**NeuDep: Neural Binary Memory Dependence Analysis**

Kexin Pei  
kpei@cs.columbia.edu  
Columbia University  
New York, USA

Dongdong She  
dongdong@cs.columbia.edu  
Columbia University  
New York, USA

Michael Wang  
mia27950@mit.edu  
MIT  
Cambridge, USA

Scott Geng*  
scott.geng@columbia.edu  
Columbia University  
New York, USA

Zhou Xuan  
xuan1@purdue.edu  
Purdue University  
West Lafayette, USA

Yaniv David  
yaniv.david@columbia.edu  
Columbia University  
New York, USA

Junfeng Yang  
junfeng@cs.columbia.edu  
Columbia University  
New York, USA

Suman Jana  
suman@columbia.edu  
Columbia University  
New York, USA

Baishakhi Ray  
rayb@cs.columbia.edu  
Columbia University  
New York, USA

**ABSTRACT**

Determining whether multiple instructions can access the same memory location is a critical task in binary analysis. It is challenging as statically computing precise alias information is undecidable in theory. The problem aggravates at the binary level due to the presence of compiler optimizations and the absence of symbols and types. Existing approaches either produce significant spurious dependencies due to conservative analysis or scale poorly to complex binaries.

We present a new machine-learning-based approach to predict memory dependencies by exploiting the model’s learned knowledge about how binary programs execute. Our approach features (i) a self-supervised procedure that pretrains a neural net to reason over binary code and its dynamic value flows through memory addresses, followed by (ii) supervised finetuning to infer the memory dependencies statically. To facilitate efficient learning, we develop dedicated neural architectures to encode the heterogeneous inputs (i.e., code, data values, and memory addresses from traces) with specific modules and fuse them with a composition learning strategy.

We implement our approach in NeuDep and evaluate it on 41 popular software projects compiled by 2 compilers, 4 optimizations, and 4 obfuscation passes. We demonstrate that NeuDep is more precise (1.5×) and faster (3.5×) than the current state-of-the-art. Extensive probing studies on security-critical reverse engineering tasks suggest that NeuDep understands memory access patterns, learns function signatures, and is able to match indirect calls. All these tasks either assist or benefit from inferring memory dependencies. Notably, NeuDep also outperforms the current state-of-the-art on these tasks.

**CCS CONCEPTS**

- **Security and privacy → Software reverse engineering:** • Computing methodologies → Machine learning.

**KEYWORDS**

Memory Dependence Analysis, Reverse Engineering, Large Language Models, Machine Learning for Program Analysis

**ACM Reference Format:**

Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: Neural Binary Memory Dependence Analysis. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’22), November 14–18, 2022, Singapore, Singapore. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549147

