Scalable, Accurate, Robust Binary Analysis with Transfer Learning

Suman Jana, Columbia University

Joint work with Kexin Pei, Jonas Guan, Zhou Xuan, David Williams-King, Dongdong She, Michael Wang, Scott Geng, Yaniv David, Baishakhi Ray, and Junfeng Yang
(Binary) Program Analysis Consists of Many Fundamental Tasks

Has Many Security-and Performance-Critical Applications!
**Binary Analysis** is a Process of

Understanding, Analyzing, and Answering Questions regarding **Program Behavior**

Can Function `foo()` Call Function `bar()`?

Disassembly

```
push rbp;mov rbp, rsp
```

Functions

```
Function Start/End
```

Data Types

```
Int64_t *
```

Control/Data Flow

```
Can i or buf Overflow?
```

```
int i;
char buf[10];
```

• `foo()`
• `bar()`

x86 -O0

ARM -O3

Requires Recovering **Program Properties and Dependencies**
Binary Analysis is Fundamentally Challenging

High-Level Semantics are Absent due to Compilation, Stripping, etc.

Source-Level Constructs

<table>
<thead>
<tr>
<th>Variables</th>
<th>Arguments</th>
<th>Types</th>
<th>Data Structures</th>
<th>Names</th>
<th>...</th>
</tr>
</thead>
</table>

| Registers | Memory Accesses |

Further Complicated by Factors that Result in Diverse Syntactic Differences

Existing solutions are heavily heuristic-driven, slow, and brittle
Machine Learning is Promising

**Cheap:** Data-driven Approach Saves Manual Effort

**Efficient:** Neural Net Architecture with GPU

Learned Knowledge is not Retained:
- Single-task training Might Learn Spurious Features
- Wasting Repetitive Effort
We Explore a Transfer Learning Paradigm

**Pretrain** to Learn General Program Behavior

Exploit the Interdependency between Tasks:
- Learn Key Features: Program Behavior
- Generalizable: Avoid Learning Task-specific Spurious Features
- Adaptive: Updated Knowledge from One Task can Easily Transfer to Other Tasks
Three Different Approaches to Binary Analysis with Transfer Learning

- XDA: Accurate, Robust Disassembly with Transfer Learning (NDSS 2021)
- StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling (FSE 2021)
- NeuDep: Neural Binary Memory Dependence Analysis (FSE 2022)
XDA: Transfer Learning Disassembly

1. Recovering Functions (Boundaries)

```
cc cc 40 55 57 41 56 41 57 48 83 ec 28 ... 41 5f 41 5e 5f 5d c3 90 90
```

Function Start

Function End

2. Recovering Assembly Instruction (Boundaries)

```
cc cc 40 55 57 41 56 41 57 48 83 ec 28 ... 41 5f 41 5e 5f 5d c3 90 90
```
XDA: Transfer Learning Disassembly

1. Recovering Functions (Boundaries)

Function Start

cc cc 40 55 57 41 56 41 57 48 83 ec 28 ... 41 5f 41 5e 5f 5d c3 90 90

Function End

2. Recovering Assembly Instruction (Boundaries)

push rbp  push r14  push rdi
sub rsp,0x28  pop r15  pop rdi  ret
push rdi  push r15  pop r14  pop rbp
Many Existing Works and Tools

Reverse Engineering Tools

Non-ML:
Zhang and Sekar, 2013 (Bin-CFI), Wang et al., 2015 (UROBOROS), Khadra et al., 2016 (Spedi), Andriesse et al., 2017 (Nucleus), Qiao and Sekar, 2017 (Function Interface), Wang et al., 2017 (Ramblr), Bauman et al., 2018 (Superset Disassembly), Alves-Foss and Song, 2019 (Jima), Flores-Montoya and Schulte (Ddisasm), etc.

ML:
Kruegel et al., 2004 (CFG and Statistical Analysis), Rosenblum et al., 2008 (CRF), Bao et al., 2014 (ByteWeight), Wartell et al., 2014 (Probabilistic FSM), Shin et al., 2015 (bidirectional-RNN), Wang et al., 2017 (FID), Guo et al., 2018 (LEMNA), He et al., 2018 (Debin), Miller et al., 2019 (Probabilistic Disassembly), etc.

