs for training with x64 and x86 than with ARM and MIPS. However, manual inspection of the generated data-flow graphs for all architecture shows that they are rather similar, meaning that more data should strictly boost the performance on ARM and MIPS.

Architecture	Opt. Level	Acc.	Top-3 Acc.
x64	O0	77.1	94.1
	O1	76.1	90.1
	O2	76.6	91.0
	O3	79.2	92.3
	OAll	77.2	92.8
x86	O0	73.5	91.4
	O1	80.9	92.3
	O2	78.9	91.7
	O3	80.6	90.9
	OAll	77.1	91.6
ARM	O0	68.6	90.5
	O1	57.7	82.6
	O2	60.0	84.5
	O3	58.7	83.5
	OAll	66.1	88.5
MIPS	O0	62.6	88.2
	O1	42.4	76.5
	O2	58.4	84.8
	O3	59.2	84.2
	OAll	60.9	87.0

TABLE III: Accuracy results for Bityr.

B. RQ2: Comparison Against Baselines

We compare against the performance of three baselines: IDA, which uses heuristics for type inference; Debin, which uses probabilistic decision trees; and StateFormer, which uses a transformer neural network model. An exact comparison is infeasible because implementation differences between these tools result in outputs that cannot always be directly compared. We therefore report results of a best-effort comparison, first with IDA and then with Debin and StateFormer.

Arch	#Vars	Acc.	Acc. No Sign
x64	24221	43.1	61.0
x86	18251	33.1	45.1
ARM	34983	45.9	60.9
MIPS	38776	49.1	63.8

TABLE IV: IDA's performance in type inference. We use all binaries in our dataset that are under 200KB and track variable-type pairs that are detected by both us (via pyelftools) and IDA. We report IDA's accuracy for both exact predictions and predictions where integer signs are ignored.

1) Comparison with IDA*:* Because the functions from different binaries are mixed among the training, validation, and testing data, we found it challenging to decouple them such that IDA is tested on the same set as Bityr. Furthermore, we want to avoid testing Bityr on functions it was trained on. Instead of running a direct comparison between Bityr and IDA, we instead run IDA on the available dataset and observe its performance. For each architecture, we select all binaries (from x64-OAll, x86-OAll, etc.) that are under 200KB (with debug information) and produce copies that are

Arch	Opt	Bityr	StateFormer F1	Debin F1
X64	O0	77.1	81.4	73.8
	O1	76.1	74.9	
	O2	76.6	70.1	
	O3	79.2	71.3	
X86	O0	73.5	84.5	63.7
	O1	80.9	71.6	
	O2	78.9	72.8	
	O3	80.6	81.6	
ARM	O0	68.6	78.1	67.4
	O1	57.7	77.1	
	O2	60.0	75.4	
	O3	58.7	90.4	
MIPS	O0	62.6	95.2	—
	O1	42.4	75.7	
	O2	58.4	73.4	
	O3	59.2	75.8	

TABLE V: Comparison with F1 scores from Debin [33, Table 3] and StateFormer [45, Table 2]. StateFormer treats *every* token in the assembly as a candidate for type prediction. Debin's dataset does not distinguish between optimization levels and is not public.

stripped. Then, for each function in each binary, we examine its DWARF information to extract each local stack variable and its type. This is then compared to the local stack variables and their types predicted by IDA over the stripped version of each binary. In particular, we take the stack variables that IDA successfully identified *and* predicted the type for, and check what percentage of those predictions are correct. Interestingly, IDA mispredicts between signed and unsigned integers frequently, and so we also re-ran the experiments where signed and unsigned integer types of the same bitsize were treated as equal.

Our evaluation results for IDA are shown in Figure IV. We found that IDA does not have strong prediction accuracy for type inference over stripped binaries. By ignoring integer signedness, however, accuracy improves but does not become competitive with Bityr.

2) Comparison with Debin *and* StateFormer*:* We found it challenging to perform a direct comparison with Debin. Debin takes as input stripped binaries and outputs binaries augmented with DWARF debug information. However, Debin performs both type inference *and* source-level variable recovery, meaning that the amount of variable-type pairs present was often small. We were unable to run Debin to do only type inference. For StateFormer, we contacted the authors and asked for help with running their tool. Unfortunately, they are still in the process of implementing an end-to-end prediction pipeline that can output variable-type mappings, so are unable to run a direct comparison at the moment.

Given these difficulties, we instead directly take the numbers reported by both Debin [33, Table 3] and StateFormer [45, Table 2], and show the comparisons in Table V. We would like to emphasize that these numbers are the result of different experimental setups and evaluation metrics, and so should not be used for rigorous direct comparisons. Nevertheless, we believe that they are still useful in illustrating the relative performance of each tool.

	X64	X86	ARM	MIPS
No Type Hint	77.2	77.1	66.1	60.92
With At Most 2 Type Hints	83.9	83.8	78.1	77.0
With At Most 4 Type Hints	91.2	92.4	89.5	88.2

TABLE VI: Effectiveness of type hints in improving accuracy.