1 INTRODUCTION

Binary memory dependence analysis, which determines whether two machine instructions in an executable can access the same memory location, is critical for many security-sensitive tasks, including detecting vulnerabilities [18, 36, 86], analyzing malware [38, 93], hardening binaries [4, 29, 44, 90], and forensics [19, 35, 58, 91]. The key challenge behind memory dependence analysis is that machine instructions often leverage indirect addressing or indirect control-flow transfer (i.e., involving dynamically computed targets) to access the memory. Furthermore, most commercial software is stripped of source-level information such as variables, arguments, types, data structures, etc. Without this information, the problem of memory dependence analysis becomes even harder, forcing the analysis to reason about values flowing through generic registers and memory addresses. Consider the following code snippet where we show two instructions (within the same function) at different traces.

```c
void f(int a, int b) {
    int c = a + b;
    int d = a - b;
    printf("%d %d %d\n", c, d, d + 1);
}
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.  
ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore  
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-9413-0/22/11...$15.00  
https://doi.org/10.1145/3540250.3549147
The two instructions are memory-dependent (read-after-write) when `rax=0x3` (Trace 1). When analyzing the code statically, it requires precise value flow analysis to determine what values can flow to `rax` from different program contexts.

Over the last two decades, researchers have made numerous attempts to improve the accuracy and performance of binary memory dependence analysis [5, 6, 11, 16, 22, 34, 71]. The most common approach often involves statically computing and propagating an over-approximated set of values that each register and memory address can contain at each program point using abstract interpretation. For example, a seminal paper by Balakrishnan and Reps on value set analysis (VSA) [5] adopts strided intervals as the abstract domain and propagates the interval bounds for the operands (e.g., registers and memory locations) along each instruction. VSA detects two instructions to be dependent if their intervals intersect. Unfortunately, these static approaches have been shown to be highly imprecise in practice [96]. Composing abstract domains along multiple instructions and merging them across a large number of paths quickly accumulate prohibitive amounts of over-approximation error. As a result, the computed set of accessed memory addresses by such approaches often ends up covering almost the entire memory space, leading to a large number of false positives (i.e., instructions with no dependencies are incorrectly detected as dependent).

With the advent of data-driven approaches to program analyses [3, 69, 89], state-of-the-art memory dependence analysis is increasingly using statistical or machine learning (ML) based methods to improve the analysis precision [35, 58, 87, 96], but they still suffer from serious limitations. For example, DeepVSA [35] trains a neural network on static code to classify the memory locations accessed by each instruction into a more coarse-grained abstract domain such as stack, heap, and global, and use the predicted memory region to instantiate the value set in VSA. However, such coarse-grained prediction results in high false positives as any two instructions accessing the same region (e.g., stack) will always be detected as dependent even when the instructions access two completely different addresses. To avoid the precision losses by the static approaches, BDA [96] uses a dynamic approach that leverages probabilistic analysis to sample program paths and performs per-path abstract interpretation. However, as real-world programs often have many paths, the cost of performing per-path abstract interpretation for even a smaller subset of paths adds prohibitive runtime overhead, e.g., taking more than 12 hours to finish analyzing a single program. It is perhaps not surprising—while a dynamic approach can be more accurate than static approaches, it can incur extremely high runtime overhead, especially while trying to achieve good code coverage.

To achieve higher accuracy in a reasonably faster time, we propose an ML-based hybrid approach. Our key strategy is to learn to reason about approximate memory dependencies from the execution behavior of generic binary code during training. We then apply the learned knowledge to static code during inference without any extra runtime overhead (see Figure 1). Such a hybrid approach, i.e., learning from both code and traces, has been shown promise in several Software Engineering applications, including clone detection, type inference, and program fixing and synthesis [60, 63, 64, 89]. However, none of these works can reason fine-grained value flows through different memory addresses as they do not explicitly model memory. To bridge this gap, we aim to model the memory addresses in the ML-based hybrid framework and try to make fine-grained predictions differentiating the memory contents of different data pointers. Modeling memory address is, however, challenging as it requires the model to (i) distinguish between different memory addresses, (ii) learn to reason about indirect address references and memory contents, and (iii) learn the compositional effects of multiple instructions that involve memory operations.

To this end, we propose a new learning framework comprising pretraining and finetuning steps inspired by masked language model (MLM) [25], as shown in Figure 1. Unlike traditional MLM, where the input is restricted to a single input modality (e.g., text), our model learns from multi-modal information: instructions (static code), traces (dynamic values), and memory addresses (code spatial layout). We deploy a novel fusion module to simultaneously capture the interactions of these modalities for predicting memory dependencies. During pretraining, we mask random tokens from these modalities. While predicting masked opcode teaches the model to synthesize the instruction, predicting masked values in traces and memory addresses forces the model to learn to interpret instructions and their effect on registers and memory contents. For instance, if we mask the value of `rax` in `mov [rax], rbx` in the above example and train the model to predict it, the model is forced to interpret the previous instructions in the context and reason about how they compute their trace values that flow into `rax`. We hypothesize that

| Address | Instruction | Trace 1 | Trace 2 |
|---------|-------------|---------|---------|
| ......  | `mov [rax], rbx` | `rax=0x3; rbx=0x1` | `rax=0x5; rbx=0x1` |
| 0x06:  | `mov rbx, [0x3]` | `rbx=0x1` | `rbx=0x1` |
| 0x0f:  | `mov rdi, [0x3]` | `rdi=0x1` | `rdi=0x1` |

*In Intel x64 syntax [77]: `mov [rax], rbx` means writing register `rbx` to the memory pointed by register `rax`. `[]` means dereference a memory address.
such pretraining helps the model gain a general understanding of the value flow behavior involving memory operations.

After pretraining, the model is finetuned to statically (without the trace values) reason about the value flows (based on its learned knowledge from pretraining) across memory along multiple paths and predict the memory-dependent instruction pairs. Both pretraining and finetuning steps are automated and data-driven without manually defining any propagation rules for value flows. As a result, we show that our model is faster and more precise than the state-of-the-art systems (§5).

We implement our approach in NeuDep by carefully designing a new neural architecture specialized for fine-grained modeling of pointers to distinguish between unique memory addresses (Challenge i). We develop a novel fusion module to facilitate efficient training on the multi-modal bi-directional masking task, which helps the model to understand memory address content and thus, indirect memory references (Challenge ii). Finally, to teach the compositional effects of instructions on memory values (Challenge iii), we leverage the principle of curriculum learning [8], i.e., expose short training examples in the initial learning phase, and gradually increase the sample difficulties as the training progresses.

We evaluate NeuDep on a wide range of popular software projects compiled with diverse optimizations and obfuscation passes. We demonstrate that NeuDep is more precise than state-of-the-art binary dependence analysis approaches, widely-used reverse engineering frameworks, and even a source-level pointer analysis tool that has access to much richer program properties. We also show that NeuDep generalizes to unseen binaries, optimizations, and obfuscations, and is drastically faster than existing approaches. We perform extensive ablation studies to justify our design choices over other alternatives studied in previous works [64, 66]. Moreover, NeuDep is surprisingly accurate at many additional security-critical reverse engineering tasks, which either support or benefit from inferring memory dependencies, such as predicting memory-access regions, function signatures, and indirect procedure calls – NeuDep also outperforms the state-of-the-arts on all these tasks.

We make the following contributions:

(1) We propose a new neural architecture that can jointly learn memory value flows from code and the corresponding traces for predicting binary memory dependencies.