Research Prototypes
Limitations of Existing Tools

1. **Inaccurate:** distinguishing between **Code** and **Data**

   - **Objdump** Treats as Code
   - **Ghidra** Treats as Data

   - *False Positive*

   - *False Negative*

2. **Brittle and Not Robust:**

   - *Function Prologue Pattern*

   - *Increase Optimization O1 to O2*

   - Neither **IDA** nor **bi-RNN** recognized the optimized function

3. **Expensive Manual Effort:** Updating and Maintaining Heuristics

   - **IDA** ~41 MB Pattern Database
   - **Ghidra** ~179 MB Pattern Database

---

**Vim 8.2 compiled by MSVC 2019**
Key Insight

Based on 2-step Transfer Learning Paradigm (BERT - Devlin et al.)

1. **Pretrain** to Understand Machine Code Semantics from Raw Bytes

2. **Transfer** Learned Knowledge for Disassembly

**Masked Language Modeling** (Self-supervised)

**Finetuning** for Recovering Functions and Assembly Instructions

```
cc cc 40 55 57 41 56 41 57 48 83 ec ... 41 5f 41 5e 5f 5d c3 90 90
```
Quick Summary of XDA

1. **Accurate:** 99 and 99.7 F1 in Recovering Functions and Assembly Instructions. Outperforms State-of-the-Art by 17.2 on Average

2. **Robust:** Remains Accurate Across Compilers, Operating Systems, Architectures, Optimizations, Obfuscations, and Train-test Overlap.

3. **Automated Semi-supervised Learning:** Robust Features Useful for Disassembly

**Code and Dataset** Available at: [https://github.com/CUMLSec/XDA](https://github.com/CUMLSec/XDA)
The Key Idea: Learning Code Representation

1. **Pre-Training:** Learn Generic Code Representation In a Task Agnostic Manner

   Large set of programs/binaries collected
   - Structure
   - Semantics

   Neural Net
   - Learn
   - Network Weights
     - Code Representation

2. **Fine-Tuning:** Transfer Learned Representation for Downstream Binary Analysis Tasks

   Static Binary
   - Matching Code Similarity
   - Type inference
   - Alias analysis
The Key Idea: Learning Code Representation

Benefits of Learned Representation

- **Generalizable**: Avoid learning task-specific spurious features
- **Adaptive**: Updated knowledge from one task can easily transfer to other tasks
- **Cheap**: Self-supervised training saves manual effort collecting ground truths
Limitations of XDA pretraining: no direct teaching of semantics

.text:
cc cc 40 55 57 41 56 41 57 48 83 ec 28 b8 8b e9 45 8b f0 48 8b fa 4c 8b f9 48 83 c4 28 41 5f 41 5e 5f 5d c3 90 90

XDA Pretraining:
(1) Mask Random Bytes, and
(2) Train the Model to Predict the Masked Part based on the Context

Masking/predicting bytes based on nearby bytes does not directly teach code semantics
How to make learned code representation semantics-aware

- The goal of pre-training is to learn program representations that can help in downstream tasks
- A semantics-aware code representation should preserve as much program semantics (input/output behavior) as possible
- Learning to separate code with similar input/output behaviors from other code statically can be very hard

```plaintext
mov eax, 0
add eax, 1
xor eax, eax
sub eax, -1
```

Without Understanding `mov, xor, add, sub, etc.`
ML Algorithms often Pick
**Spurious Correlation**

Same Behavior
Even More Challenging Examples:

Looking at Static Code it is hard to learn the Inherent Same Semantics (Input/Output Behavior)
Key Idea: **Learning semantics-aware code representations with Trace Modeling**

1. Learn semantics-aware representation from multiple input modality

   ![Diagram](image)

   - **Large set of binaries**
     - **Instructions**
     - **Trace Values**

   **Learn interactions between instructions and trace Values**

   **Neural Net**
   - Learn
   - Program Representation
   - **Execution semantics**

   **Advantage:**
   - **Input/output behavior**: Program execution semantics to support reasoning program behavior
   - **Generalizable**: Avoid learning task-specific spurious features
   - **Adaptive**: Updated knowledge from one task can easily transfer to other tasks
   - **Cheap**: Self-supervised training saves manual effort collecting ground truths

2. **Transfer and fine-tune** program representation for Downstream Program Analysis Tasks

   ![Diagram](image)