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. In contrast, our evaluation with Bityr focuses solely on accuracy. As F_1 is a mixture of precision and recall, we chose to compare Bityr's accuracy against Debin and StateFormer's F_1 scores.

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: namely pointers like `struct*` and `char*` are uniformly identified 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 [45, Table 1], and also means that many of StateFormer's type predictions are not useful for an end-user without additional post-processing. In addition, 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 `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.

C. RQ3: Adaptiveness of Type Hints

In this experiment, we preprocess the dataset to simulate having external knowledge provide up to n type hints per function. Given a function and a target variable that we want to predict its type, we randomly sample in the same function at most n other variables, which we call *hinting variables*. We collect the nodes in the graph that are associated with hinting variables. As described in Section III-D, we

	X64	**X86**	**ARM**
Syntactic (Top-3)	72.3 (88.6)	71.6 (87.5)	34.6 (59.3)
Data-Flow (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 VII: Effectiveness of data-flow analysis (Data-Flow) over light-weight syntactic analysis (Syntactic).

incorporate ground truth variable types in the form of type hints on the above nodes. All the other nodes have zero vector as their type hint. When there are less than n remaining variables, we pick all the other variables for type hint. For the functions with only one variable (27.8% of the whole dataset), the prediction is performed as if there is no type-hint. Note that this setup augments data, since each function in the training set now generates multiple training samples per variable—we could sample different hinting variables for the same to-predict variable.

We perform the experiment on $n \in \{2, 4\}$ and the OAll dataset for all 4 architectures. Since on average a function contains 5 variables, we stop at $n = 4$ so that the model does not possess excessive amount of information. As can be seen from Table VI, with more types being hinted by the system, the model can consistently predict better type information. When up to 2 type hints provided ($n = 2$), the accuracy goes up by 6.7% to 16.1%, with the most significant improvement observed in MIPS. This suggests that in real life, the user's effort hinting 2 variables' types can improve the prediction significantly. Nevertheless, our experiment with $n = 4$ shows that more type hints can further boost the prediction accuarcy. We don't expect the user to directly help hinting 4 variable types. However, with our adaptive type inference setup, one can imagine user using the system in the following way: A first round of type inference is performed, giving every variable a type for the user to inspect. During user's interaction with the system, they are able to inspect the top-3 predictions, confirm predictions and correct mistakes, turning type predictions into type hints. Now, the system will adapt to these external knowledge, and utilize the type hints to help the prediction of types for the remaining variables. Therefore, it is highly possible that at certain stage the system possesses even more than 4 variable types, further strengthening the type prediction.

D. RQ4: Effectiveness of Data-Flow Analysis

Our earlier versions of Bityr did not use sophisticated analysis techniques. Rather than leveraging any of ANGR's implementation of VEX IR semantics, we instead construct light-weight syntactic data-flow graphs by parsing over VEX IR instruction sequences. While our underlying goal to track data-flow was the same, this light-weight implementation means that reads and writes across registers and memory are not precisely tracked by the system. Consequently, we performed poorly on RISC architectures like ARM that make heavy use of loads and stores. Our comparison improvements are shown in Table VII, with ARM seeing the most improvement. The numbers for the earlier syntactic data-flow analysis are done over the Linux Coreutils and we did not yet have a MIPS dataset at the time.

VI. 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 [43]. 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.

VII. 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 [41]. Some works focus on inferring a limited set of types such as signed/unsigned integers [64], strings [13], `struct` types [56]. On the other hand, works such as TIE [37] and Retypd [43] 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. As shown in Table I, however, these techniques suffer from scalability issues and cannot be applied to real-world programs.

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 [37] and Retypd [43] 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 [65] such as class hierarchy [27, 52] and virtual table layout [19].

Some techniques use dynamic analysis [16, 31, 35, 67], 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, 44, 59], function identification [8, 49, 53, 58], and code clone detection [18, 23, 25, 46, 61–63]. Most of these works use traditional ML models such as SVMs. However, recent works [53, 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 [33] 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 [45] 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. Unfortunately, none of these techniques are architecture-agnostic and require considerable effort to extend to a new architecture. Finally, as shown in Table I, none of these techniques are adaptive to feedback and hence cannot be readily used in interactive settings.

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. In addition to being more accurate, our approach also enables us to adapt based on feedback, thereby enabling interactive use-cases such as reverse engineering. Recent work [38] also shows that adapting to feedback can improve the effectiveness of certain security tasks such as vulnerability detection.

VIII. 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] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[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] EN Dolgova and AV Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[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] Ilfak Guilfanov. Decompiler internals: microcode. *Black Hat USA*, 2018(S 91), 2015.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. Arbitrar: User-guided api misuse detection.

[39] 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.

[40] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473, 2015.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 311–322, 2013.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[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] Qiuchen Yan and Stephen McCamant. Conservative signed/unsigned type inference for binaries using minimum cut. 2014.

[65] 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.

[66] Bin Zeng. Static analysis on binary code. Technical report, Tech-report, 2012.

[67] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. 2012.