(2) We implement our approach in NeuDep that contains a dedicated fusion module for learning encodings of memory addresses/traces values, and a composition learning strategy.
(3) Our experimental results demonstrate that NeuDep is (3.5×) faster and more accurate (1.5×) than the state-of-the-art.
(4) Our extensive ablation studies and analysis on downstream tasks suggest that our pretraining substantially improves the prediction performance and helps the model to learn value flow through different instructions.

2 OVERVIEW

2.1 Motivating Example

Figure 2 shows that two instructions $I_2$: `mov rdi, [rax+0x8]` and $I_4$: `mov [rbp+rbx], rdi` access the same memory location (and are thus memory-dependent) but via different addressing registers. To detect the dependency, the model needs to first understand that the behavior of `mov`: both line 1 ($I_1$) and line 4 ($I_4$) set `rax` and `rbx` to the same value. Then it needs to understand `xor` in line 6 sets `rbp` to 0, and `add` in line 6 performs addition and sets `rbp` to `0x8`. Finally, the model needs to compose these facts and concludes that `rax+0x8` is semantically equivalent to `rbp+rbx` in such a context, i.e., they both evaluate to 0x123c.

**Gap in Existing Solutions.** We find that when running the ML model trained only on static code for this task [35], it mispredicts that $I_2$ and $I_4$ are not dependent as their memory-access regions ($I_2$ accesses heap while $I_4$ is mispredicted to access stack) do not intersect, possibly because its inference depends on the spurious pattern that the stack base pointer `rbp` is used at line 7. Such mispredictions [35] might lead to a false negative by flagging two instructions as accessing non-overlapping memory regions.

**Proposed Solution.** The above observation underscores the importance of encoding the knowledge about each instruction’s contribution to value flows through memory and their compositions as part of the ML model. However, integrating a memory model as part of the encoded knowledge is challenging due to the presence of potentially complex flows involving indirect address references and their compositions. We address these challenges by designing

(1) A novel training objectives to distinguish between unique memory addresses (§2.2)
(2) A dedicated fusion module specialized to capture the interaction between instruction, trace, and memory addresses (§2.3.2). Our new tracing and sampling strategies (§2.3.1) help the ML model to learn value flows across memory addresses.
(3) Curriculum learning [8] in the training process to incrementally learn the compositional effects (§2.3.3).

Table 1 shows some examples of how the pretraining task works and how it teaches the model to reason about value flows.
Table 1: Examples of masking (in grey) instructions and traces (represented as input (In) and output (Out) of each instruction). The model has to dereference the memory content and interpret or synthesize instruction(s) to infer the masked parts. We include the actual operations performed by the instructions (noted in green) and the formal semantics that the model essentially needs to learn for each example (last column).

| Example Descriptions                                                                 | Instruction(s) Mnemonic Operand | Trace In Out | Underlying Semantics |
|--------------------------------------------------------------------------------------|---------------------------------|--------------|-----------------------|
| Example 1: Interpreting memory operands in bitwise operations                        | rax=xor@[rbx]                   | 0x2 0x5 0x7   | \(v_1 = 0x4, v_2 = 0x7\) |
| Let the output value of rax in xor rax,[rbx] be masked. To predict the masked value, (i.e., rax=0x5), the model needs to understand the semantics of xor on its inputs rax=0x2 and [rbx]=0x7. | xor rax [rbx]                   | 0x7 0x7 0x7   | \(v = v_1 \land v_2\) |
| Example 2: Synthesizing Arithmetic Operations with memory operands                   | rbp=rbp@[rdi]                   | 0x4 0xc      | \(v_1 = 0x4, v_2 = 0x8\) |
| Let add be masked in add rbp,[rdi]. To predict the masked add (e.g., out of sub, mov, etc.), the model needs to associate add to the behavior that increments its its first operand by that of its second operand. | add rbp [rdi]                   | 0x8 0x8 0x8  | \(v = add\) |
| Example 3: Reverse Interpreting Arithmetic Operations with memory operands           | rcx=rax+[@rdx]                  | 0x9 0x8 0x1  | \(v_1 = 0x1, v_2 = 0x8\) |
| Let the input value of rcx in sub rcx,[rdx] be masked. To predict the masked value, the model needs to interpret sub backward given its output 0x8 and the value of its second operand 0x1 stored on memory. | sub rcx [rdx]                   | 0x1 0x1 0x1  | \(v_1 = 0x8\) |

2.2 Problem Formulation

Let \(f\) denote an ML model parameterized by \(\theta\). Before directly training \(f\) towards predicting memory dependencies, we pretrain \(f\) to reason about the value flows (see §3.4 for details). Now consider \(f\) with pretrained parameters \(\theta\), we formalize the task of analyzing memory dependencies as follows.

Definition 2.1 (Memory Dependency Prediction). Given a pair of assembly code instructions \(\{i_t, i_f\}\) within a code block \(l\) consisting of \(n\) assembly instructions: \(\{i_1, \ldots, i_n\}\), our neural memory dependency predictor \(f\), parameterized by the pretrained weight \(\theta\), predicts whether the instruction pair can access the same memory location, i.e., \(y = f(\{i_t, i_f\}, \theta), y \in \{0, 1\}\), \(y = 0\) denotes \(\{i_t, i_f\}\) do not have memory dependency, and \(y = 1\) denotes dependent.

Any \(y\) between 0 and 1 denotes the probability of \(i_t\) and \(i_f\) being dependent. Figure 2 shows an example how our model predicts different instruction pairs. We elaborate on how our neural architecture implements \(f\) in the above definition in §3.5.

2.3 NEUDep’s Design

Training the model to learn value flows for memory dependency analysis opens up several interesting design spaces, ranging from tracing to introduce diverse behaviors to designing appropriate inductive biases in the model architecture and training strategies. We overview our design in the following and provide detailed descriptions in §3.

2.3.1 Trace Collection. Pretraining requires high-quality training data to expose diverse program execution. We implement a forced execution engine [30, 67] to execute individual functions with full path coverage without the reliance on program test cases. Our execution engine differs from the existing works in two key aspects.

First, we note that existing works [28, 66, 67] implement the forced execution by violating the control flow semantics, i.e., stepping through control transfer instructions, to obtain traces with high coverage. However, this introduces noisy traces as they are not realizable in practice. On the contrary, respecting control transfers [64] will inevitably suffer from the coverage problem, as they have to find test cases to cover different paths [12, 20, 59]. To circumvent this problem, we implement a coverage-guided semantic-preserving branch-flipping mechanism to expose diverse paths within the function without breaking the branching instructions’ semantics (§3.1).

Second, existing works do not trace behaviors of the external procedure calls [64, 66], but this is especially important to model memory operations as heap allocation is often performed via library calls (e.g., via malloc). Our tracing engine provides complete environment support by pre-loading the whole program and its dependent libraries. The side effect of all function call instructions can thus be traced and logged as their input-output behavior (§3.1).

2.3.2 Representing and Fusing Code, Trace, and Memory. Assembly instructions and their traces are highly heterogeneous, i.e., instruction consists of discrete tokens like mnemonics and operands, while trace consists of mostly continuous values. To better encode the nature of each sequence, we employ two distinct modules to learn on these two inputs and then fuse them to make the joint inference. Specifically, we learn the instruction sequence with self-attention layers [83] to encode the instructions grounded on their neighboring context. We learn the trace values by a per-byte convolution network. After learning a basic representation of the code and trace values, we employ a fusion module (§3.3) to augment the contextualized instruction embeddings with the trace value embeddings.

We represent the code address space during execution as an additional input aligned to the instructions. This helps the model stay...
aware of the instructions’ order to their execution effect. Moreover, we observe that rip-relative addressing is frequently used to access global variables in position-independent code. Therefore, feeding addresses can help the model to learn the semantics of memory addressing. For example, consider the following instructions from the function `quotearg_free` in `runcon` from Coreutils-8.30.

| Address | Instruction                                      | Notes                       |
|---------|--------------------------------------------------|-----------------------------|
| 0x449c  | `cmp [rip+0x4d65], 2` # rip=0x4d65=0x9208         |                             |
| 0x44bc  | `movsx rax,[rip+0x4d45]` # rip=0x4d45=0x9208       |                             |
| 0x450e  | `mov [rip+0x4cf0], 1` # rip=0x4cf0=0x9208         |                             |