   **Static code**
   - **Binary similarity**
   - **Type inference**
   - **Alias analysis**

   **Advantage:**
   - **Complete**: Covers effects of all instructions
   - **Lightweight**: No Execution Overhead
   - **Efficient**: Neural net architecture with GPU
Micro-traces for Modeling

- **Problem**: Generating complete traces of large programs with high coverage is prohibitively expensive
- **Solution**: Collect micro traces by force executing partial code piece (e.g., a function $f$)
  - Skip Jumps/Calls with Unreachable Target
  - Ignore invalid memory accesses
  - Exposes more code behaviors, e.g., skipping potential input check exceptions
  - Large-scale pretraining can learn diverse execution behavior from such noisy micro-traces
  - Pretraining aims to learn local behavior of instructions, less sensitive to violations introduced by under-constrained traces
**Multiple Input Modalities**

Learn How Traces are Generated by Instructions

Pretrained Model

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>sub</th>
<th>ecx</th>
<th>num</th>
<th>add</th>
<th>ecx</th>
<th>num</th>
<th>jmp</th>
<th>ecx</th>
<th>push</th>
<th>ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace-Value Sequence</td>
<td>##</td>
<td>2</td>
<td>1</td>
<td>##</td>
<td>1</td>
<td>3</td>
<td>##</td>
<td>4</td>
<td>##</td>
<td>##</td>
</tr>
<tr>
<td>Memory Addresses</td>
<td>0x3</td>
<td>0x3</td>
<td>0x3</td>
<td>0x6</td>
<td>0x6</td>
<td>0x6</td>
<td>0x9</td>
<td>0x9</td>
<td>0xc</td>
<td>0xc</td>
</tr>
<tr>
<td>Control Flow Coverage</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Opcode/Operand Position</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Architecture Annotation</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
<td>x86</td>
</tr>
</tbody>
</table>

Four Modalities:
1. Static Code
2. Execution Trace
3. Memory Layout
4. Control Flows

Auxiliary Annotations

Micro-Traces

![Diagram of Micro-Traces and Pretrained Model](https://via.placeholder.com/150)
Neural Net Architecture and Pretraining Task

Pretraining Task Intuition:

**Forward Reasoning:**
\[
\text{Deduction: } 2 - 1 = ?
\]

**Backward Reasoning:**
\[
\text{Abduction: } ? - 1 = 1
\]

**Inverse Reasoning:**
\[
\text{Induction: } 1 ? 3 = 4
\]

\[
\text{Deduction, Abduction, and Induction: Irreducible Set of Reasoning Primitives (Peirce, 1992)}
\]

Predicting **Multiple** Masked Tokens Forces the Model to **Understand** Diverse Instructions’ Semantics and **Reason About** their Compositions
Improving Binary analysis with Trace Modeling

**Accurate:** Outperforms the State-Of-The-Art

- Matching Semantically Similar Code: +14.3%
- Type Inference: +12.6%
- Memory Dependence (Alias) Analysis: +52.6%

**Efficient:** Outperforms the State-Of-The-Art

- Matching Semantically Similar Code: 8x speedup
- Type Inference: 98.1x speedup
- Memory Dependence (Alias) Analysis: 3.5x speedup

**Robust:** Remains Accurate Across

- Compilers
- Architectures
- Optimizations
- Obfuscations
Trace Modeling for binary analysis

- StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling (FSE 2021)
- NeuDep: Neural Binary Memory Dependence Analysis (FSE 2022)
Stateformer: Binary Type Inference with trace modeling
Trace Collection using Micro-Execution

- Collected by Executing Partial Code Piece $f$ (a Function)
- Initialize All Registers with Random Values Except the Special Purpose Registers
- When Encountering Memory Access, Map the Memory On-Demand
  - Initialize the Memory with Random Values if it is Read from Memory
- Log Dataflow and Control Flow States of Each Instruction in $f$
  - Align the Execution States (values of registers, memory) to Instructions

**Instructions**

```
mov ebp,esp
add [ebp+0x8],0x3
cmp [ebp+0x8],0x2
jle 0x6
sub [ebp+0x8],0x1
mov eax,0
```

**Trace Values**

```
mov 0x1c,0x4
add [0x4+0x8],0x3
cmp [0x4+0x8],0x2
jle 0x6
sub [0x4+0x8],0x1
mov 0x0,0
```
Trace Modeling in StateFormer

