Learning Program Dependencies with ML

Dongdong She

Kexin Pei

Abhishek Shah

Columbia University
IN THE CITY OF NEW YORK

What’s happening?

```
input_byte = read("foo.txt")
if input_byte[0] == '1':
    // divide by zero error
else:
    // exit gracefully
```

Problem

- Programs have complex dependencies
  - Control flow
  - Data flow
- Program Analysis
  - Examples:
    - Taint Analysis
    - Symbolic Execution
  - Do not scale well

Solution

- Use ML to automatically learn dependencies
  - ML can excel at finding relationships in data
    - NLP: tagging parts-of-speech in a sentence
- Finding them is useful
  - Code Coverage
  - Debugging
  - Vulnerability Discovery

Outline

- Learn Dependencies
  - Examine 1 program dynamically
    - Neuro-symbolic Execution
      - Learn path dependencies
  - Examine many programs statically
    - Idea of "Big Code"
    - Binaries
      - Learn dependency between variable and registers
    - Source Code
      - Learn dependency between variable use and definition

Neuro-symbolic Execution: Augmenting Symbolic Execution with Neural Constraints

Dongdong She

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Problem

Symbolic execution has many limitations.

- Poor scalability by path explosion.
- Language-specific implementation.
- Failure to model complex dependencies.
- Limited expressiveness of satisfying theories.

A simple example

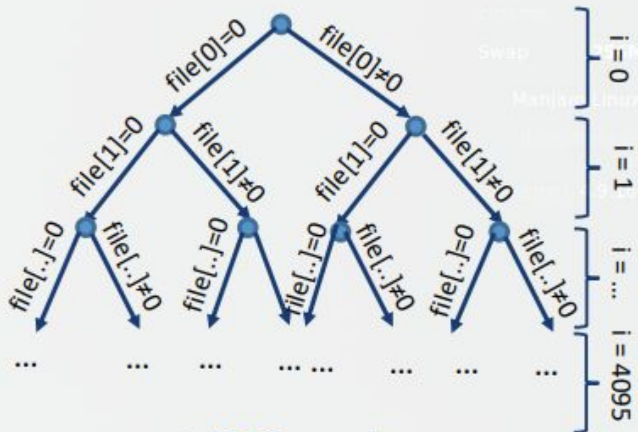
```
int main (…) {
  if (strlen(filename)>1 && filename[0]=='-')
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int *file,...) {
  static double data[4096], value;
  read_double_value(file, ... );
  value = fabs (data [0]);
  for(i=0; i<4096; i++)
    if(file[i] == 0.0) count++;
  data[1] /= (value+count-3);
  …
}
```

#1 Limitations of SMT Solvers

#2 Unmodeled Semantics

#3 Path Explosion

Candidate Vulerability Point (CVP): Divide-by-zero



2^{4096} paths

Locations of interest

Candidate Vulnerability Point(CVP)

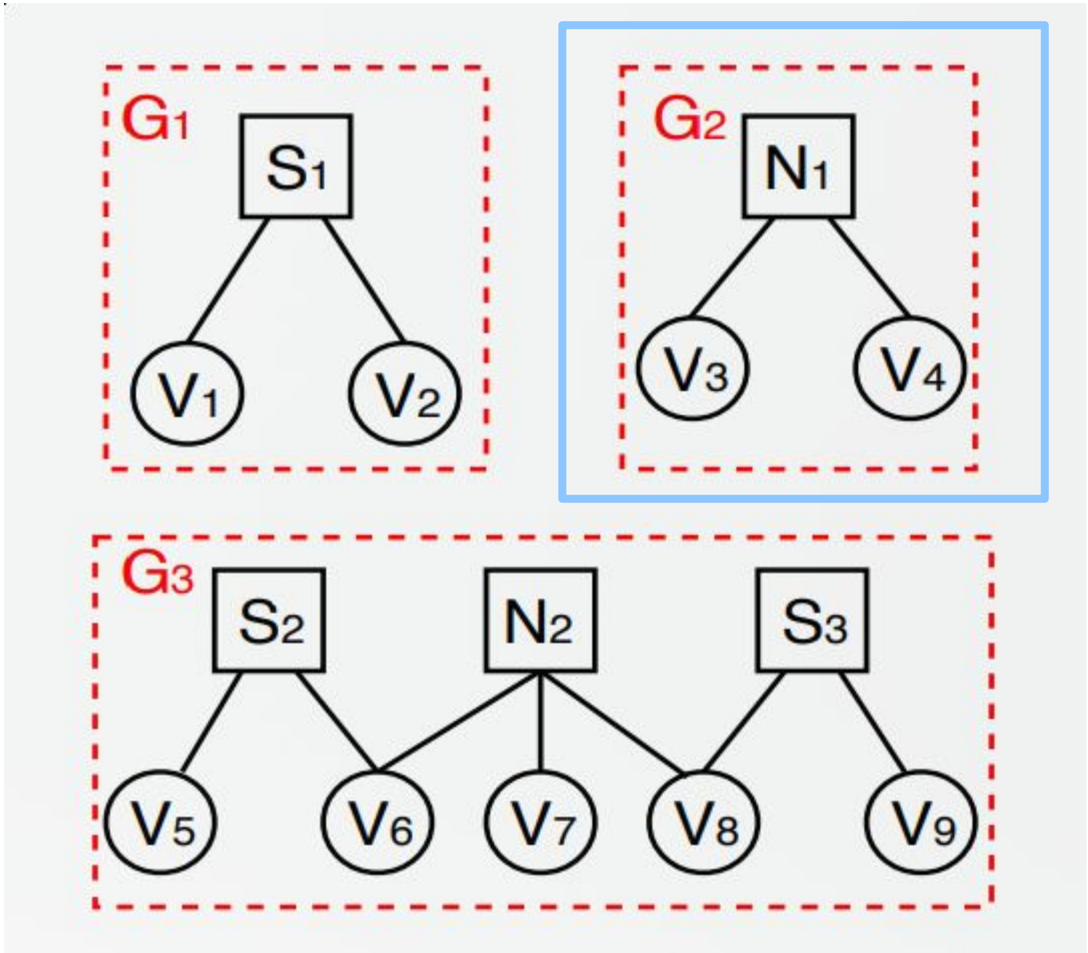
- Statically analyze program in advance.
- Identify two specific program locations.
  - Division operations (check zero division).
  - Boundary checking in buffer accesses.
- Instrument CVP to record values of denomimators/ index number for further training.

Neuro-symbolic execution

- Represent most of program logic with symbolic constraints.
- Approximate the remaining logic that is hard to solve with NN.
- Solve the combination of exact constraints & approximated constraints.

Constraints = S_1 ^ N_1 ^ N_2 ^ S_2

Neuro-symbolic execution



How to generate the neuro-constraints?

Neuro-constraints

Problem

Inputs:

1. Source code
2. Symbolic Variables (e.g., `filename` & `file`)
3. Candidate Vulnerability Points (CVPs)
   - Divide by zero
   - Buffer overflow

Outputs:

Validated Exploits

```
int main (…) {
  if (strlen(filename)>1 && filename[0]=='-')
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int *file,...) {
  static double data[4096], value;
  read_double_value(file, ... );
  value = fabs (data [0]);
  for(i=0; i<4096; i++)
    if(file[i] == 0.0) count++;
  data[1] /= (value+count-3);
  …
}
```

CVP: Divide-by-zero

Neuro-constraints

Key Insights

Values of Symbolic Variables

Values of Vulnerable Variables in CVP

Learn an approximation with small number of I/O examples

100,000 samples

```
int main (…) {
  if (strlen filename >1 && filename[0]==‘-’)
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int * file ...) {




  data[1] /= (value+count-3);
  …
}
```

CVP: Divide-by-zero

Neuro-constraints

Approach

1. Neural nets can represent a large category of functions (universal approximation theorem).

2. Multiple applications show that neural nets are learnable for many practical functions.

```
int main (…) {
  if (strlen filename >1 && filename[0]==‘-’)
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int * file ...) {
```

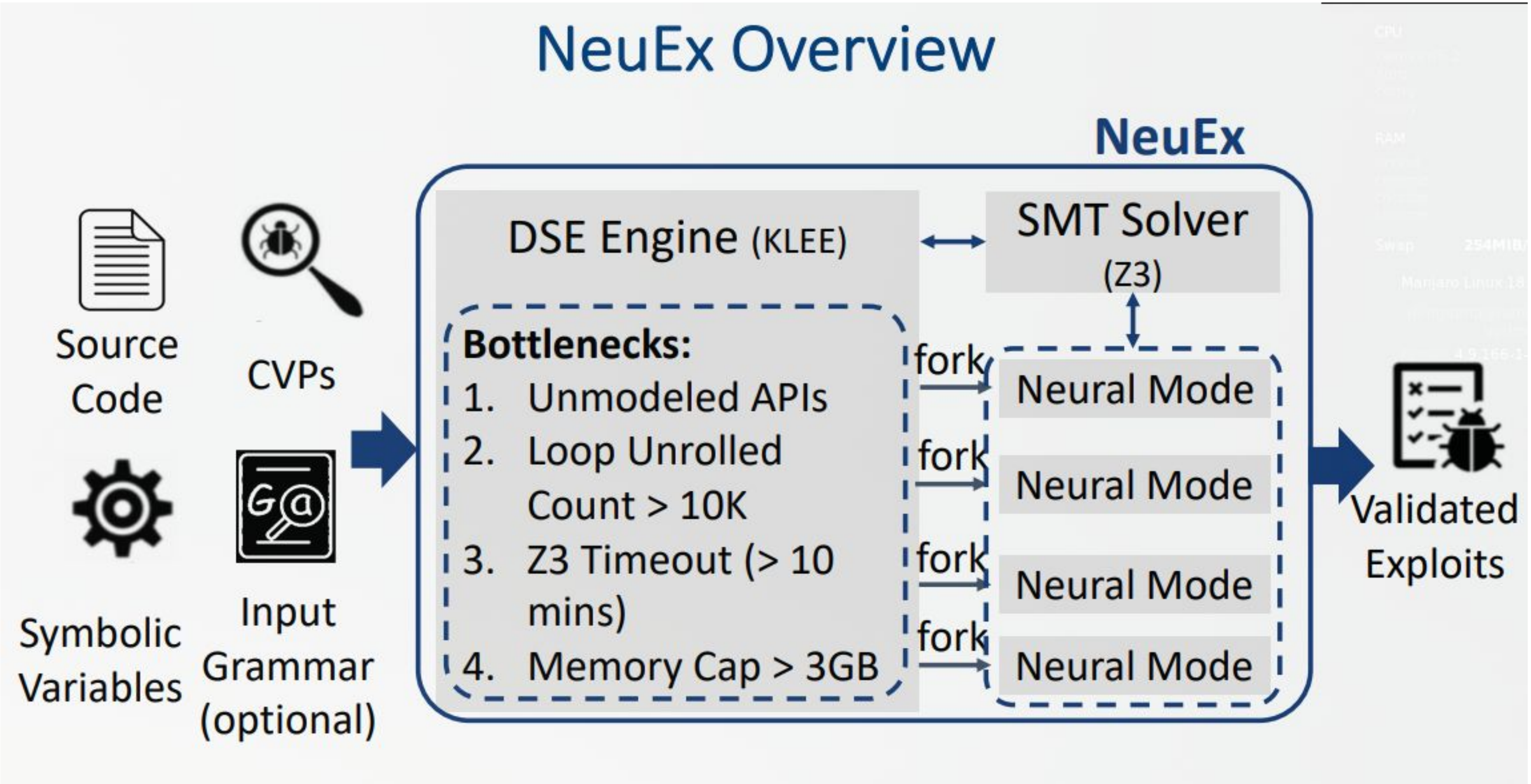
Approximate Constraint (as a neural net):