Three instructions use `rip` with different offsets to access the same global variable stored at 0x9208. By encoding the address of each instruction, we help the model infer the value of `rip` and thus assist reasoning on the memory dependencies.

### 2.3.3 Training Design (Composition Learning)

Inspired by how humans learn, we aim to develop a strategy that trains the model to gradually build up its knowledge. Ideally, the model should start by learning easy samples and then generalize its learned knowledge by getting exposed to more challenging training samples. As demonstrated in Table 1, the training samples with more instructions are more challenging to predict than those with fewer instructions, as the model has to learn the compositional execution effect of multiple instructions. Moreover, the more masks applied, the less context the model can leverage to make the prediction, thus increasing difficulty. Therefore, we develop a curriculum learning strategy [8] by sorting the training samples based on their length and increasing the masking rate at each training epoch. As a result, the model always starts learning from short code pieces with fewer masks at the early batches within each epoch, and the length of the code piece and the number of masks applied are increased in later epochs.

### 2.4 Additional Reverse Engineering Tasks

To explore how exactly pretraining helps analyze memory dependencies, we investigate what knowledge or properties of programs the pretrained model learns. We resort to `probing`, which uses the encoded instruction representations of the pretrained model and finetunes them on the probing tasks, usually with a small number of labeled data and training epochs [54]. Specifically, we consider three critical reverse engineering tasks, which either assist or benefit from analyzing memory dependencies. If the pretrained model performs well on these reverse engineering (i.e., probing) tasks, it gives evidence that pretraining has encoded useful representation for analyzing memory dependencies.

#### Inferring Memory Regions

Inferring memory-access regions helps reduce the spurious dependencies reported by VSA (§1). We consider the task sketched in DeepVSA [35], where the model needs to statically predict the memory region accessed by each instruction that operates on memory.

#### Definition 2.2 (Memory Region Prediction)

Given a code block consisting of a sequence of n assembly instructions: \( I = (I_1, ..., I_n) \), a memory region predictor \( f_y \), parameterized by the pretrained weights \( \theta \), predicts the memory region accessed by each instruction: \( y = f_y(I; \theta), y \in \mathbb{M}^n, \mathbb{M} = \{\text{stack, heap, global, other}\} \).

#### Inferring Function Signature

Traditionally, function signatures are predicted by analyzing the memory access patterns of variables and propagating the types implied by the inferred patterns up to the function argument. Memory dependencies help the propagating types along the dependent instructions [49]. The inferred variable types, in turn, also help reduce the spurious bogus dependencies, i.e., two memory accesses with different types are not dependent. We consider the task described in EKLAVYA [15], where the model statically predicts the function signature, including the (i) argument types, (ii) argument types, and (iii) function return types.

#### Definition 2.3 (Function Signature Prediction)

Given an n-instruction procedure \( P: P = (I_1, ..., I_n) \), a function signature predictor \( f_y \), parameterized by the pretrained weights \( \theta \), predicts the function signature as follows. (i) When \( P \) is treated as callee, \( f_y \) predicts \( P \)'s signature: \( y = f_y(P; \theta) \). (ii) When \( P \) is treated as caller, \( f_y \) takes call site \( I_c \in P \) as an additional input, and predicts the signature of the procedure that \( I_c \) calls: \( y = f_y(P, I_c; \theta) \). In both cases, \( y = (a, A, r) \) is a tuple where (i) \( a \in [0, 7] \) denotes argument arity with at most 7 arguments. (ii) \( A = (A_1, A_2, A_3) \) denotes \( P \)'s first 3 argument types: \( A_i \in \{\text{int, char, float, ptr, enum, union, struct}\} \). (iii) \( r \in \{\text{int, char, float, ptr, enum, union, struct, void}\} \) is the procedure \( P \)'s return type.

#### Matching Indirect Calls

Analysis of memory dependencies has been extensively applied to infer indirect calls [47, 96]. Therefore, we study how the pretrained model performs on this task.

#### Definition 2.4 (Matching Indirect Calls)

Given a pair of procedures \( P_i, P_j \), an indirect call predictor \( f_y \) predicts whether \( P_i \) can call \( P_j \) during runtime: \( y = f_y(P_i, P_j) \), where \( y \in \{0, 1\} \). If \( y = 1 \) denotes \( P_i \) can call \( P_j \) while \( y = 0 \) denotes \( P_i \) cannot.

Unlike the first two tasks, we define \( f_y \) as deterministic function that takes as input the inferred function signatures (Definition 2.3) of \( P_j \) and the call-site within \( P_i \). \( f_y \) outputs 1 if and only if the signature of \( P_j \) closely matches at least one call-site signature within \( P_i \). We elaborate on the matching criteria in §3.6.

### 3 METHODOLOGY

This section elaborates on the design of NeuDep, including the tracing framework, the model’s input representation, the neural architecture, and the training tasks.

#### 3.1 Tracing Framework

Algorithm 1 shows how our tracing framework works on a procedure. We consider the following two key designs (§2.3).

##### Environment Support

As shown in Algorithm 1 line 1 and 2, we first load the entire binary into an emulator and make a snapshot of the process image after initializing all dependent libraries. We then iterate every function inside the binary and execute each function (line 4 and 5). Before the execution, we restore the process memory using the saved snapshot (line 6) to ensure that all the functions, including external library functions, are properly resolved.

##### Branch Flipping

Inspired by coverage-guided fuzzing, we design a dynamic branch flipping mechanism for recording complete and diverse execution behaviors. We first maintain a list of covered basic blocks during past execution (line 8-15). We then hook every conditional branch during forced execution and monitor the jump target. If the jump target has already been covered before and another is not covered yet, we flip the branch. In order to ensure
Algorithm 1 Coverage-Guided Semantic-Preserving Execution

1: Load(binary) ▶ Load binary into emulator
2: mem = Snapshot() ▶ Save memory snippet after initialization
3: covered_bb = {} ▶ Loop every function
4: for func ∈ binary do ▶ Loop every conditional branch
5: Restore(mem) ▶ Restore memory snapshot
6: Initialize(stack, regs) ▶ Initialize stack and registers with random values
7: /* Loop every conditional branch */
8: for cond_branch ∈ ForceExec(func) do ▶ /* bb1, bb2 are jump targets. bb1 is the default one */
9: if bb1 ∈ covered_bb and bb2 ∈ covered_bb then
10: FlipBranch(cond_branch)
11: covered_bb.add(bb2)
12: else
13: covered_bb.add(bb1)
14: /* we show the byte value before normalization to save space. As T_{10} corresponds to rdi, which is not executed, its value is 0, which is 0x100 before normalizing.

3.2 Input Representation
At a high level, NeuDep takes three sequences as input, i.e., assembly instructions, trace values, and instruction addresses.

Assembly. We represent the assembly instructions I = (I_1, ..., I_n) as n ordered tuples. Each tuple I_i consists of 3 members: I_i = (c_i, p_i, m_i), where c_i, p_i, m_i indicate code position, target, and whether c_i accesses memory, respectively. Specifically, c_i denotes the tokens obtained from tokenizing the assembly instructions, removing punctuation, and transforming all constants to const. As we flatten each instruction into multiple tokens, we use p_i to annotate the relative position of c_i within the instruction to specify the instruction boundary. Moreover, p_i helps the self-attention layers, which are permutation-invariant to the input tokens, to understand the relative order of the operands. Finally, m_i ∈ {T, F} denotes whether c_i accesses memory.

Example 3.1. Consider the instruction sequence add rax,0x8; mov [rax], rbx. It will be represented as:

\[
\begin{array}{c|cccccccc}
\text{c} & L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & L_7 & L_8 \\
\text{p} & F & F & F & F & F & F & T & F \\
\text{m} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

Trace. We represent the trace values using T = (T_1, ..., T_n) aligned to the assembly instruction sequence I. Each T_i ∈ T consists of a list T_i = (b_1^i, ..., b_8^i), where the numeric value is a (padded) 8-byte values b_j^i ∈ [0, 1]. This reduces a prohibitively large vocabulary (2^{64}) to a much more manageable size (2^{56}) [64]. The most and least significant byte is b_1 and b_8, respectively. We further normalize each byte b_j^i ∈ [0, 1] to stabilize the training. For instruction tuples I_i whose c_i is not a register or a constant, its aligned trace values T_i contains 8 dummy values (.), which are not assigned pretraining (§3.4). For T_i whose aligned I_i is not executed, we assign the value b_j^i = 0x100, ∀j ∈ [1, 8]. In pretraining, to predict the trace value consisting of all 100s instead of regular bytes, the model needs to determine whether the corresponding assembly instructions are executed by reasoning the branch predicate and control flow.

Example 3.2. Consider the following 4 instructions: add rax,0x8; cmp rax,0x18; je 0x1004ab5f; push rdi; input rax=0x0. T would look like (aligned with c_i):

\[