Learn How Traces are Generated by Instructions

Pretrained Model

Code Semantics

Two Modalities:
1. Static
2. Dynamic Behavior

Auxiliary Annotations

Instruction Sequence

Instruction Position

Opcode/Operand Position

Architecture Annotation

Instructions

Micro-Traces

Learn

 ecx=2

 ecx=1

 ecx=4

 ecx=1

 ecx=3

 ecx=4

 ecx=2

 sub ecx,1

 add ecx,3

 j ecx
Trace Modeling in StateFormer

Transfer the Learned Knowledge to Predict Types

No Traces at inference time
Experimental Setup

- **Dataset**
  - 33 Popular Open-Source Software Projects:
    - Binutils, ImageMagick, OpenSSL, PuTTY-0.74, SQLite-3.34.0, etc.
  - Compiled to 4 Architectures
    - x86, x64, ARM, MIPS
  - Compiled by 2 Compilers
    - GCC, LLVM
  - Compiled by using 4 Compiler Optimizations
    - -00, -01, -02, -03
  - Compiled by 3 types of obfuscations by Hikari (https://github.com/HikariObfuscator/Hikari)
    - Bogus Control Flow, Control Flow Flattening, and Instruction Substitution

- **Baselines**
  - EKLAVYA (Chua et al.) - Function Signature Inference (Argument Type)
  - Debin (He et al.) - Variable Name and Type Recovery
  - TypeMiner (Maier et al.) - More Fine-Grained Variable Type Recovery

- **Metrics**
  - Precision, Recall, F1 Score
StateFormer Performance

77.9 F1 On Average
Comparison to SOTA

Outperforms EKLAVYA by 13.3%

Outperforms Debin by 14.6%
Trace Modeling for Binary Analysis

- StateFormer: Fine-Grained Type Recovery from Binaries using Generative State Modeling (FSE 2021)
- NeuDep: Neural Binary Memory Dependence Analysis (FSE 2022)
NeuDep: Binary Memory Dependence Analysis

Can two machine instructions access the same memory location during execution?

```
......   ......
x = a    mov [rax], rbx
......   ......
int y = x    mov rdi, [rcx]
......   ......
```

Memory
Challenges in Binary Memory Dependence Analysis

High-Level Semantics are Absent

- Variables, Types, Names, etc.
- Registers
- Memory Accesses

Further Complicated by

- Compilers
- -00 -01 -02 -03 -0d -0x
- Optimizations
- Obfuscations

Limitations of Existing Approaches

Static Analysis: Value Set Analysis
- Abstract interpretation based on strided interval
- **Imprecise** due to over-approximation when combining strided intervals
- **Not scalable** to real-world programs
- Angr-VSA and BAP-VSA times out after 12 hours on SPEC benchmarks (Zhang et al., OOPSLA, 2019)

Machine Learning Approaches: DeepVSA
- Spurious Pattern Matching
- Coarse-Granularity

**Instructions I**

- $I_1$ mov rax,0x1234
- $I_2$ mov rdi,[rax+0x8]
- $I_3$ mov rdx,[rip+0xbf]
- $I_4$ mov rbx,0x1234
- $I_5$ xor rbp,rbp
- $I_6$ add rbp,0x8
- $I_7$ mov [rbp+rbx],rdi
- $I_8$ mov [rip+0x81],rax

$I_2$: Heap $\times$ $I_7$: Stack

$I_3$: Global $\checkmark$ $I_8$: Global
NeuDep: Why Trace Modeling is Useful?

How do Human Analysts Infer Memory Dependence?

Reason about **Value Flows** by following instructions through memory addresses

What values can flow to \texttt{rax}?

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Instructions</th>
<th>Trace 1</th>
<th>Trace 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>\texttt{mov [rax],rbx}</td>
<td>\texttt{rax=0x5;rbx=0x1}</td>
<td>\texttt{rax=0x3;rbx=0x1}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1f</td>
<td>\texttt{mov rdi,[0x3]}</td>
<td>\texttt{rdi=0x0}</td>
<td>\texttt{rdi=0x1}</td>
</tr>
</tbody>
</table>

When \texttt{rax=0x5}, two instructions are not dependent

When \texttt{rax=0x3}, two instructions are dependent (read after write)
Learning to Reason About Value Flows using Trace modeling

Sample Value Flows from traces $I, T, A \sim (\text{Dynamic Analysis})$

- Self-Supervised Pre-Training
  - Instructions $I$
  - Traces $T$
  - Memory Layout $A$

Model Input

- Sample Masks

Model Output

- Synthesize: $P(I_{\text{masked}} \mid I-I_{\text{masked}}, T, A)$
- Interpret: $P(T_{\text{masked}} \mid T-T_{\text{masked}}, I, A)$

Supervised Fine-Tuning (Static)

- Sample Instruction Pairs $I_i, I_j$

- Memory Dependence Prediction: $P(\text{Dep}_{i,j} \mid I_i, I_j, I, A)$

Gain a general understanding of the value flow behavior involving memory operations
### Why Interpret and Synthesize Instructions?