file  count & value

```
  data[1] /= (value+count-3);
  …
}
```

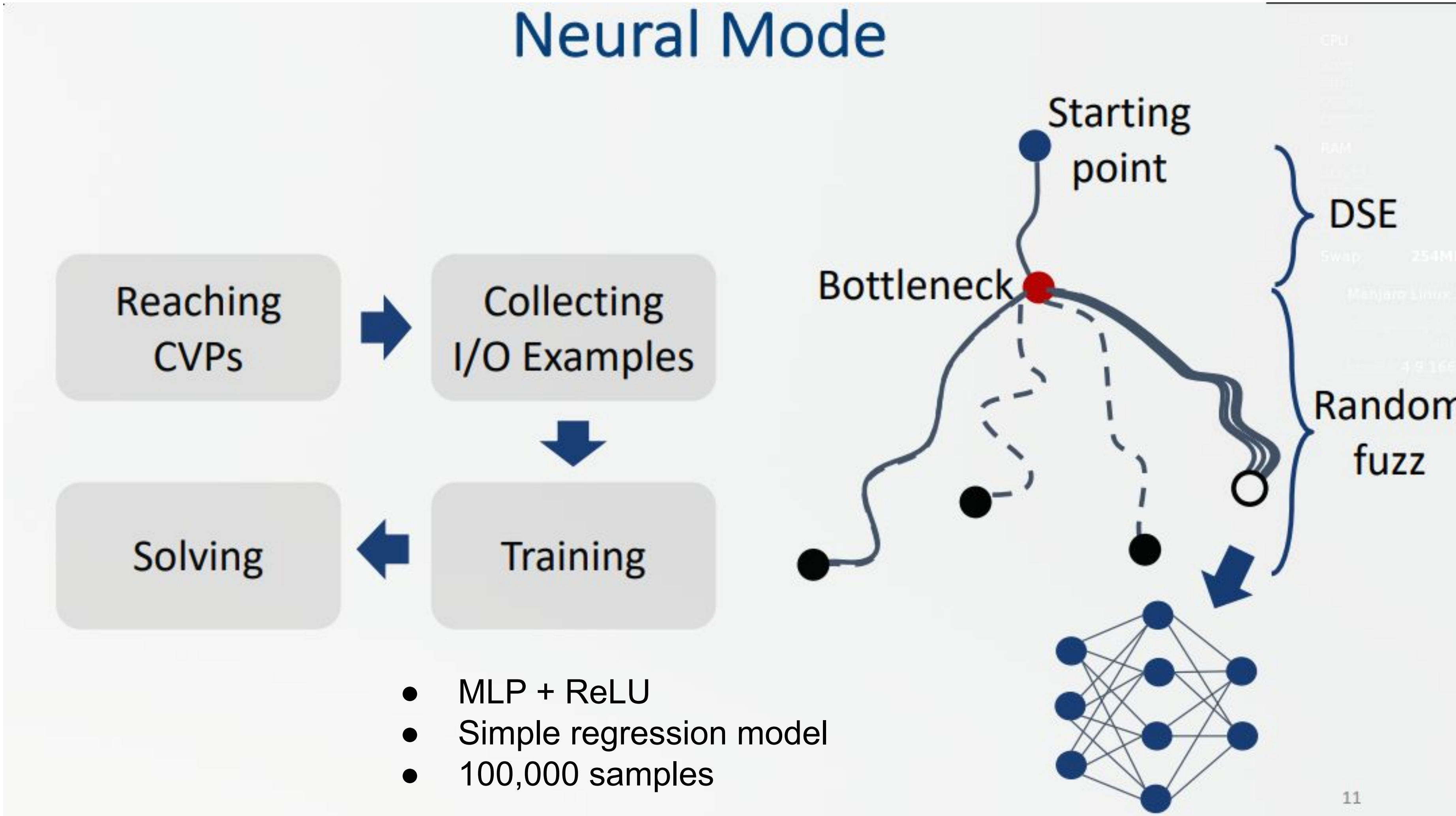
CVP: Divide-by-zero

Overview



Hybrid mode design (symbolic mode and neural mode)

Neural Mode



- MLP + ReLU
- Simple regression model
- 100,000 samples

Constraints

Generated Constraints

1. Reachability constraints:

$$strlen(filename) \leq 1$$
$$\vee\ filename \neq '-'$$

$$\wedge$$

$$N: infile \rightarrow (value, count)$$

2. Vulnerability condition:

$$value + count - 3 == 0$$

```
int main (…) {
  if (strlen(filename)>1 && filename[0]==‘-’)
    exit(1)
  copy_data(…);
  …
}
void copy_data(..., int *file,...) {
  static double data[4096], value;
  read_double_value(file, ... );
  value = fabs (data [0]);
  for(i=0; i<4096; i++)
    if(file[i] == 0.0) count++;
  data[1] /= (value+count-3);
  …
}
```

CVP: Divide-by-zero

Constraints

Constraint Solving

1. Reachability constraints:

$$\boxed{\begin{array}{c} strlen(filename) \leq 1 \\ \lor\ filename \neq '-' \end{array}}$$

Purely symbolic constraints: ➡ SMT solver

No variable shared with neural constraints

$$\land$$

$$N: infile \rightarrow (value, count)$$

2. Vulnerability condition:

$$value + count - 3 == 0$$

Mixed constraints:

Including both neural constraints and symbolic constraints with shared variables

How to solve mixed constraints

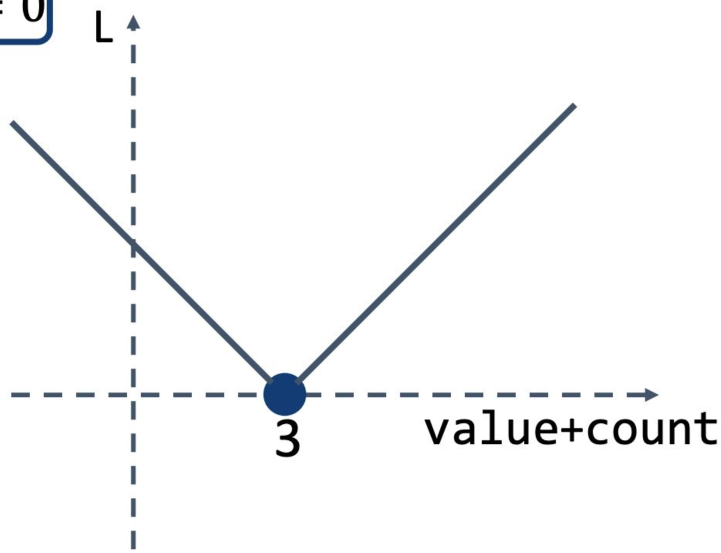
Symbolic constraints

Optimization objectives of the neural net

Encoding Symbolic Constraints as an Optimization Objective

$N: infile \rightarrow (value, count) \wedge \boxed{value + count - 3 == 0}$

Symbolic constraint



Criterion for crafting the loss function:
The minimum point of the loss function satisfies the symbolic constraints.

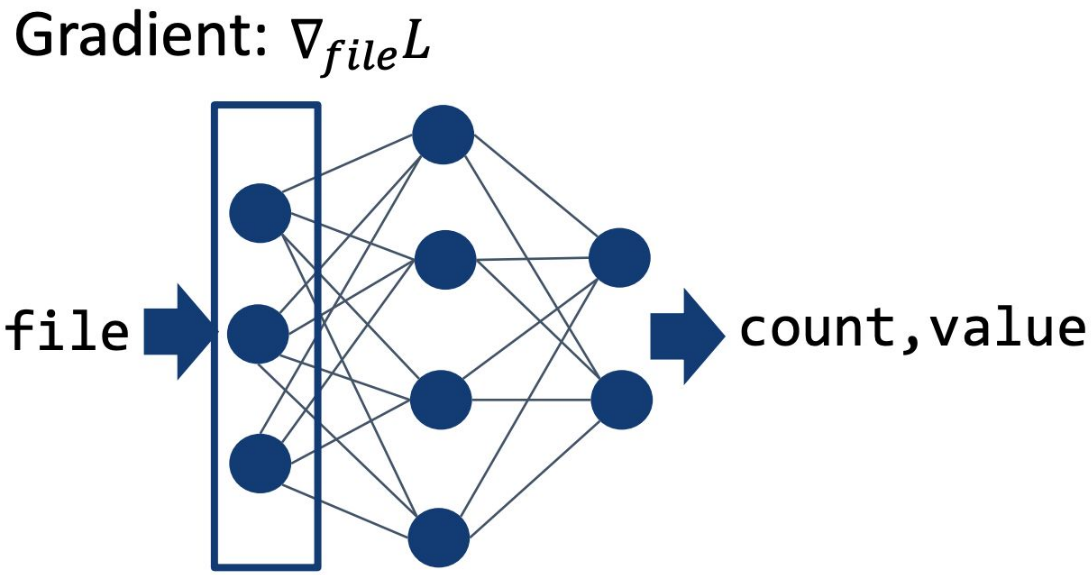
One possible encoding:

$$L = abs(value + count - 3)$$

Constraints => Loss

Symbolic Constraint	**Loss Function (L)**
$S_1 ::= a < b$	$L = max(a - b + \alpha, 0)$
$S_1 ::= a > b$	$L = max(b - a + \alpha, 0)$
$S_1 ::= a \leq b$	$L = max(a - b, 0)$
$S_1 ::= a \geq b$	$L = max(b - a, 0)$
$S_1 ::= a = b$	$L = abs(a - b)$
$S_1 ::= a \neq b$	$L = max(-1, -abs(a - b + \beta))$
$S_1 \wedge S_2$	$L = L_{S_1} + L_{S_2}$
$S_1 \vee S_2$	$L = min(L_{S_1}, L_{S_2})$