\begin{array}{cccccccc}
& c & push & cmp & rax & const & je & const & push & rdi \\
T & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & T_8 & T_9 \\
0 & 0 & 0 & 0 & - & - & - & - & - & - \\
1 & 0 & 0 & 0 & - & - & - & - & - & - \\
2 & 0 & 0 & 0 & - & - & - & - & - & - \\
3 & 0 & 0 & 0 & - & - & - & - & - & - \\
4 & 0 & 0 & 0 & - & - & - & - & - & - \\
5 & 0 & 0 & 0 & - & - & - & - & - & - \\
6 & 0 & 0 & 0 & - & - & - & - & - & - \\
7 & 0 & 0 & 0 & - & - & - & - & - & - \\
\end{array}
\]

As the machine instruction of push rdi only takes one byte (0x55), the addresses of two instructions are off by one byte.

3.3 NeuDep Architecture
Figure 3 illustrates NeuDep’s architecture. In the following, we describe how the inputs (§3.2) are embedded, fused, and further processed to make the prediction. All these steps are handled by neural modules that can be stacked together and trained end-to-end.

Input Embeddings. Let d denote the embedding dimension, we denote the embeddings of each tuple (c_i, p_i, m_i) as E(c_i), E(p_i), E(m_i) ∈ IR^d. We sum these embeddings to form the embeddings of I_i: E(I_i) = E(c_i) + E(p_i) + E(m_i). We denote the embedding of all tokens as E^9(I) = (E(I_1), ..., E(I_n)), representing the instructions’ embeddings before the first self-attention layer. We first apply I self-attention layers on E^9(I) to learn the contextual embeddings of the assembly instructions: E^d(I).

To embed the 8-byte values in T and A into a space that preserves their numerical properties, we employ a convolution network with 8 kernels to learn how bytes within each neighboring size interact with each other. Let C_w denote applying a convolution filter with width w and output channel C_w. We first apply 8 convolution filters on T_i = (b_1, ..., b_8) and concatenate them: C_out = concat(ϕ(max(C_w(T_i)))) as the activation function, and we use ReLU in this paper.
We give the formal definition of our pretraining task as follows. 

\[ f = \text{pretrain}(\theta) \]

3.4 Pretraining: Interpret and Synthesize Code

We give the formal definition of our pretraining task as follows.

**Definition 3.1 (Pretraining).** Given (i) a code block \( I = (I_1, \ldots, I_n) \), (ii) its trace \( T: T = (T_1, \ldots, T_m) \), and (iii) a mask rate \( r \), we pretrain the model \( f \), parameterized by \( \theta \), by the following training objectives.

1. **Interpret \( I \):** predict the masked trace \( T_{MT}: T_{MT} \subseteq T, |T_{MT}| = |T| \cdot r \), given \( I \) and \( T - T_{MT} \): \( T_{MT} = f(I, T - T_{MT}; \theta) \).

2. **Synthesize \( I \):** predict the masked instructions \( I_{MI}: I_{MI} \subseteq I, |I_{MI}| = |I| \cdot r \), given \( I - I_{MI} \) and \( T: I_{MI} = f(I - I_{MI}, T; \theta) \).

3. **Both:** predict both \( I_{MI} \) and \( T_{MT} \) given \( I - I_{MI} \) and \( T - T_{MT} \): \( I_{MI}, T_{MT} = f(I - I_{MI}, T - T_{MT}; \theta) \).

Specifically, the pretraining takes as input the output of the last self-attention layer \( E_l^I = (E_1^I, \ldots, E_p^I) \), and minimize the \((1)\) cross-entropy (CE) between the predicted masked code \( \hat{e} \) and the actual code \( e_{MI} \), and the \((2)\) mean squared error (MSE) between predicted masked values (8 bytes) \( T_{MT} \) and the actual values \( T_{MT} \):

\[
\arg \min_{\theta} \sum_{i \in MI} -c_i \log \hat{e_i} + \alpha \sum_{j \in MT} (\hat{T}_j - T_j)^2
\]

\( \theta \) denote the trainable parameters \textsc{Neude}'s model \((3.3)\) and the prediction heads: \( MLP_p \) and \( MLP_T \), two multilayer perceptrons that take \( E_l^I \) as input and predict the masked instructions: \( e_{MI} = MLP_p(E_{MI}^I) \) and trace values: \( T_{MT} = MLP_T(E_{MT}^I) \).

**Composition Learning.** We increase the masking percentage \( r \) \((3.3)\) at each epoch. Let \( L, U \) denote the lower and upper bound of \( r \), respectively, and \( EPO_{pretrain} \) denote the pretraining epochs, at \( k \)-th epoch, \( r = L + (U - L) \cdot (k - 1) / EPO_{pretrain} \).

3.5 Finetuning: Predict Memory Dependencies

As shown in Figure 1, after the model is pretrained, we let the model predict value flows between instructions based on its learned representation of assembly code without traces. To this end, we detach the fusion module and the convolution module for embedding the trace \( T \) and addresses \( A \) (right part in Figure 3) and directly stack the upper \( L - l \) self-attentions on top of the first \( l \) self-attention layers.

Given \( E_l^I = (E_1^I, \ldots, E_p^I) \) \((3.3)\), we employ a prediction head \( MLP_{dep} \) that minimizes the binary cross-entropy (BCE) between the predicted dependency \( \hat{y} \) of \( (I_i, I_j) \subseteq I \) and their ground truth \( y \) \((Definition 2.1)\): arg \( \min_{\theta} -y \cdot \log \hat{y} - (1 - y) \cdot \log(1 - \hat{y}) \), where

\[
\hat{y} = MLP_{dep}(\text{concat}(\hat{y}_a(E_l^I), E_1^I, \ldots, E_p^I))
\]

Here \( \hat{y}_a \) denotes taking the mean pooling of \( E_l^I \) and \( \odot \) denotes element-wise multiplication. This results in the input shape to \( MLP_{dep} \) to be \( R^{5d} \).

3.6 Downstream Reverse Engineering Tasks

As described in \( \S 2.4 \), we consider three security-critical reverse engineering tasks as our probing tasks. We follow the similar setup in \( \S 3.5 \) and stack separate prediction heads on top of \( E_l^I \), and train with additional training samples collected for probing.

**Infer**ing **Memory** **Regions.** Given the output of the last self-attention layer \( E_l^I = (E_1^I, \ldots, E_p^I) \), we stack a prediction head \( MLP_r \) that predicts the memory-access regions for each instruction \( I_i \). The training task then minimizes the sum of cross-entropy between the predicted memory regions \( \hat{y} \) of each instruction and their ground truth memory region \( y \): arg \( \min_{\theta} \sum_{i=1}^n -y_i \cdot \log(MLP_r(E_{MI}^I)) \).

**Infer**ing **Function** **Signature.** As shown in Definition 2.3, predicting function signatures consists of predicting 5 types of labels: \( \{a_1, a_2, a_3, a_4, r\} \). For each label, we create two prediction heads: \( MLP_{callr} \) and \( MLP_{callc} \). For example, \( MLP_{callr} \) takes as input the embedding corresponding to the call site \( c \in [1, n] \) from the last
self-attention layer $E^L$, and predicts the number of arguments that the call site prepares: $a = \text{MLP}^\text{call} \left( E^C_i \right)$. $\text{MLP}^\text{call}$ takes as input the embeddings from the last self-attention layer $E^L$ and predicts the number of arguments the callee expects: $a = \text{MLP}^\text{call} \left( \bar{y}_a \left( E^L \right) \right)$ where $\bar{y}_a$ denotes the average pooling of all embeddings in $E^L$. The training objective for each head then minimizes the cross-entropy loss between the predicted label and the ground truth label.

**Matching Indirect Calls.** Given the signatures of a call site $P_i$ and a callee $P_j$, we implement the indirect call predictor $f_c$ (Definition 2.4) by considering the following 4 criteria. (i) *Loose arity:* $P_i$ must prepare at least as many arguments as $P_j$ accepts. (ii) *Strict arity:* the arities of $P_i, P_j$ must match exactly. (iii) *Argument type:* the types of $P_i$’s first three arguments must match $P_j$’s argument types in at least 2 or 3 positions. (iv) *Return type:* if $P_j$ is non-void, then $P_j$ must be non-void. The four criteria can be composed to determine whether $P_i, P_j$ matches. We evaluate the 8 compositions in §5.3.

## 4 IMPLEMENTATION AND SETUP

We implement NeuDep’s tracing framework in Qiling [81] and the model architecture based on PyTorch. We run all experiments and baselines on a Linux server, with Intel Xeon 4214 at 2.20GHz with 48 virtual cores, 188GB RAM, and 4 Nvidia RTX 2080Ti GPUs.

**Dataset.** We collect 41 open-source projects, ranging from utility libraries like Binutils to popular libraries like OpenSSL (see Appendix). We compiled these projects with 4 optimizations, i.e., O0-O3, using GCC-9.3.0, and 4 obfuscations based on Clang-8 [94], i.e., bogus control flow (bfc), control flow flattening (cfl), basic block splitting (spl), and instruction substitution (sub). Among the 41 projects, we select 9 projects as our finetuning set and the rest for pretraining. They include bash-5.0, bc-1.07.1, binutils-2.30, bison-3.3.2, cflow-1.6, coreutils-8.30, curl-7.76, findutils-4.7.0, gawk-5.1.0. The 9 projects have disparate functionalities and sizes such that they are diverse and representative of real-world software. We perform static disassembly (taking less than 0.1 seconds per input) followed by a simply post-processing to parse the raw assembly into the format that the model accepts (§3.2).

**Ground Truth Dependencies.** We follow [96] by using dynamic analysis to collect the ground truth memory dependencies. To quantify how NeuDep and the baselines perform, we measure the detected dependencies among the reference ones (detect) and mark the rest as miss; we treat the predicted dependencies not included in the references as potential false positives (FP).

**Baselines.