Gain a general understanding of the value flow behavior involving memory operations.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Trace</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xor rax,[rbx]</strong></td>
<td>$\text{In}(\text{rax})=0x2, \text{In}([\text{rbx}])=0x7$ \hspace{1cm} $\text{Out}(\text{rax})=0x5, \text{Out}([\text{rbx}])=0x7$</td>
<td>$v_1 = 0x2, v_2 = 0x7$ \hspace{1cm} $v = v_1 \oplus v_2$ \hspace{1cm} $v = 0x5$</td>
</tr>
<tr>
<td><strong>push rdi</strong></td>
<td>$\text{In}(\text{rdi})=0x6, \text{In}(\text{rsp})=0x8$ \hspace{1cm} $\text{Out}(\text{rdi})=0x6, \text{Out}(\text{rsp})=0x0$</td>
<td>$\text{Out}(\text{rdi})=0x6, \text{Out}([\text{rsp}])=0x6$</td>
</tr>
</tbody>
</table>
Using Code Address Space as an Input Modality

Method: `quote arg_free` in Program: `runcon` in Coreutils-8.30

```
......
0x449c: cmp [rip+0x4d65],2  # rip+0x4d65=0x9208
......
0x44bc: movsxd rax,[rip+0x4d45]  # rip+0x4d45=0x9208
......
0x450e: mov [rip+0x4cf0],1  # rip+0x4cf0=0x9208
......
```

3 Instructions are memory dependent
Representing Different Input Modalities

Modality 1: Instructions

Example: `add rax, 0x8; mov [rax], rbx`

<table>
<thead>
<tr>
<th>Code Sequence $c$</th>
<th>Instruction Position $p$</th>
<th>Memory Access $m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(add)$</td>
<td>(1)</td>
<td>(F)</td>
</tr>
<tr>
<td>$(rax)$</td>
<td>(2)</td>
<td>(F)</td>
</tr>
<tr>
<td>$(const)$</td>
<td>(3)</td>
<td>(F)</td>
</tr>
<tr>
<td>$(mov)$</td>
<td>(1)</td>
<td>(F)</td>
</tr>
<tr>
<td>$(r ax)$</td>
<td>(2)</td>
<td>(T)</td>
</tr>
<tr>
<td>$(rb x)$</td>
<td>(3)</td>
<td>(F)</td>
</tr>
</tbody>
</table>
Representing Different Input Modalities

Modality 2: Traces

Example: \texttt{add rax,0x8;cmp rax,0x10;je 0x1004a8b5f;push rdi}
Let \texttt{rax=0x0}

<table>
<thead>
<tr>
<th>(c)</th>
<th>(T)</th>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_3)</th>
<th>(T_4)</th>
<th>(T_5)</th>
<th>(T_6)</th>
<th>(T_7)</th>
<th>(T_8)</th>
<th>(T_9)</th>
<th>(T_{10}^*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_1)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_2)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_3)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_4)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_5)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_6)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_7)</td>
<td>-</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>(00)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
<tr>
<td>(b_8)</td>
<td>-</td>
<td>(00)</td>
<td>(08)</td>
<td>(08)</td>
<td>(10)</td>
<td>(10)</td>
<td>(10)</td>
<td>(5f)</td>
<td>-</td>
<td>-</td>
<td>100*</td>
</tr>
</tbody>
</table>
Representing Different Input Modalities