Solving Mixed Constraints via Gradient Descent



count	value	loss
257	20	274
1	20	18
0	20	17
…	…	…
0	3	0

Concretely validate the exploit

file: 000…

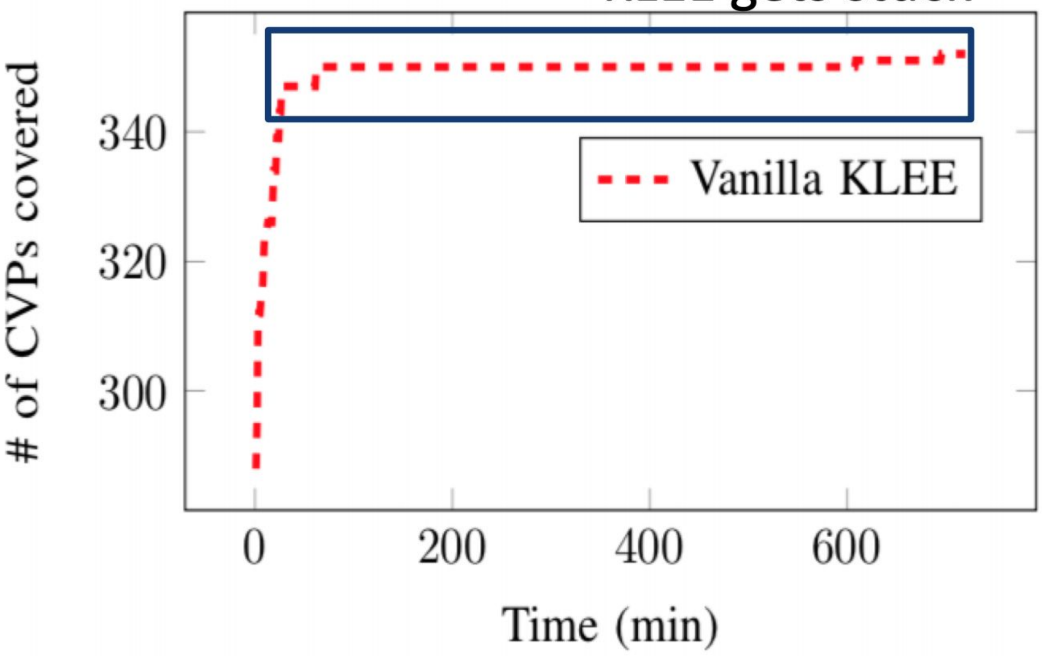
Evaluation

- **Recall**: Neural mode is only triggered when DSE encounters bottlenecks
- **Benchmarks**: 7 Programs known to be difficult for classic DSE
  - 4 Real programs
    - cURL: Data transferring
    - SQLite: Database
    - libTIFF: Image processing
    - libsndfile: Audio processing
  - LESE benchmarks
    - BIND, Sendmail, and WuFTP

Include:
1. Complex loops
2. Floating-point variables
3. Unmodeled APIs

CVP Coverage & Bottlenecks for DSE

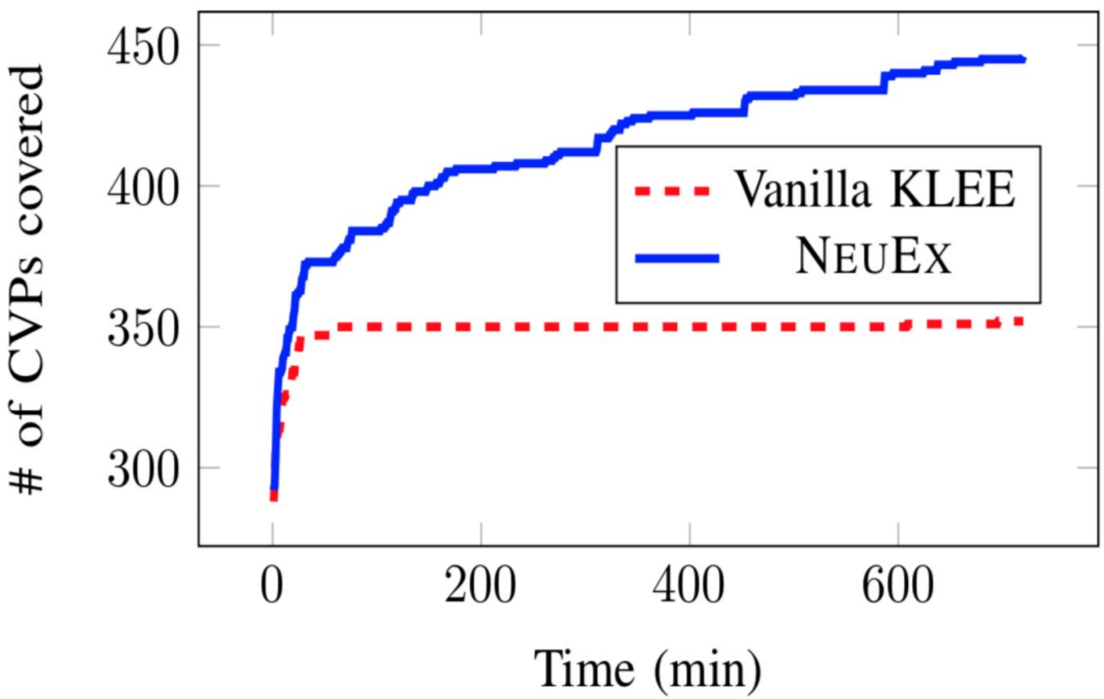
KLEE gets stuck



<u># of bottlenecks: 61</u>

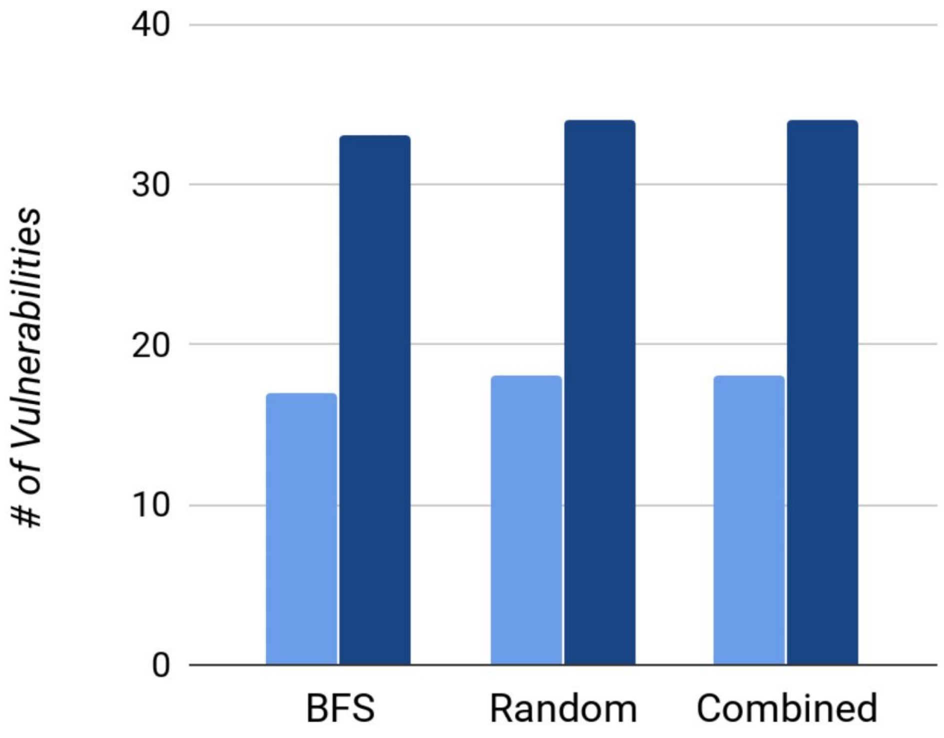
- Unmodeled APIs (6)
- Complicated loops (53)
- Z3 timeout (1)
- Memory exhaustion (1)

CVP Coverage of NeuEx vs KLEE



The number of CVPs reached or covered by NeuEx is **<u>25%</u>** higher than vanilla KLEE.

of Bugs Found by NeuEx vs KLEE



NeuEx finds **94%** and **89%** more bugs than vanilla KLEE in BFS and RAND mode in 12 hours.

Debin: Recovering Stripped Info from Binaries

Kexin Pei

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Binaries with debug symbols

x86 malware samples from VirusShare

Assembly

```
80534BA:
push  %ebp
push  %edi
push  %esi ...
```

Debug symbols

```
80534BA  rfc1035_init int
8053DB1  fopen64      int
8063320  num_entries  int
              ⋮
```