** We compare NeuDep to Angr [85], Ghidra [1], SVF [78], and DeepVSA [35]. As SVF does support dumping its result to the compiled binary (confirmed with the authors) [78], we propagate its result using the DWARF information. As one source statement can map to multiple assembly instructions, we treat it as a true positive if its detected dependencies include the ground truth instruction pair. We thus omit evaluating SVF on the obfuscated binaries as the obfuscator significantly distorts the mapping in Dwarf. For DeepVSA, its VSA implementation requires taking a crash dump as input and does not work for general memory dependence analysis. Therefore, we run its released trained model and use its predicted memory region to determine whether two memory-access instructions are dependent. We note that PalmTree [51] also compared to DeepVSA on the standalone memory region prediction task without running its VSA module. However, as PalmTree does not release its trained model for memory region prediction, we cannot run PalmTree on our dataset to predict memory dependencies. Therefore, we instead compare NeuDep to PalmTree and its evaluated baselines (including DeepVSA) in our probing studies (5.3).

BDA [96] is the state-of-the-art binary memory dependence analysis tool, to the best of our knowledge. We reached out to the authors and confirmed that BDA targets reducing false negatives in the inter-procedural setting, and they evaluated it on only O0 binaries. Per our requests, BDA authors performed preliminary studies and observed BDA achieves low miss rate (0.02%), but suffers from high false positive rate and runtime overhead. For example, on readdir compiled by O0, BDA has around 2.23% precision (detecting 5,742 true dependencies out of a total of 256,596 predicted dependencies). Due to the different focuses between BDA and NeuDep, we thus omit including BDA results to avoid unfair comparison.

For probing tasks, we compare NeuDep to (i) PalmTree [51] and the other baselines that PalmTree evaluated such as DeepVSA [35], Asm2Vec [27], and Instruction2Vec [50], on predicting memory access regions, (ii) EKLAVYA [15] on predicting function signatures, and (iii) EKLAVYA and TypeArmor [82] on predicting indirect calls.

**Hyperparameters.** For composition learning, we set $L = 0.2$ and $U = 0.8$ (§3.4). For pretraining (Equation 1), we set $\alpha = 100$ by observing that MSE loss is around 100× smaller than that of CE in our experiments.

## 5 EVALUATION

We focus on three main research questions in the evaluation.

- **RQ1:** How well does NeuDep perform in analyzing memory dependencies? (§5.1)
- **RQ2:** How much does each design choice in NeuDep contribute to its performance? (§5.2)
- **RQ3:** How well does NeuDep perform in downstream reverse engineering tasks? (§5.3)

### 5.1 NeuDep Performance

Table 2 presents the results of NeuDep and other baselines on the test set categorized by their optimization and obfuscation flags. NeuDep’s results are obtained from finetuning a single model on all datasets, excluding the testing set. On average, NeuDep detects 1.5× more dependencies than the second-best (DeepVSA), while having 4.5× fewer misses than the second-best (SVF). Ghidra has fewer false positives than NeuDep, but at the cost of missing substantial dependencies, detecting 3.3× fewer dependencies than NeuDep. Besides, we note that DeepVSA produces 6.4× more false positives on higher optimizations when compared to O0. This is likely because memory access patterns (e.g., using rbp and rax to access stack and heap, respectively, are largely broken by compilers).

**Zero-Shot Generalizability to Unseen Projects.** In the above experiment, our training and testing set are randomly sampled with non-overlapping pairs, but they could come from the same software project. Therefore, we investigate how NeuDep performs when its testing set comes from entirely different software projects. We collect 2 software projects featuring a web server, i.e., Nginx-1.21.1,
Table 2: **NeuDep** and other baselines’ results on the test set categorized by the compiler optimizations and obfuscations.

| Flags | # Dep | Angr Detect | Miss | FP | Ghidra Detect | Miss | FP | SVF Detect | Miss | FP | DeepVSA* Detect | Miss | FP | NeuDep Detect | Miss | FP |
|-------|-------|-------------|------|----|---------------|------|----|------------|------|----|----------------|------|----|---------------|------|----|
| O0    | 1,013 | 28          | 985  | 16 | 355           | 658  | 7  | 380        | 633  | 392| 420           | 593  | 329| 930           | 83   | 147|
| O1    | 1,310 | 12          | 1,298 | 14 | 387           | 923  | 19 | 486        | 824  | 1,474| 617           | 693  | 2,013| 898           | 412  | 576|
| O2    | 1,103 | 14          | 1,089 | 10 | 330           | 773  | 6  | 403        | 698  | 1,392| 464           | 639  | 1,512| 822           | 281  | 480|
| O3    | 1,132 | 14          | 1,118 | 14 | 332           | 800  | 14 | 425        | 707  | 1,351| 472           | 660  | 1,527| 827           | 305  | 501|
| bcf   | 3,144 | 23          | 3,121 | 14 | 160           | 2,984 | 0  | -          | -    | -  | 613           | 2,531| 445| 3,122         | 22   | 127|
| cff   | 758   | 22          | 736  | 9  | 208           | 550  | 0  | -          | -    | -  | 337           | 421  | 113| 724           | 34   | 56 |
| spl   | 1,296 | 24          | 1,272 | 18 | 173           | 1,123 | 2  | -          | -    | -  | 515           | 781  | 181| 1,245         | 51   | 77 |
| sub   | 938   | 22          | 916  | 14 | 264           | 674  | 7  | -          | -    | -  | 393           | 545  | 236| 885           | 53   | 127|
| Avg.  | 1,337 | 20          | 1,317 | 14 | 276           | 716  | 7  | 424        | 716  | 1,152| 478           | 858  | 795| 1,182         | 155  | 261|

*DeepVSA’s VSA implementation takes crash dumps as input and does not work on our dataset (regular binary code without crashes). Therefore, we run DeepVSA’s released model on our dataset and use its predicted memory region to flag dependent instructions (§4).

Table 3: **NeuDep** performance when dividing its dataset by non-overlapping train-test vs. non-overlapping programs, optimizations, and obfuscations. Δ denotes the performance change scaled by the number of reference dependencies.

| Cross- | # Dep | Regular Detect | FP | Unseen Detect | FP | Δ (+/-) Detect | FP |
|--------|-------|----------------|----|---------------|----|----------------|----|
| Proj.  | Nginx | 226            | 164 | 155          | 68 | -4%            | +5.3% |
|        | Lynx  | 322            | 240 | 92           | 244 | +12%           | +8.7% |
| Opt.   | O0    | 1,013          | 959 | 90           | 958 | -0.1%          | +1.3% |
|        | O1    | 1,310          | 1,026 | 112        | 988 | -2.9%          | +14.2% |
|        | O2    | 1,103          | 919 | 195         | 908 | -1%            | +1.2% |
|        | O3    | 1,132          | 922 | 207         | 933 | +1%            | +2.5% |
| Obf.   | bcf   | 3,144          | 3,131 | 104       | 2,946 | -5.9%          | +36.1% |
|        | cff   | 758            | 746 | 38          | 746 | +0.3%          | -0.4% |
|        | spl   | 2,217          | 2,171 | 121       | 2,076 | -4.3%          | -0.9% |
|        | sub   | 938            | 907 | 76          | 914 | +0.8%          | +1.4% |
| Avg.   | 1,216 | 1,119          | 109 | 1,087       | 218 | -2.6%          | +9% |

and a web browser, *i.e.*, Lynx-2.8.9, to which none of the projects in our dataset has similar functionality. We compile each software project with 4 optimizations (O0–O3), and test NeuDep on these unseen software projects. As a baseline, we fine-tune a model where its training set includes the project, but with non-overlapping instruction pairs ("Regular" in Table 3).

The first two rows in Table 3 demonstrate that NeuDep remains relatively robust when the testing set is collected from unseen projects, *i.e.*, on average, the number of detected dependencies only drops 1.4% and false positives increased by 7%. Interestingly, training without samples from Lynx even increases the detected dependencies, but at the expense of much higher false positives.

**Zero-Shot Generalizability to Unseen Optimizations/Obfuscations.** Aggressive compiler transformations can bring many challenges to inferring memory dependencies, *e.g.*, substituting instructions introduces more pointless arithmetic operations, which requires reasoning over the bloated instructions to detect the value flows. To study whether NeuDep generalizes to unseen optimizations and obfuscations, we exclude binaries optimized or obfuscated by each strategy (§4) in training and test NeuDep on the excluded binaries.

Table 3 presents results when testing NeuDep on each unseen optimizations and obfuscations. We also include the baseline results when its training set includes those optimized or obfuscated binaries (but with non-overlapping pairs). We observe that NeuDep generalizes to unseen optimizations and obfuscations, with only 2.6% drop in detection rate and 9% increase in false positives.

**Runtime Performance.** One of the most significant benefits of NeuDep over traditional approaches comes from its speed, as its analysis is amenable to parallelization with GPUs. Table 4 compares the speed of NeuDep to Angr and Ghidra. We run each tool on each project compiled with O0 from our finetuning dataset (§4). We observe that Angr often takes long time and cannot finish running (as also confirmed by [96]). Thus, we time it out after 5 minutes. Consequently, Angr’s actual runtime is under-estimated. We do not compare to (i) DeepVSA because it still relies on VSA, so it is at least as slow as any VSA implementation, and (ii) SVF because it works only on LLVM IR, not directly on binaries. Therefore, SVF has extremely high overhead from mapping LLVM IR results to binary. Table 4 shows that NeuDep is 3.5× faster than the second-best tool (Ghidra) and orders of magnitude faster (125.2×) than Angr.

5.2 Ablation Study