Modality 3: Code Address Layout

Example: `push rbp; jmp rax`
Let code be loaded at address 0x14a8b

<table>
<thead>
<tr>
<th>c</th>
<th>push</th>
<th>rbp</th>
<th>jmp</th>
<th>rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
</tr>
<tr>
<td><code>{b₆}</code></td>
<td><code>{01}</code></td>
<td><code>{01}</code></td>
<td><code>{01}</code></td>
<td><code>{01}</code></td>
</tr>
<tr>
<td><code>{b₇}</code></td>
<td><code>{4a}</code></td>
<td><code>{4a}</code></td>
<td><code>{4a}</code></td>
<td><code>{4a}</code></td>
</tr>
<tr>
<td><code>{b₈}</code></td>
<td><code>{8b}</code></td>
<td><code>{8b}</code></td>
<td><code>{8c}</code></td>
<td><code>{8c}</code></td>
</tr>
</tbody>
</table>
NeuDep: Overall Architecture

Modality 1: Instructions
- Value-Flow Augmented Code Embedding
- Fusing
  - Self-Attention
  - Fuse $E^l(I)$, $E(T)$, $E(A)$
  - $E^l(I)$
  - $E^0(I)$
  - $I$
  - $E(c)$, $E(p)$, $E(m)$

Modality 2: Traces
- Fusing
  - Self-Attention
  - $E(T)$
  - Conv
  - $T$
  - Weight Sharing

Modality 3: Addresses
- Fusing
  - Self-Attention
  - $E(A)$
  - Conv
  - $A$
NeuDep Experimental Setup

● **41** Popular Open-Source Software Projects:
  ○ OpenSSL, Binutils, etc.
  ○ Compiled by using **4** Compiler Optimizations (-O0, -O1, -O2, -O3) by gcc-9.3
  ○ Compiled by **4** types of obfuscations by Hikari (clang-8)
    ■ Bogus Control Flow (bcf), Control Flow Flattening (cff), Basic Block Splitting (spl), and Instruction Substitution (sub)

● Baselines:
  ○ Angr’s VSA implementation, Ghidra, SVF, and DeepVSA
NeuDep: Downstream Task

- 4 Downstream Binary Analysis Tasks either assist or benefit from memory dependence analysis
  - Predicting Memory Access Regions (Stack, Heap, Global, Other)
  - Predicting Function Signature (Argument Count and Type)
  - Matching Indirect Calls
NeuDep Accuracy

<table>
<thead>
<tr>
<th>Flags</th>
<th># Dep</th>
<th>Detect</th>
<th>Angr</th>
<th>Ghidra</th>
<th>SVF</th>
<th>DeepVSA*</th>
<th>NEUDEP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>Detect</td>
<td>Miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>Detect</td>
<td>FP</td>
</tr>
<tr>
<td>O0</td>
<td>1,013</td>
<td>28</td>
<td>985</td>
<td>16</td>
<td>355</td>
<td>658</td>
<td>7</td>
</tr>
<tr>
<td>O1</td>
<td>1,310</td>
<td>12</td>
<td>1,298</td>
<td>14</td>
<td>387</td>
<td>923</td>
<td>19</td>
</tr>
<tr>
<td>O2</td>
<td>1,103</td>
<td>14</td>
<td>1,089</td>
<td>10</td>
<td>330</td>
<td>773</td>
<td>6</td>
</tr>
<tr>
<td>O3</td>
<td>1,132</td>
<td>14</td>
<td>1,118</td>
<td>14</td>
<td>332</td>
<td>800</td>
<td>14</td>
</tr>
<tr>
<td>bcf</td>
<td>3,144</td>
<td>23</td>
<td>3,121</td>
<td>14</td>
<td>160</td>
<td>2,984</td>
<td>0</td>
</tr>
<tr>
<td>cff</td>
<td>758</td>
<td>22</td>
<td>736</td>
<td>9</td>
<td>208</td>
<td>550</td>
<td>0</td>
</tr>
<tr>
<td>spl</td>
<td>1,296</td>
<td>24</td>
<td>1,272</td>
<td>18</td>
<td>173</td>
<td>1,123</td>
<td>2</td>
</tr>
<tr>
<td>sub</td>
<td>938</td>
<td>22</td>
<td>916</td>
<td>14</td>
<td>264</td>
<td>674</td>
<td>7</td>
</tr>
<tr>
<td>Avg.</td>
<td>1,337</td>
<td>20</td>
<td>1,317</td>
<td>14</td>
<td>276</td>
<td>1,061</td>
<td>7</td>
</tr>
</tbody>
</table>

NeuDep detects 1.5× more dependencies while having 4.5× fewer misses than the second-best tool
NeuDep is 3.5× faster than the second-best tool (Ghidra) and orders of magnitude faster (125.2×) than Angr.
## Ablation Study: Effect of different modalities