Binary with debug symbols

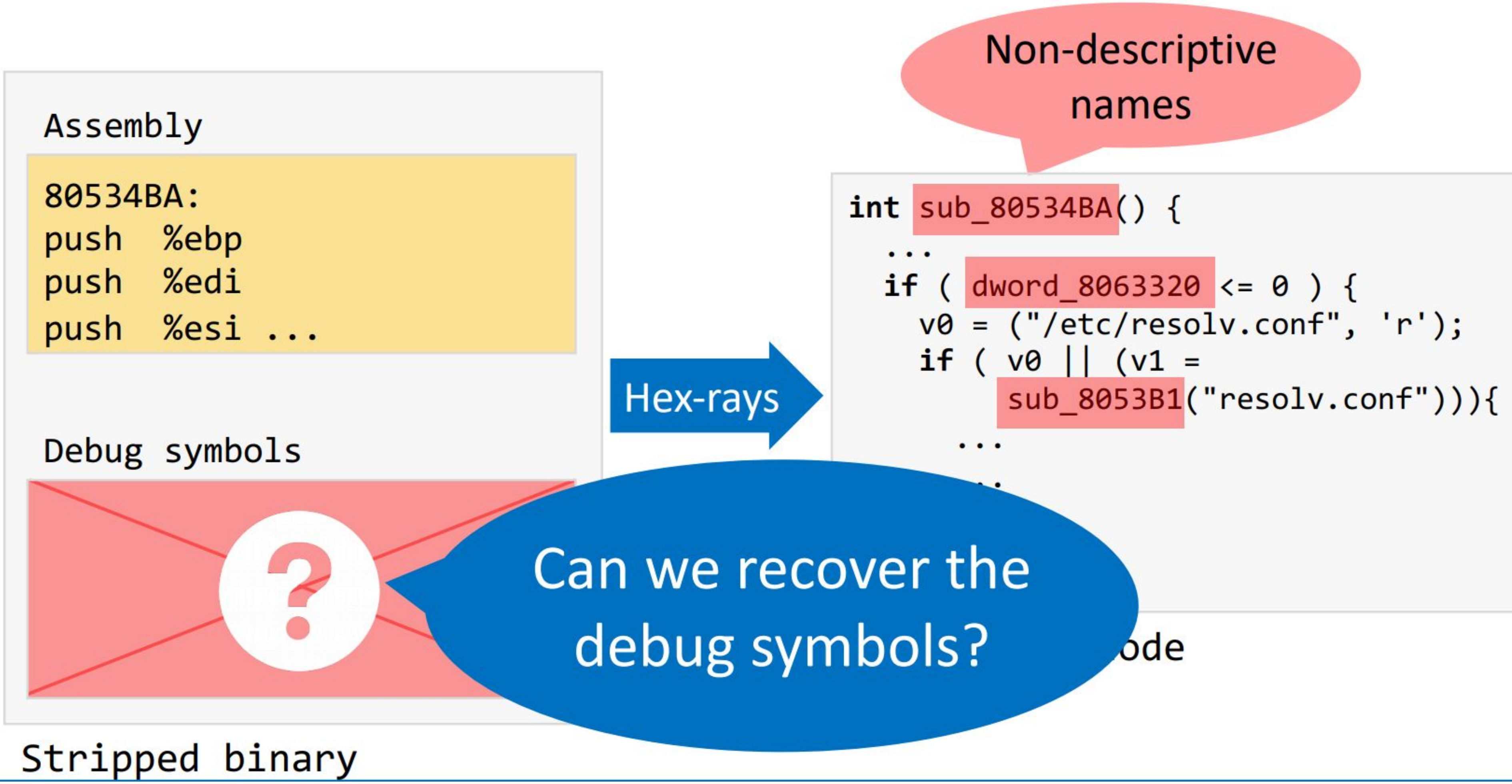
Hex-rays

Descriptive names for functions and variables

```
int rfc1035_init() {
  ...
  if ( num_entries <= 0 ) {
    v0 = ("/etc/resolv.conf", 'r');
    if ( v0 || (v1 =
           fopen64("resolv.conf"))){
      // code to read and
      // manipulate DNS settings
    }
  ...
}
```

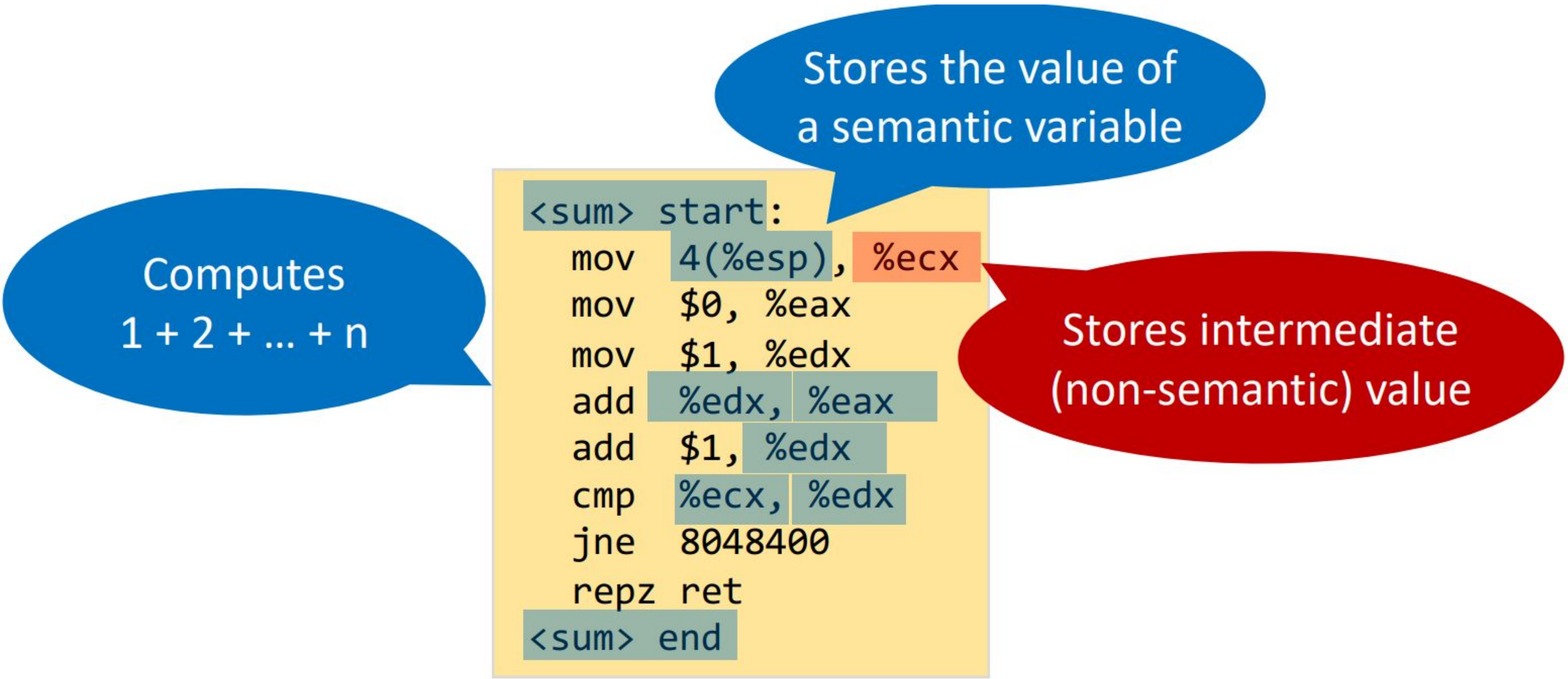
Decompiled code

Stripped Binaries



Challenges

1. No mapping from registers and memory offsets to semantic variables



Challenges

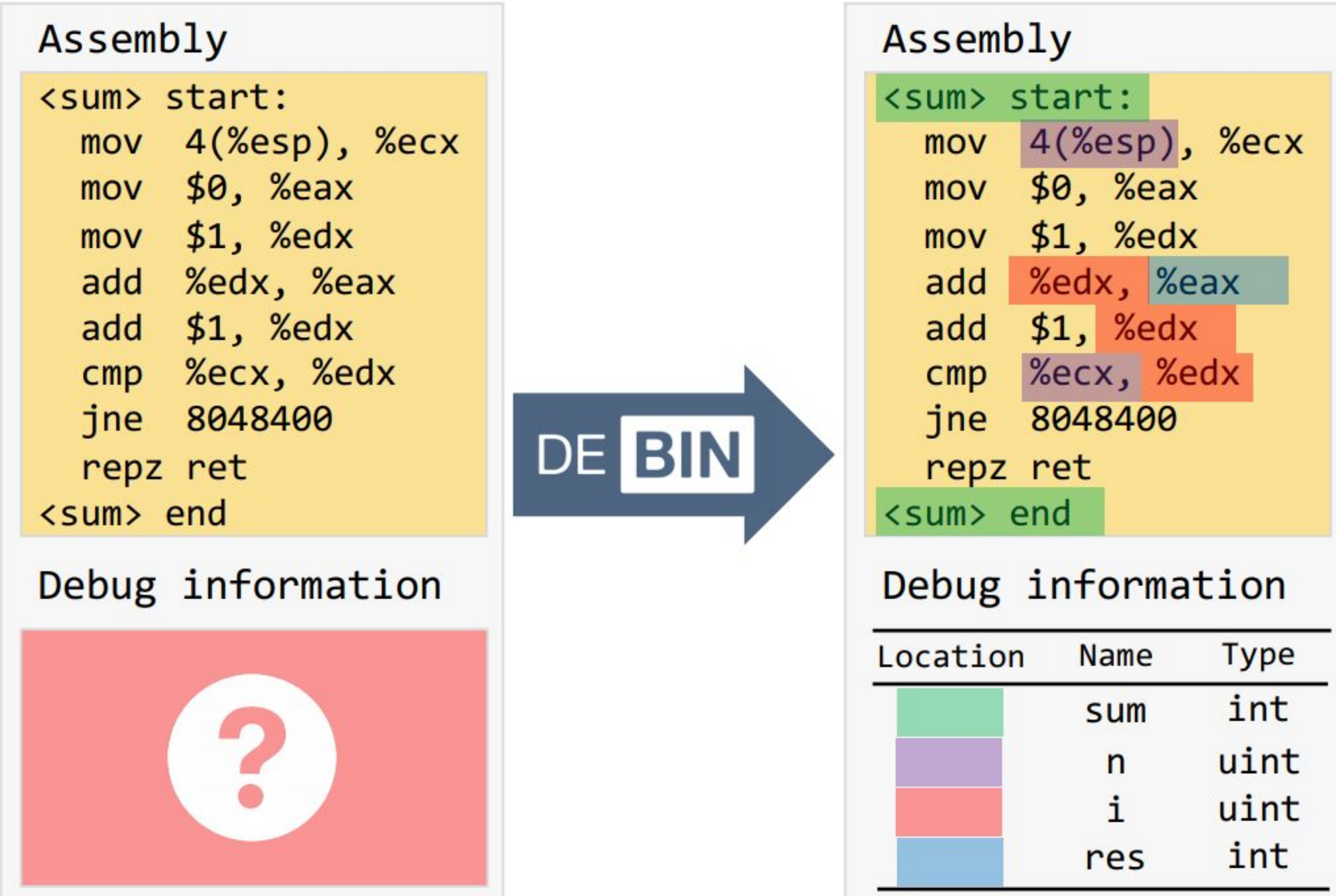
2. No names and types

```
<sum> start:
  mov  4(%esp), %ecx
  mov  $0, %eax
  mov  $1, %edx
  add  %edx, %eax
  add  $1, %edx
  cmp  %ecx, %edx
  jne  8048400
  repz ret
<sum> end
```