We study how much each design in NeuDep (§3) contributes to its performance. We follow the setup in §5.1. Table 5 summarizes the results where we bold NeuDep’s default choice.

**Pretraining.** We first ablate the effectiveness of pretraining in assisting memory dependence analysis. Table 5 shows that pretraining NeuDep significantly improves its performance by 12.6% in the number of detected dependencies. The number of misses and false positives drop by 57.6% and 31.4%, respectively.

Table 4: Runtime of NeuDep vs. Angr and Ghidra. The last column shows the speedup over the second-best tool.

| Size (MB) | Angr Inference Time (s) | Ghidra Inference Time (s) | NeuDep Inference Time (s) | Speedup |
|----------|-------------------------|--------------------------|---------------------------|---------|
| bash     | 2.8                      | 7685.9                   | 60.4                      | 24.4    | 2.5×     |
| bc       | 0.5                      | 2988.5                   | 183.1                     | 20.8    | 1.4      |
| binutils | 74                       | 70157.1                  | 3077.2                    | 935.6   | 4.4×     |
| bison    | 1.6                      | 1730.1                   | 90.2                      | 18.4    | 2.9×     |
| cflow    | 0.56                     | 695.9                    | 6.5                       | 112.5   | 2.2×     |
| coreutils| 16                       | 40,188.2                 | 392.2                     | 105.9   | 3.7×     |
| curl     | 0.77                     | 91.5                     | 14.5                      | 6.3     | 7.0×     |
| findutils| 2.3                      | 882.7                    | 80.1                      | 23.2    | 3.5×     |
| gawk     | 3.8                      | 2905.3                   | 55.0                      | 52.1    | 5.6×     |
| Avg.     | 12.8                     | 13781.7                  | 413.5                     | 311.4   | 3.5×     |
Table 5: Ablation on NeuDep designs. We treat the first row of each design as the baseline and compute the improvement of other alternatives.

| Ablation Setup | Detect | Miss FP | Improve (+/-) | Detect | Miss FP |
|----------------|--------|---------|---------------|--------|---------|
| Pretrain w/o  | 8,780  | 1,914   | 1,477 0.0%  | 0.0%   | 0.0%    |
| w             | 9,882  | 812     | 1,013 +12.6% | -57.6% | -31.4%  |
| Value | 9,666  | 1,208   | 1,027 0.0%   | 0.0%   | 0.0%    |
| Embed Conv    | 9,882  | 812     | 1,013 +2.2%  | -32.8% | -1.4%   |
| Sum           | 9,752  | 942     | 1,419 0.0%   | 0.0%   | 0.0%    |
| Fusing 1st Layer | 9,882 | 812    | 1,013 +1.3% | -13.3% | -28.6%  |
| 3rd Layer     | 9,870  | 824     | 1,209 +1.2%  | -12.5% | -14.7%  |
| 5th Layer     | 9,700  | 994     | 1,186 +0.5%  | -5.5%  | -16.4%  |
| Compos. Learning w/o Compos. | 9,806 | 888    | 1,389 0.0%   | 0.0%   | 0.0%    |
| w | 9,882  | 812     | 1,013 +0.8%  | -8.6%  | -27.1%  |
| Code w/o Addr | 9,667  | 1,027   | 1,407 0.0%   | 0.0%   | 0.0%    |
| Add | 9,882  | 812     | 1,013 +2.2%  | -20.9% | -28%    |

Byte Aggregation. We study the effectiveness of encoding numeric values using convolutions with highway network (§3.2) by comparing it to the baseline that concatenates the input bytes. Table 5 shows that our encoding mechanism outperforms the baseline by 2.2% and significantly reduces the miss detection rate by 32.8%.

Input Fusion. We explore the effectiveness of input fusion by comparing it to the baseline that takes the vector sum of the input embeddings [64, 66]. We also study fusion after which layer is the most effective. We note that simply summing the embeddings of code and trace values at input by assuming they are homogeneous performs the worst. This confirms our intuition that code and trace are heterogeneous data that benefit from different encoding mechanisms. In addition, we note that combining code and trace at earlier layers performs the best. This is likely because trace values can participate early in the model’s computation of interactions between instructions and trace values, i.e., fusing in the later layers implies it has fewer remaining layers to learn how code and trace interacts.

Composition Learning. We study whether composition learning (§2.3) would help the model’s finetuning performance for detecting memory dependencies. We compare it to the fixed masking percentage strategy where the samples are shuffled randomly, and the masking rate is fixed to 0.5 on both the code and trace tokens. Table 5 shows that composition learning moderately improves the model by 0.8% in detected dependencies but substantially reduces the number of false positives by 27.1%. This observation confirms our intuition that arranging the training samples based on their difficulty helps the model learn more efficiently.

Modeling Address Layout. We study whether annotating the binary code with its loaded addresses would bring a useful inductive bias to the model by comparing to the baselines that do not model them [64, 66]. Table 5 shows that annotating the code with addresses significantly reduces the model’s missed detection and false positives, i.e., by 20.0% and 28%, respectively. This shows that the code address helps the model reduce the spurious dependencies.

5.3 Performance on Reverse Engineering Tasks

We probe pretrained NeuDep using three reverse engineering tasks that either assist or benefit from memory dependence analysis.

Table 6: Comparison of F1 scores on memory region prediction between NeuDep and PalmTree and other baselines.

| Ret | Caller | Callee |
|-----|-------|-------|
|     | O0    | O1    | O2    | O3    | O0    | O1    | O2    | O3    | Avg.  |
|     |       |       |       |       |       |       |       |       |       |
|     | EKLAVYA | DeepVSA | PalmTree | O0 O1 O2 O3 | O0 O1 O2 O3 | O0 O1 O2 O3 | O0 O1 O2 O3 | O0 O1 O2 O3 | O0 O1 O2 O3 |
| EKLAVYA | 68.62 | 70.59 | 73.63 | 76.19 | 91.59 | 88.87 | 91.92 | 95.32 | 96.06 |
| NeuDep | 94.65 | 93.33 | 95.75 | 96.41 | 95.37 | 93.42 | 96.06 | 98.20 |       |
| EKLAVYA | 91.56 | 90.38 | 91.21 | 91.55 | 95.62 | 92.40 | 93.05 | 92.56 |       |
| NeuDep | 97.01 | 96.97 | 98.47 | 99.17 | 97.24 | 95.10 | 97.81 | 97.84 |       |
| EKLAVYA | 81.82 | 78.70 | 81.81 | 82.03 | 87.25 | 82.67 | 82.40 | 85.07 |       |
| NeuDep | 96.08 | 94.86 | 97.68 | 97.51 | 93.30 | 91.34 | 94.66 | 92.45 |       |
| EKLAVYA | 80.28 | 79.85 | 81.35 | 76.63 | 77.42 | 69.18 | 70.93 | 69.80 |       |
| NeuDep | 96.88 | 96.89 | 97.09 | 97.24 | 96.55 | 94.42 | 94.66 | 95.68 |       |
| EKLAVYA | 92.03 | 86.02 | 83.80 | 82.79 | 97.48 | 76.24 | 77.49 | 78.69 |       |
| NeuDep | 98.84 | 95.44 | 96.35 | 95.86 | 99.23 | 92.57 | 95.04 | 96.40 |       |

Memory-Access Regions. We follow PalmTree by running NeuDep on the DeepVSA’s dataset and compare NeuDep to the reported F1 scores of PalmTree, DeepVSA, and other baselines (§4). We note that DeepVSA’s datasets are all 32-bit x86 binaries, but NeuDep is pretrained on x86-64 binaries. However, we find that just our vocabulary constructed from x86-64 binaries covers 89.9% of DeepVSA’s dataset vocabulary, likely because both belong to the x86 family. Therefore, we simply apply our vocabulary on DeepVSA’s dataset and replace unseen tokens with "unknown" in the vocabulary.

Table 6 shows that NeuDep remains robust across different memory regions. On average, NeuDep outperforms PalmTree by 0.069. On more challenging labels such as heap, NeuDep outperforms PalmTree and DeepVSA by 0.19 and 0.32, respectively. This is likely because accessing these memory regions involves more diverse patterns, e.g., via the stack pointer register (Figure 2).

Function Signature. We compare NeuDep to EKLAVYA on recovering function signatures. Table 7 shows that NeuDep outperforms EKLAVYA on all signature inference tasks, achieving 12.6% higher accuracy on average. Most notably, NeuDep’s performance remains robust across different tasks and optimization levels, while EKLAVYA's accuracy shows clear drops. For instance, when comparing the prediction accuracy of 3rd argument (A3) and 1st argument (A1), EKLAVYA decreases by 16.61% while NeuDep’s drops by only 1.19%. Likewise, within the arity task, EKLAVYA’s accuracy decreases 14.02% from O0 to O3, while NeuDep decreases 2.91%.

Indirect Calls. Finally, we compare how well NeuDep, EKLAVYA, and TypeArmor detect indirect calls (Definition 2.4). We consider 8 matching algorithms (§3.6) grouped row-wise by arity matching criteria detailed in Table 8. On all algorithms, NeuDep outperforms EKLAVYA and TypeArmor, achieving 0.032 and 0.07 higher F1 scores, respectively. With loose arity and return type matching –
the criterion adopted in TypeArmor – NeuDep outperforms TypeArmor by 0.052 in F1 score. We also note that NeuDep’s performance increases as the matching algorithm incorporates more conditions, while the performance of other systems remains roughly the same.

6 THREATS TO VALIDITY

Architecture Bias. We only consider x86-64 binaries. While we have shown NeuDep generalizes to several x86-32 binaries (§5.3), it cannot directly be applied to binaries with significantly different syntax, e.g., those running on ARM or MIPS architectures. However, as our trace engine supports other architectures well [81], we can potentially pretrain the model for other architectures. We also plan to extend our models to different programming languages that come with efficient tracing support [31, 57, 68].