<table>
<thead>
<tr>
<th>Ablation Setup</th>
<th>Detect</th>
<th>Miss</th>
<th>FP</th>
<th>Improve (+/-)</th>
<th>Detect</th>
<th>Miss</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pretrain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/o</td>
<td>8,780</td>
<td>1,914</td>
<td>1,477</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>w/</td>
<td>9,882</td>
<td>812</td>
<td>1,013</td>
<td>+12.6%</td>
<td>-57.6%</td>
<td>-31.4%</td>
<td></td>
</tr>
<tr>
<td>Value Embed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concat</td>
<td>9,666</td>
<td>1,208</td>
<td>1,027</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>Conv.</td>
<td>9,882</td>
<td>812</td>
<td>1,013</td>
<td>+2.2%</td>
<td>-32.8%</td>
<td>-1.4%</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>9,752</td>
<td>942</td>
<td>1,419</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>Fusing Strategy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1st Layer</td>
<td>9,882</td>
<td>812</td>
<td>1,013</td>
<td>+1.3%</td>
<td>-13.8%</td>
<td>-28.6%</td>
<td></td>
</tr>
<tr>
<td>3rd Layer</td>
<td>9,870</td>
<td>824</td>
<td>1,209</td>
<td>+1.2%</td>
<td>-12.5%</td>
<td>-14.7%</td>
<td></td>
</tr>
<tr>
<td>5th Layer</td>
<td>9,700</td>
<td>994</td>
<td>1,186</td>
<td>+0.5%</td>
<td>-5.5%</td>
<td>-16.4%</td>
<td></td>
</tr>
<tr>
<td>Compos. Learning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/o Compos.</td>
<td>9,806</td>
<td>888</td>
<td>1,389</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>w/ Compos.</td>
<td>9,882</td>
<td>812</td>
<td>1,013</td>
<td>+0.8%</td>
<td>-8.6%</td>
<td>-27.1%</td>
<td></td>
</tr>
<tr>
<td>Code Addr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/o Addr.</td>
<td>9,667</td>
<td>1,027</td>
<td>1,407</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>w/ Addr.</td>
<td>9,882</td>
<td>812</td>
<td>1,013</td>
<td>+2.2%</td>
<td>-20.9%</td>
<td>-28%</td>
<td></td>
</tr>
</tbody>
</table>
## Downstream Task 1: Memory Region Prediction (F1 score)

<table>
<thead>
<tr>
<th></th>
<th>Global</th>
<th>Heap</th>
<th>Stack</th>
<th>Other</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction2Vec</td>
<td>0.654</td>
<td>0.566</td>
<td>0.914</td>
<td>0.947</td>
<td>0.77</td>
</tr>
<tr>
<td>Asm2Vec</td>
<td>0.517</td>
<td>0.359</td>
<td>0.911</td>
<td>0.948</td>
<td>0.684</td>
</tr>
<tr>
<td>DeepVSA</td>
<td>0.835</td>
<td>0.584</td>
<td>0.944</td>
<td>0.959</td>
<td>0.831</td>
</tr>
<tr>
<td>PalmTree</td>
<td>0.855</td>
<td>0.714</td>
<td>0.95</td>
<td>0.971</td>
<td>0.873</td>
</tr>
<tr>
<td><strong>NEUDep</strong></td>
<td><strong>0.91</strong></td>
<td><strong>0.904</strong></td>
<td><strong>0.977</strong></td>
<td><strong>0.976</strong></td>
<td><strong>0.942</strong></td>
</tr>
</tbody>
</table>
### Downstream Task 2: Function Signature Prediction (Accuracy)