Store the values of the unsigned integer variable n

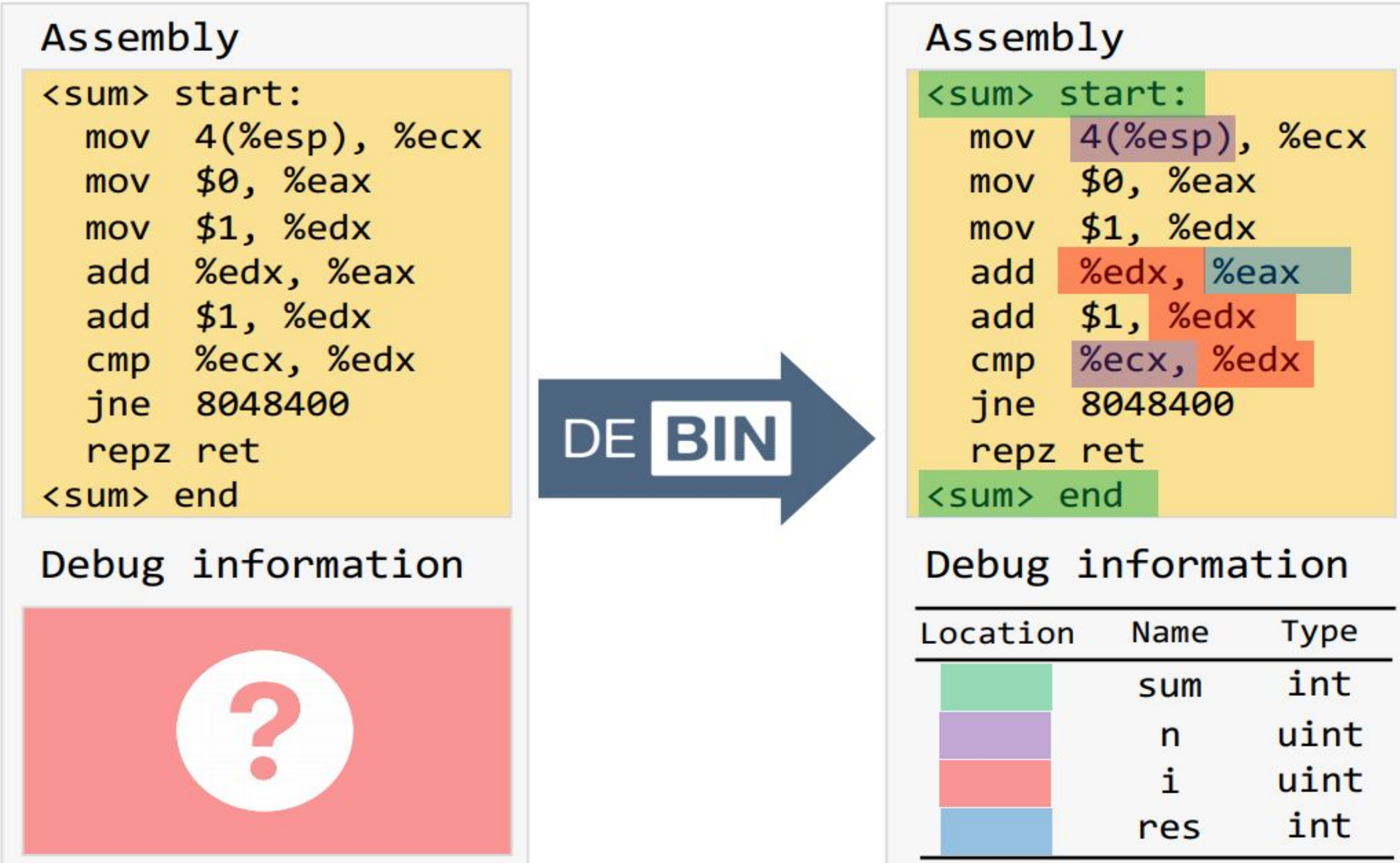
Stores the result in an integer variable res

DeBIN: Recovering debug info



Design Choice

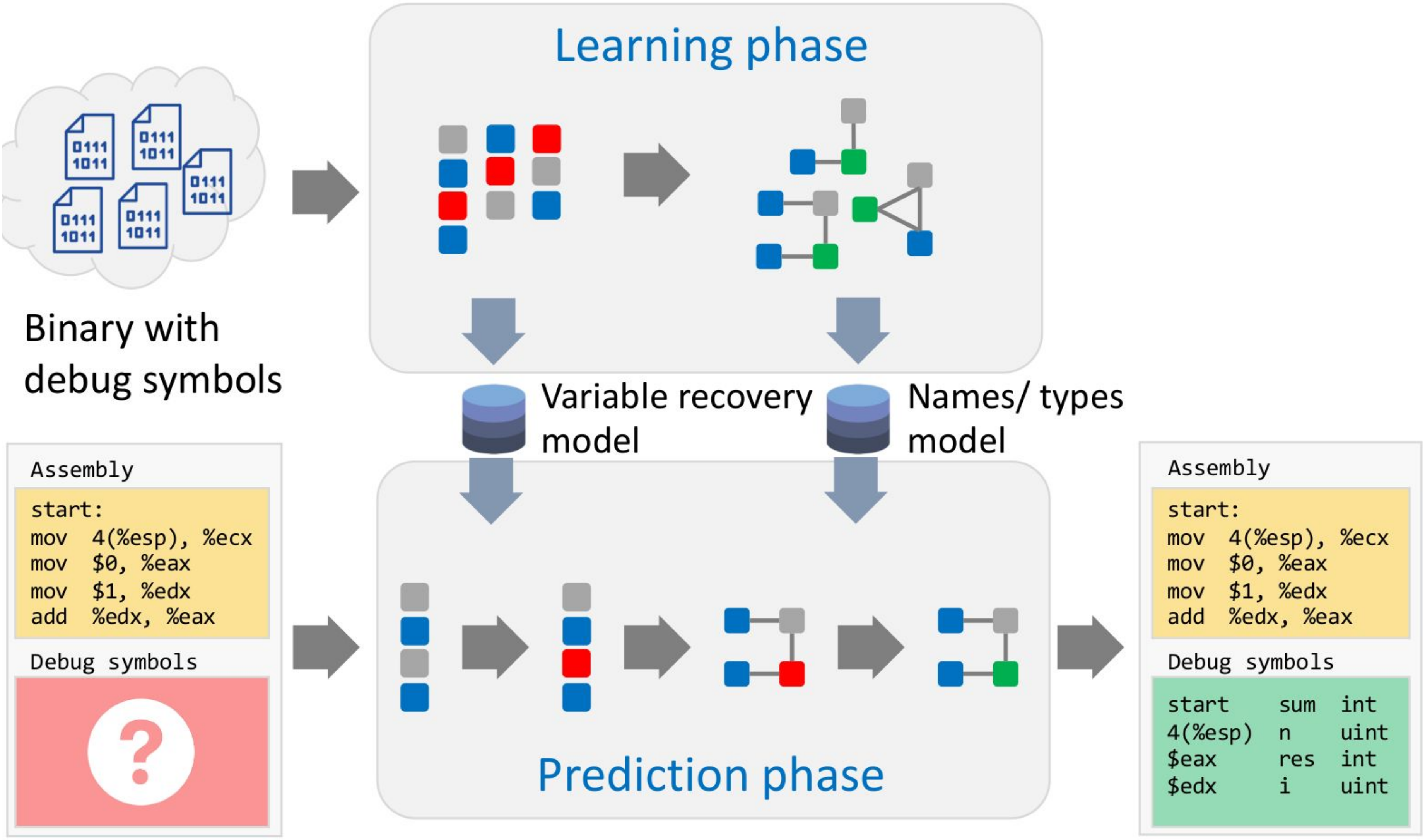
How will you do this?



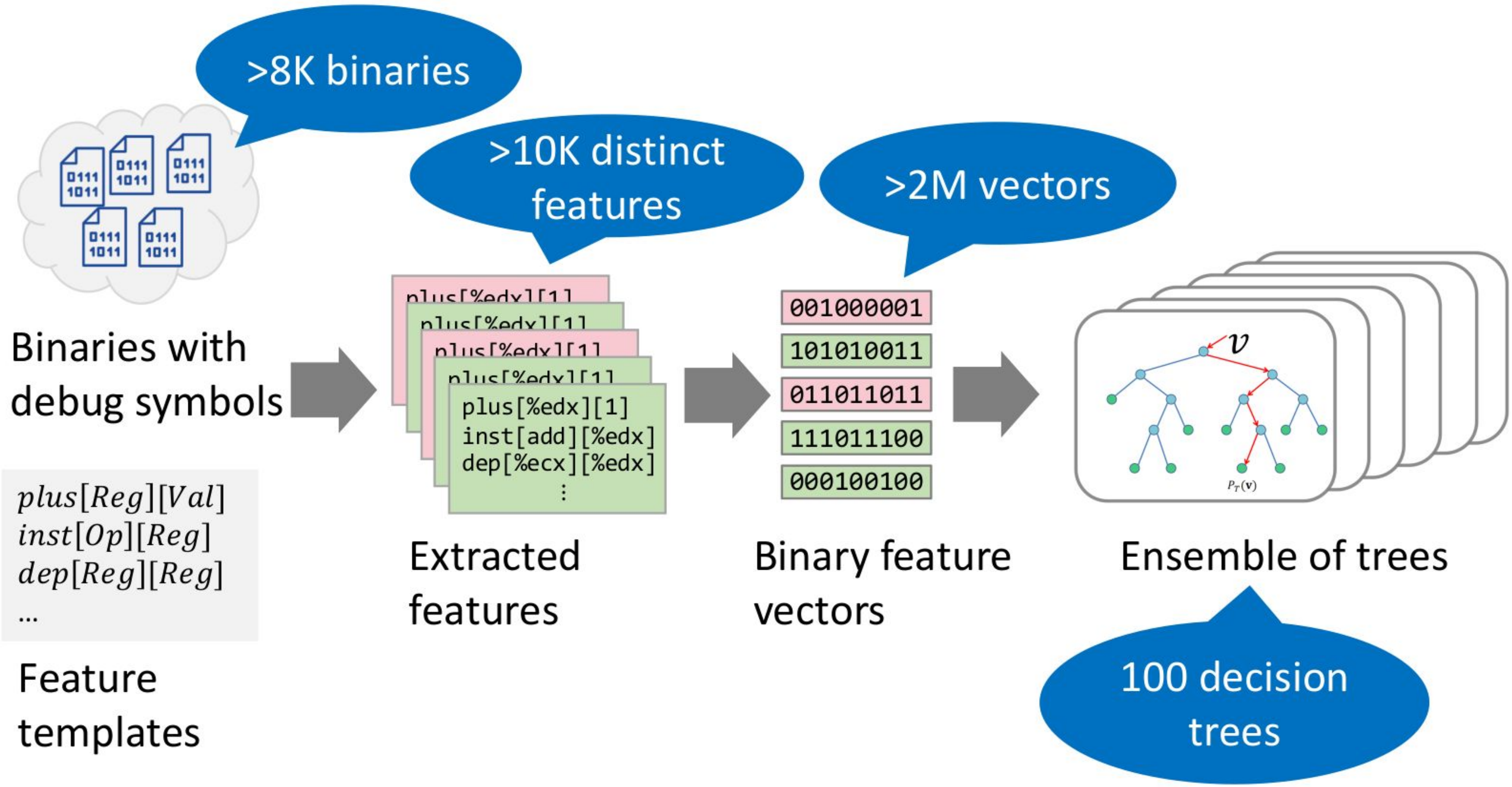
Recap: importance of dependency

1. Naive way of doing this?
   a. Feature template
   b. Individual classification
2. Smarter way of doing this?
   a. RNN/LSTM
   b. Sequential dependency
3. More advanced (best result):
   a. PGM(CRF,MRF,Bayesian Network)/TreeLSTM/GNN/GCN/GGNN...
   b. **Structured** learning

How does Debin work?