Performance Bias. We only compare NeuDep’s runtime performance on GPUs with other baselines (§5.1), as NeuDep’s neural module runs on GPU by default. However, we believe that significantly benefitting from GPU is indeed a key advantage of ML-based techniques over traditional binary analysis that cannot easily exploit GPU parallelism and thus struggle to scale to large binaries.

Ground Truth Bias. Obtaining complete ground truth for memory dependencies in real-world programs is intractable. Therefore, following BDA’s approach [96], we resort to dynamic analysis and use the accessed memory locations observed during execution to collect the reference dependencies. While we cannot guarantee the ground truth to be complete, this approach can still quantify how many dependencies are missed by the evaluated tools. Table 2 shows that NeuDep outperforms all baselines with the fewest misses.

Inter-Procedural Analysis. We only capture full execution behavior starting from a callee. Therefore, NeuDep primarily expects instruction pairs to come from the same function. However, as we trace the full execution behavior of method calls, our model potentially learns to reason about the value flows across procedures. We plan to explore NeuDep’s capability in inter-procedural analysis by modeling the complete calling context in our future study.

7 RELATED WORK

Binary Memory Dependence Analysis. There has been a long history of efforts to approach the problem of analyzing memory dependencies in executables [5, 6, 11, 16, 22, 34, 35, 71, 96]. Debray et al. [22] and Cifuentes et al. [16] pioneered this field by using abstract interpretation to propagate the abstract domain along the registers of each instruction. VSA [5] improves on their idea by supporting tracking value flows along both the registers and memory locations. DeepVSA [35] further improves on VSA by learning a neural network to predict the memory-access regions of each instruction to pre-filter those not sharing the regions. BDA [96] uses probabilistic analysis to uniformly sample paths and performs per-path abstract interpretation to avoid precision losses from path merging. While both DeepVSA and BDA sacrifice soundness, they have been shown to significantly assist in debugging crashes [19, 58] and malware analysis. However, they still incur high runtime overhead and produce many false positives for optimized binaries – NeuDep substantially outperforms these tools (§5).

8 CONCLUSION

We present a new ML-based approach, NeuDep, to predict memory dependencies. We first pretrain NeuDep to understand how instructions propagate dynamic values across memory and registers, then finetune the model to detect memory dependencies statically. We demonstrate that NeuDep is precise and efficient, outperforming the state-of-the-art in both detection accuracy (1.5×) and speed (3.5×). Extensive probing studies demonstrate that NeuDep understands memory access patterns, learns function signatures, and can match indirect calls – these tasks either assist or benefit from inferring memory dependencies. Notably, NeuDep also outperforms the state-of-the-art on these tasks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive and valuable feedback. We thank the author of BDA, Zhuo Zhang, for providing valuable insight and suggestion, and running experiments of BDA. This work is sponsored in part by NSF grants CCF-1845893, CCF-2107405, CNS-1564055, and IIS-2221943; ONR grant N00014-17-1-2788; an NSF Career Award; an Accenture Faculty Research Award; a Google Gift; an IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, NSF, ONR, Accenture, Google, or IBM.
REFERENCES

[1] National Security Agency. 2019. Ghidra Disassembler. https://ghidra-sre.org/.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 281–293.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. Comput. Surveys (2018).

[4] Paul-Antoine Arras, Anastasios Andronidis, Luis Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumburk, and Cristian Cadar. 2022. SABRe: load-time selective binary rewriting. International Journal on Software Tools for Technology Transfer (2022), 1–19.

[5] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In International conference on compiler construction.

[6] Gogul Balakrishnan and Thomas Reps. 2010. WISNIWX: What you see is not what you execute. ACM Transactions on Programming Languages and Systems (TOPLAS) 32, 6 (2010), 1–94.

[7] Sébastien Bardin, Tristan Benoît, and Jean-Yves Marion. 2021. Compiler and optimization level recognition using graph neural networks. In MLPLA 2020: Machine Learning for Program Analysis.

[8] Joshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In Proceedings of the 26th annual international conference on machine learning. 41–48.

[9] Tristan Benoît, Jean-Yves Marion, and Sébastien Bardin. 2021. Binary level toolchain provenance identification with graph neural networks. In 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering.

[10] Avishkar Bhoslechand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307 (2016).

[11] David Brunelley and James Newcombe. 2006. Alias analysis for assembly. Technical Report. Technical Report CMU-CS-06-180, Carnegie Mellon University.

[12] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On introducing automatic test case generation in practice: A success story and lessons learned. Journal of Systems and Software 176 (2021), 110933.

[13] Ngîq Dq Bui, Tijun Yu, and Lingxiao Jiang. 2022. Codec: Self-supervised learning of code representations by predicting substrings. In 26th ACM SIGSAC Conference on Research and Development in Information Retrieval. 511–521.

[14] Zheng Leong Chu, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In 26th USENIX Security Symposium.

[15] Cristina Cifuentes and Antoine Fraboulet. 1997. Intraprocedural static slicing of binary executables. In International Conference on Software Maintenance.

[16] Matteo Cinellii, Nathan Cooper, Luca Fascella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of BERT models for code completion. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) IEEE, 108–119.

[17] Marco Cova, Viktoria Flemetsov, Greg Banks, and Guvanna Vigna. 2006. Static detection of vulnerabilities in x86 executables. In 2006 22nd Annual Conference Applications Security Conference (ACSAC’06). IEEE, 269–278.

[18] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures in deployed software. In Operating Systems Design and Implementation.

[19] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/USE: A dynamic execution tool for binary-level analysis. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering.

[20] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–28.

[21] Saumya Debraj, Robert Muth, and Matthew Weippeart. 1998. Alias analysis of executable code. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 12–24.

[22] Renzo Degiovanni and Mike Papadakis. 2022. pBERT: Mutation Testing using Pre-Trained Language Models. arXiv preprint arXiv:2203.03289 (2022).

[23] Prem Devanbu, Matthew Dwyer, Sebastian Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Bashikhal Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep learning & software engineering: State of research and future directions. In Proceedings of the 31st ACM SIGSOFT International Symposium on Foundations of Software Engineering. 1–28.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: 3881–3893.

[25] Elizabeth Dinella, Todd Mykyto, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu K Lahiri. 2021. Deepmerge: Learning to merge programs. arXiv preprint arXiv:2105.08659 (2021).

[26] Stevens HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Armzvec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In 2019 IEEE Symposium on Security and Privacy (S&P). IEEE, 472–487.

[27] Manuel Egle, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In 23rd USENIX Security Symposium (USENIX Security 14). 303–317.

[28] Ulfar Erlingsson, Martin Abadi, Michael Vabale, Mihai Budiu, and George C Necula. 2006. XIF: Software guards for system address spaces. In Proceedings of the 7th symposium on Operating systems design and implementation. 75–88.

[29] Patrice Godefroid. 2014. Micro execution. In 36th International Conference on Software Engineering.

[30] Matthias Gremmel, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. 2017. Truffle: Dynamic execution of c on a java virtual machine. In Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools.

[31] Wenchao Gu, Zhiyuan Shao, and Haijun Lin. 2017. Firmub: Vetting usb device firmware using domain informed symbolic execution. In Computer and Communications Security.

[32] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. Commun. ACM 59, 5 (2016), 122–131.

[33] Yuan Huang, Xinyu Hu, Nan Jia, Xiangpeng Chen, Yingfei Xiong, and Zihong Zheng. 2019. Learning code context information to predict comment locations. IEEE Transactions on Reliability 68, 1 (2019), 88–105.

[34] Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anninda Ibgl, and Toufiq Ahmed. 2022. ReviewRepair: Code review aided automatic program repairing. Information and Software Technology 143 (2022), 106765.

[35] Malibeh Isfah, Roberta Gismondi, and Georgios Geoussos. 2022. CodeFill: Multi-token code completion with neural search. In Proceedings of the 30th ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[36] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmus: Vetting usb device firmware using domain informed symbolic execution. In Computer and Communications Security.

[37] K. Pei, D. She, M. Wang, S. Geng, Z. Xuan, Y. David, J. Yang, S. Jana, B. Ray Park. 2017. Learning binary code with deep learning to detect software weakness.