<table>
<thead>
<tr>
<th></th>
<th>Caller</th>
<th></th>
<th></th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
<td>O1</td>
<td>O2</td>
<td></td>
</tr>
<tr>
<td>Ret.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EKLA.</td>
<td>66.62</td>
<td>70.59</td>
<td>73.63</td>
<td>76.19</td>
</tr>
<tr>
<td>NEUDEP</td>
<td>94.65</td>
<td>93.33</td>
<td>95.75</td>
<td>96.41</td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EKLA.</td>
<td>91.56</td>
<td>90.38</td>
<td>91.21</td>
<td>91.55</td>
</tr>
<tr>
<td>NEUDEP</td>
<td>97.03</td>
<td>97.09</td>
<td>98.47</td>
<td>99.17</td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EKLA.</td>
<td>81.82</td>
<td>78.70</td>
<td>81.81</td>
<td>82.03</td>
</tr>
<tr>
<td>NEUDEP</td>
<td>96.08</td>
<td>94.86</td>
<td>97.68</td>
<td>97.51</td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EKLA.</td>
<td>80.28</td>
<td>79.85</td>
<td>81.35</td>
<td>76.63</td>
</tr>
<tr>
<td>NEUDEP</td>
<td>96.88</td>
<td>96.89</td>
<td>97.09</td>
<td>97.24</td>
</tr>
<tr>
<td>Aritiy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EKLA.</td>
<td>92.03</td>
<td>86.02</td>
<td>83.80</td>
<td>82.79</td>
</tr>
<tr>
<td>NEUDEP</td>
<td>98.84</td>
<td>95.44</td>
<td>96.35</td>
<td>95.86</td>
</tr>
</tbody>
</table>
## Downstream Task 3: Matching Indirect Calls
(F1 Score)

<table>
<thead>
<tr>
<th></th>
<th>Arity</th>
<th>Arity+Ret</th>
<th>Arity+Arg</th>
<th>Arity+Arg+Ret</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loose</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeArmor</td>
<td>0.75</td>
<td>0.752</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EKLAVYA</td>
<td>0.777</td>
<td>0.778</td>
<td>0.8</td>
<td>0.801</td>
</tr>
<tr>
<td>NEUDEP</td>
<td><strong>0.783</strong></td>
<td><strong>0.804</strong></td>
<td><strong>0.83</strong></td>
<td><strong>0.843</strong></td>
</tr>
<tr>
<td><strong>Strict</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeArmor</td>
<td>0.777</td>
<td>0.778</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EKLAVYA</td>
<td>0.818</td>
<td>0.817</td>
<td>0.811</td>
<td>0.811</td>
</tr>
<tr>
<td>NEUDEP</td>
<td><strong>0.844</strong></td>
<td><strong>0.853</strong></td>
<td><strong>0.851</strong></td>
<td><strong>0.857</strong></td>
</tr>
</tbody>
</table>
Summary

- Transfer learning can provide a unified framework for learning code representations for binary analysis

  **Trace modeling is a promising way of learning semantics-aware code representations**

  **Accurate**: Outperforms the State-Of-The-Art

  - Matching Semantically Similar Code: +14.3%
  - Type Inference: +12.6%
  - Memory Dependence (Alias) Analysis: +52.6%

  **Efficient**: Outperforms the State-Of-The-Art

  - Matching Semantically Similar Code: 8x speedup
  - Type Inference: 98.1x speedup
  - Memory Dependence (Alias) Analysis: 3.5x speedup
Thanks!
XDA Runtime Speed

File Size (Name)

- 0.5M (specrand_is)
- 4.1M (omnetpp_r)
- 7.8M (cpuxalan_r)
- 22M (blender_r)

- XDA-GPU
- XDA-CPU
- IDA
- Ghidra

- 15x Speedup
- 38x Speedup
- 26x Speedup

Seconds

- 0 100 200 300 400
- 1.6 4.8 16.4 3.5 29 92.3 146.3 185.7 317.1 136 246.2
### Numerical Representation

**Instruction Sequence**
- `sub`
- `ecx`
- `num`
- `add`
- `ecx`
- `num`
- `jmp`
- `ecx`

**Trace-Value Sequence**
- `##`
- `2`
- `1`
- `##`
- `1`
- `3`
- `##`
- `4`

Replace Numerics with Single Token `num` in **Instruction Sequence**
Put the Numerics in **Trace-Value Sequence**

Pad each Token as a Fixed-Length 8-Byte Sequence:

```
0xdeadbeef 00 00 00 00 de ad be ef
```

Represent the Value (*i.e.*, Fixed-Length 8-Byte Sequence) with an Bidirectional LSTM (bi-LSTM):

```
00 00 00 00 de ad be ef
```

**bi-LSTM**
- LSTM
- LSTM
- LSTM
- LSTM
- LSTM
- LSTM
- LSTM

`Emb(0xdeadbeef)`
Embedding of `0xdeadbeef`

64-bit Arch: $2^{64}$ Possible Values
Prohibitively Large Vocabulary when Embedding Code