Step 1: Recovering Variables



Step 1: Recovering Variables

Variable recovery

Register

%edx.2

Features

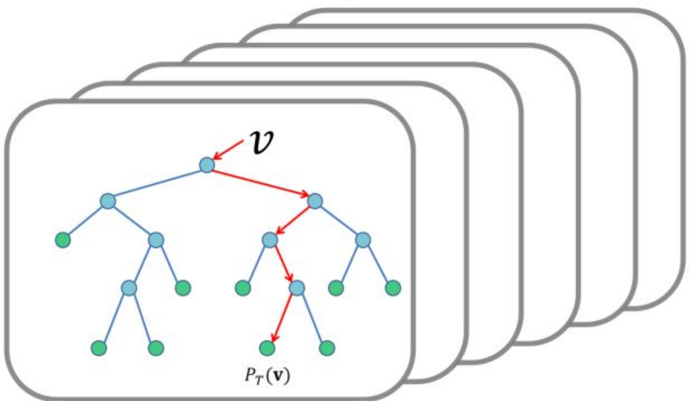
```
plus[%edx][1]
inst[add][%edx]
⋮
```

Assembly

```
mov   4(%esp), %ecx
mov   $0, %eax
mov   $1, %edx
add   %edx, %eax
add   $1, %edx.2
cmp   %ecx, %edx
jne   8048400
repz  ret
```

Feature vector v

`00100101010001`



sem
(DeBIN will predict name and type)

tmp
(stores an intermediate value)

Extremely randomized trees

Decision tree:

- One dataset
- All features

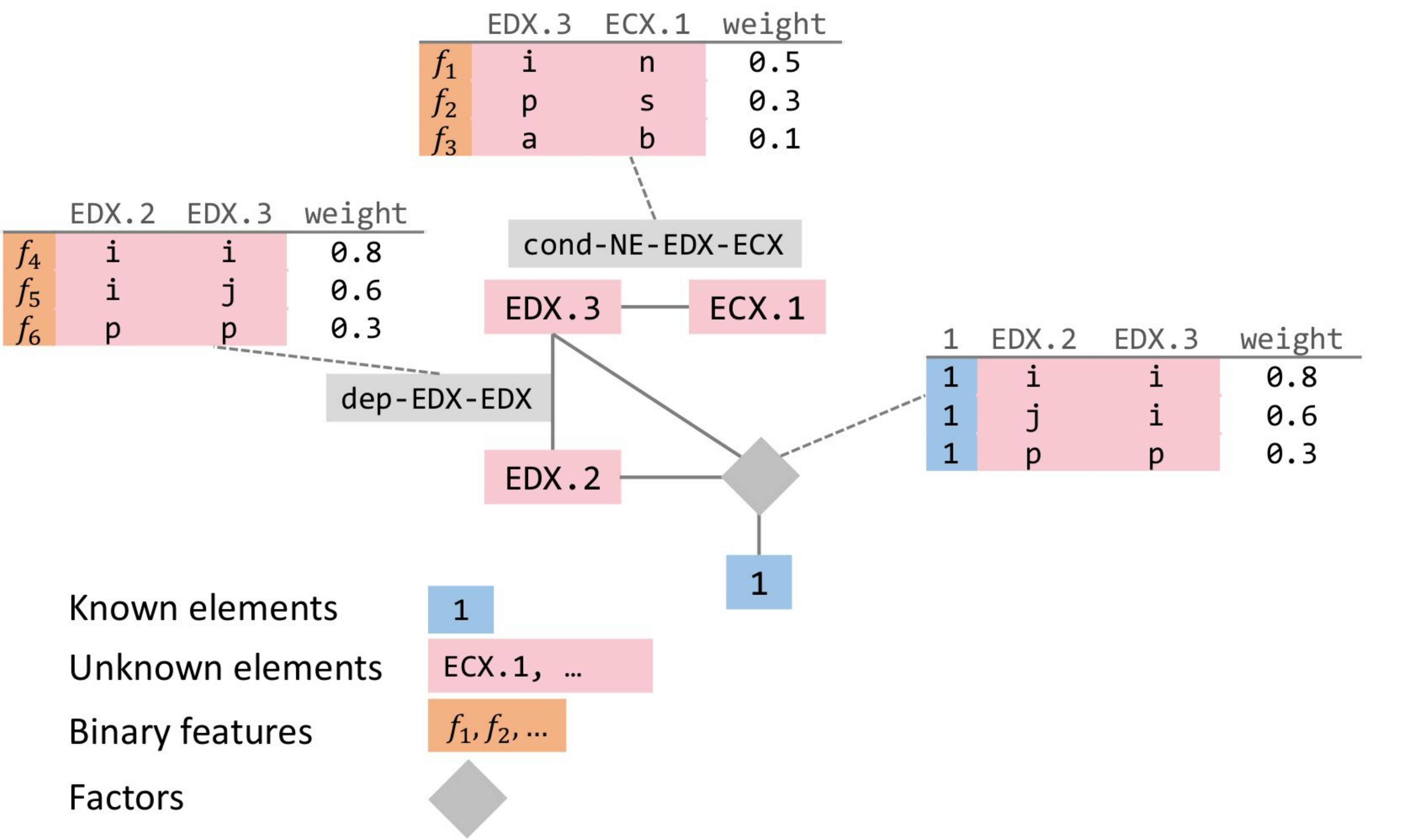
Random forest:

- Multiple sampled sub-dataset
- Sampled set of features

Extremely Randomized trees:

- Randomized division of feature values

Step 2: Predicting names and types



Pairwise Feature functions

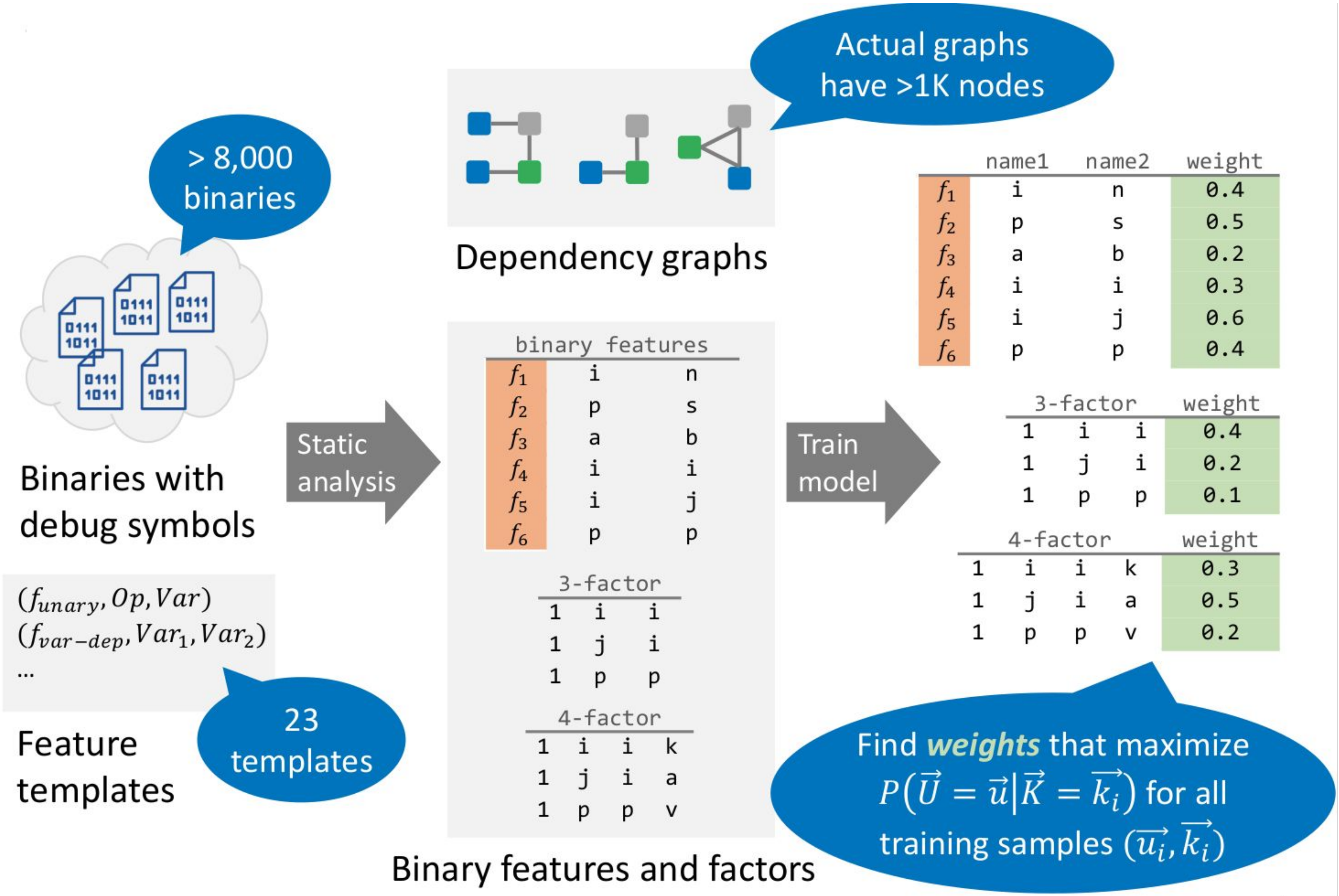
Relationship	Template	Condition for adding an edge
	Function Relationships	
Element used in Function	(f, v, `func-`**`loc`**(v)) (f, a, `arg-`**`loc`**(a)) (f, c, `func-str`) (f, s, `func-stack`)	variable v is accessed inside the scope of function f variable a is an argument of function f by calling conventions string constant c is accessed inside the scope of function f stack location s is allocated for function f
Function Call	(f_1, f_2, `call`)	function f_2 is called by function f_1
	Variable Relationships	
Instruction	(v, *insn*, `insn-`**`loc`**(v))	there is an instruction *insn* (e.g., `add`) that operates on variable v
Location	(v, l, `locates-at`)	variable v locates at location l (e.g., memory offset `mem[RSP+16]`)
Locality	(v_1, v_2, `local-`**`loc`**(v_1))	variable v_1 and v_2 are locally allocated (e.g., `EDX.2` and `EDX.3`)
Dependency	(v_1, v_2, `dep-`**`loc`**(v_1)-**`loc`**(v_2))	variable v_1 is dependent on variable v_2
Operation	(v, *op*, `unary-`**`loc`**(v)) (n_1, n_2, *op*-**`loc`**(n_1)-**`loc`**(n_2)) (v_1, v_2, `phi-`**`loc`**(v_1))	unary operation *op* (e.g. unsigned and low cast) on variable v binary operation *op* (e.g., +, left shift « and etc.) on node n_1 and n_2 there is a ϕ expression in BAP-IR: $v_1 = \phi(\ldots v_2, \ldots)$
Conditional	(v, *op*, `cond-unary`) (n_1, n_2, `cond-`*op*-**`loc`**(n_1)-**`loc`**(n_2))	there is a conditional expression *op* (v) (e.g., `not(EAX.2)`) there is a conditional expression n_1 *op* n_2 (e.g. `EDX.3!=ECX.1`)
Argument	(f, a, `call-arg-`**`loc`**(a))	there is a call $f(\ldots, a, \ldots)$ with argument a
	Type Relationships	
Operation	(t, *op*, `t-unary-`**`loc`**(t)) (t_1, t_2, `t-`*op*-**`loc`**(t_1)-**`loc`**(t_2)) (t_1, t_2, `t-phi-`**`loc`**(t_1))	unary operation *op* on type t binary operation *op* on type t_1 and t_2 there is a ϕ expression: $t_1 = \phi(\ldots t_2, \ldots)$
Conditional	(t, *op*, `t-cond-unary`) (t_1, t_2, `t-cond-`*op*-**`loc`**(t_1)-**`loc`**(t_2)))	there is a unary conditional expression *op* (t) there is a binary conditional expression t_1 *op* t_2
Argument	(f, t, `t-call-arg-`**`loc`**(t))	call $f(\ldots, t, \ldots)$ with an argument of type t
Name & Type	(v, t, `type-`**`loc`**(v)) (f, t, `func-type`)	variable v is of type t function f is of type t

Factor Feature functions

Factors:

- All nodes that appear in the same ϕ expression of BAP-IR
- Function node of a call and its arguments
- Elements that are accessed in the same statement

Learning to predict



End-to-end recovery of debug information

```
<sum> start :
  mov  4(%esp), %ecx
  mov  $0, %eax
  mov  $1, %edx
  add  %edx, %eax
  add  $1, %edx.2
  cmp  %ecx.1, %edx.3
  jne  8048400
  repz ret
<sum> end
```

Stripped binary

Registers / mem offsets

EDX.2 EDX.3

EDX.1 ECX.1

Known elements

0 1 mov

Semantic variables

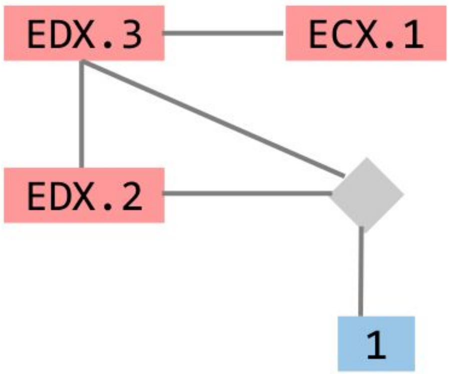
EDX.2 EDX.3 ECX.1

Temporary

EDX.1

Known elements

0 1 mov

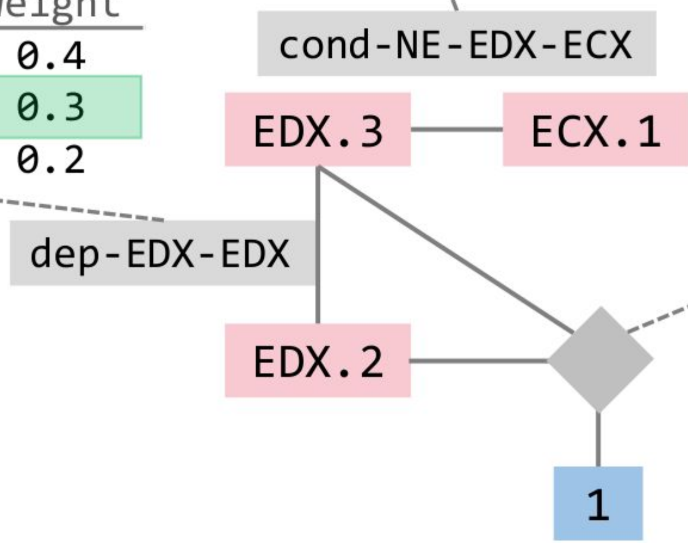


End-to-end recovery of debug information

MAP inference

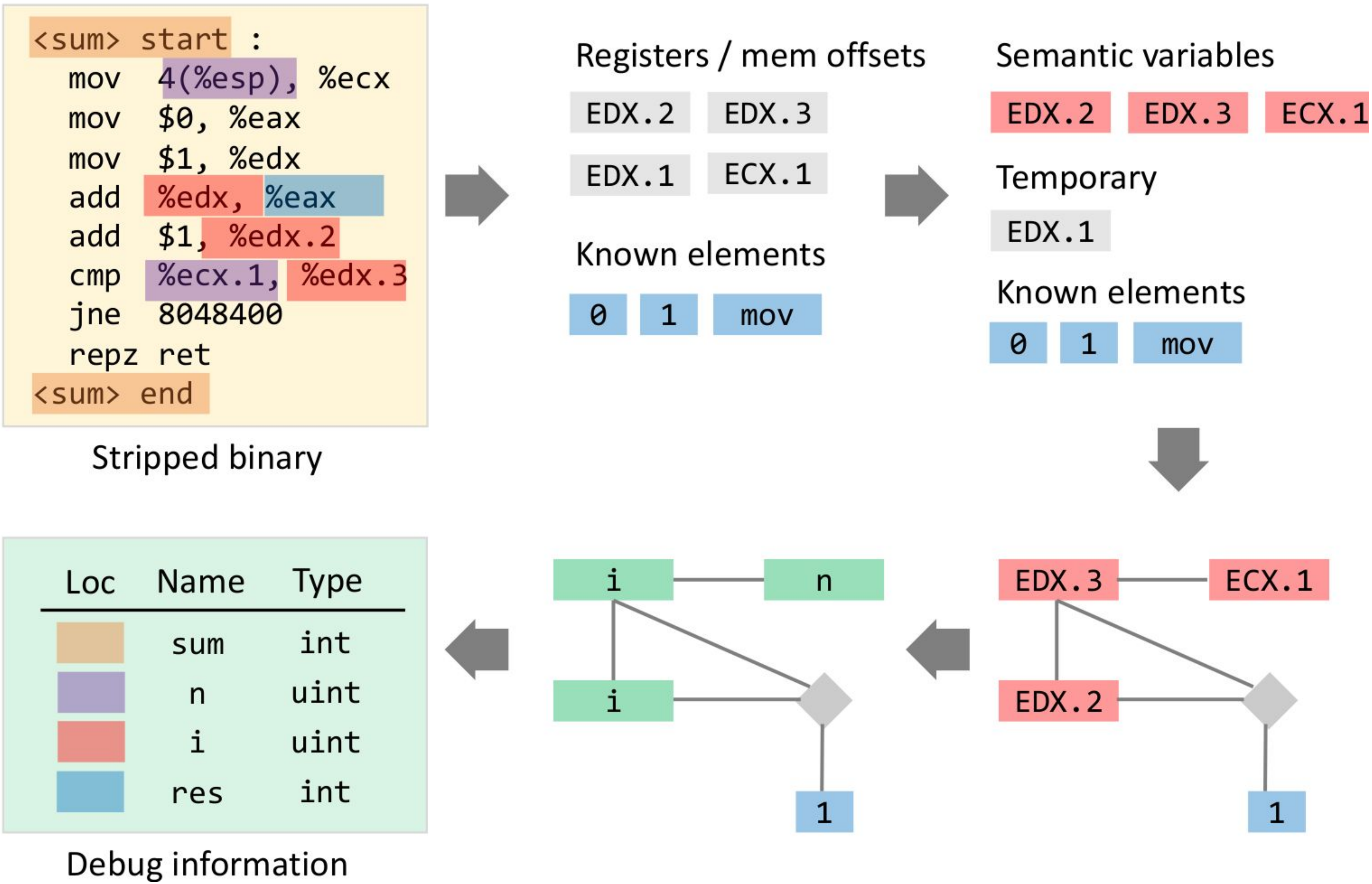
	EDX.3	ECX.1	weight
f_1	i	n	0.5
f_2	p	s	0.3
f_3	a	b	0.1

	EDX.2	EDX.3	weight
f_4	p	p	0.4
f_5	i	i	0.3
f_6	i	j	0.2



1	EDX.2	EDX.3	weight
1	i	i	0.8
1	j	i	0.6
1	p	p	0.3

End-to-end recovery of debug information



Implementation

Static analysis: BAP

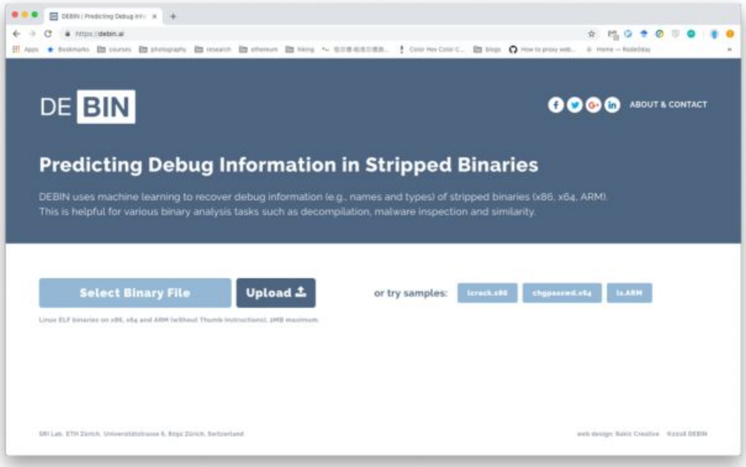
https://github.com/BinaryAnalysisPlatform/bap/

Learning and inference

scikit learn http://scikit-learn.org

NICE 2 Predict http://nice2predict.org

Linux 830 Linux packages
x86, x64, ARM

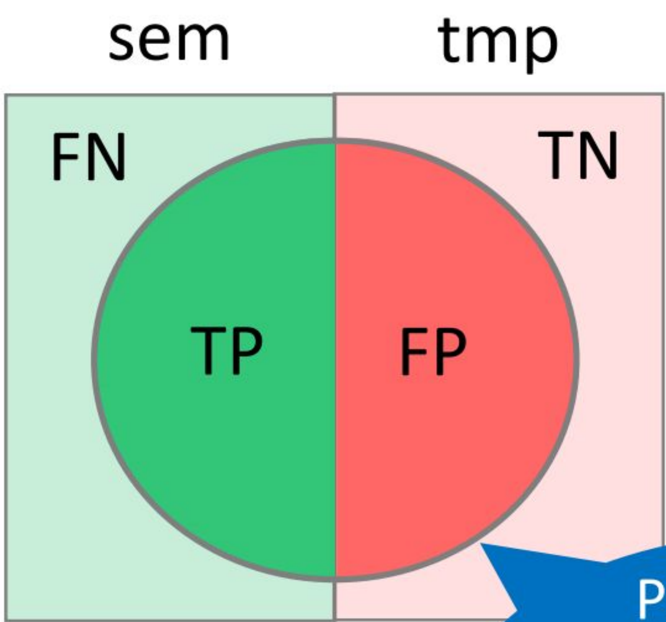


https://debin.ai

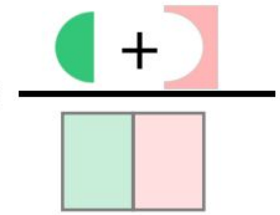
Evaluation

- How accurate is DeBIN's variable recovery?
- How accurate is DeBIN's name and type prediction?
- Is DeBIN useful for malware inspection?

Variable recovery accuracy



$$\text{Accuracy} = \frac{|\mathrm{TP}|+|\mathrm{TN}|}{|\mathrm{sem}|+|\mathrm{tmp}|} =$$

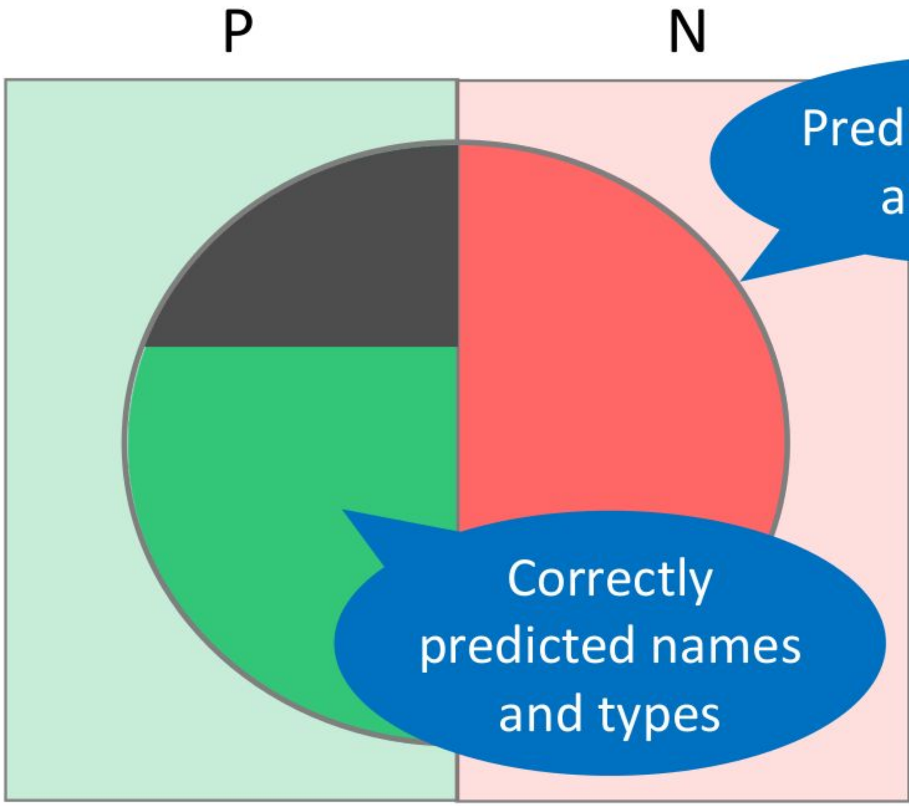


Predicted as semantic registers and memory offsets

Results

Arch	Accuracy
x86	87.1%
x64	88.9%
ARM	90.6%

Name and type prediction accuracy



$$\text{Precision} = \frac{|\mathrm{CP}|}{|\mathrm{PN}|}$$

$$\text{Recall} = \frac{|\mathrm{CP}|}{|\mathrm{P}|}$$

$$\text{F1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Total names and types (P) =

Predicted names and types (PN) =

Correct Predictions (CP) =

Evaluation of name and type prediction

Arch		Precision	Recall	F1
x86	Name	62.6	62.5	62.5
	Type	63.7	63.7	63.7
	Overall	63.1	63.1	63.1
x64	Name	63.5	63.1	63.3
	Type	74.1	73.4	73.8
	Overall	68.8	68.3	68.6
ARM	Name	61.6	61.3	61.5
	Type	66.8	68.0	67.4
	Overall	64.2	64.7	64.5

Malware inspection

Inspected 35 x86 malware from VirusShare

Manipulating DNS settings

```
int sub_80534BA() {
  ...
  if ( dword_8063320 <= 0 ) {
    v1 = ("/etc/resolv.conf", 'r');
    if (v1 || (v1 =
       sub_8053B1("resolv.conf"))){
       ...
       ...
    }
}
```

DE BIN

```
int rfc1035_init_resolv() {
  ...
  if ( num_entries <= 0 ) {
    v0 = ("/etc/resolv.conf", 'r');
    if (v0 || (v1 =
        fopen64("resolv.conf"))){
      // code to read and
      // manipulate DNS settings
    }
}
```

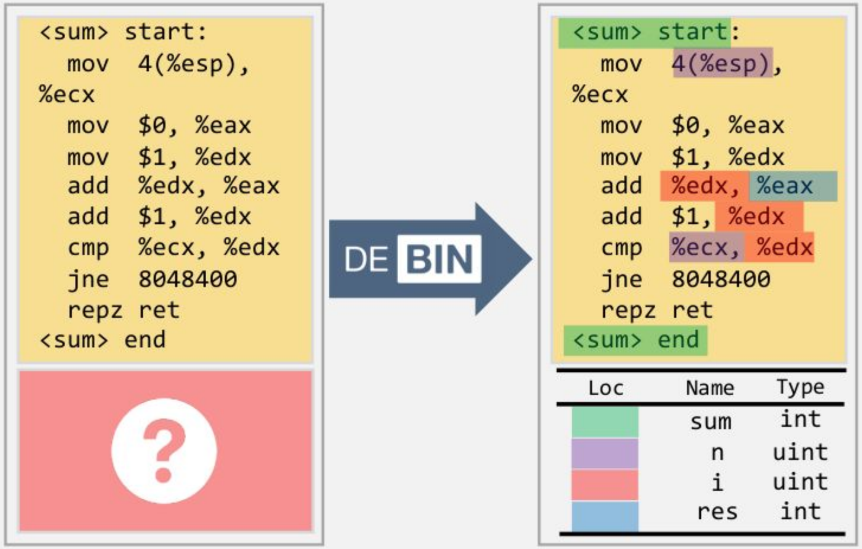
Leakage of sensitive data

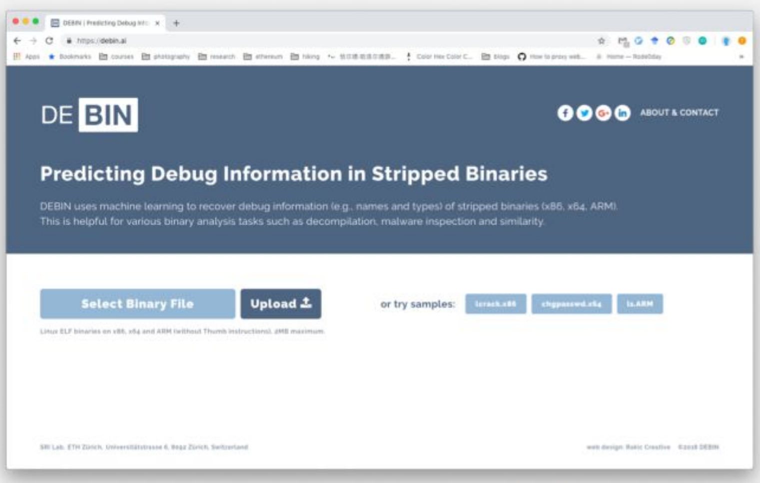
```
If (sub_806d9f0(args) >= 0) {
  ...
  sub_80522B0(args);
  ...
}
```

DE BIN

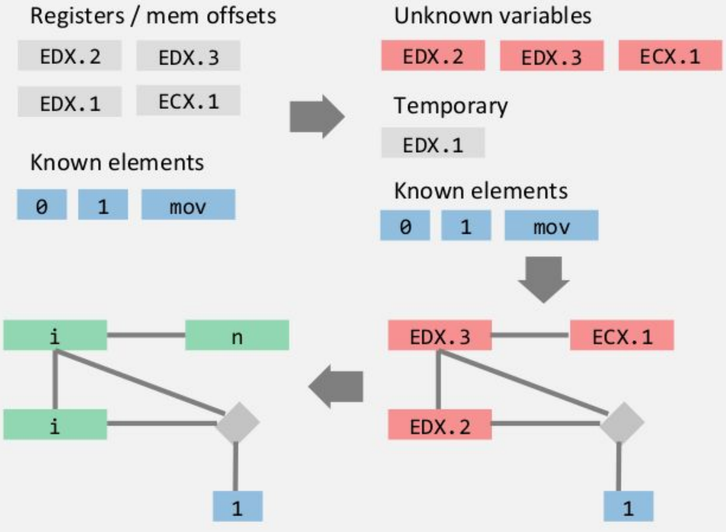
```
If (setsockopt(args) >= 0) {
  ...
  sendto(args);
  ...
}
```

Summary

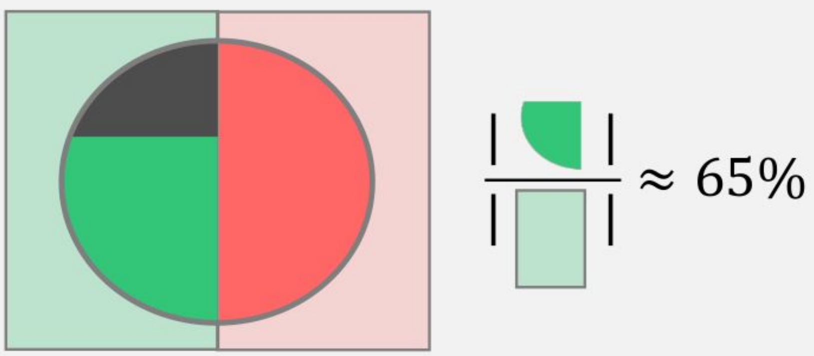




Try online: https://debin.ai



Two-stage prediction process



High precision and accuracy

How can we improve?

Learning To Represent Programs with Graphs

Abhishek Shah

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Problem

Neural Networks have understood:

- Images
- Speech
- Language
- *Source Code* **?**

Problem

```
float getHeight { return this.width; }
```

Question: what’s the bug?

Problem

Do what I **want**, not what I **wrote**

```
float getHeight { return this.width; }
```

Question: what's the bug?

Solution - Learning from “Big Code”

How to feed programs into Neural Networks?

- Sequence of Tokens (Hindle et al., 2012)
- Parse Tree (Bielik et al., 2016)

Key Insight:

- Expose semantics to NN via a Graph
  - Avoid shallow, textual structure by using data flow and type information

Outline

- Primer on Graph Neural Networks
- Converting Programs to Graphs
- Learning Representations with Graph NNs
- Downstream Tasks
- Evaluation

Primer on Graph NNs

- Why use Graphs?
  - Graphs describe a system and the complex dependencies within them
- Use Cases
  - Node Classification → is a node malicious?
  - Link Detection → are these two transactions linked in the blockchain?

Primer on Graph NNs

- Modern DL Techniques
  - CNNs → *fixed-size* images with *spatial locality*
  - RNNs → *ordered* sequences

Primer on Graph NNs

- Modern DL Techniques
  - CNNs → ~~*fixed-size*~~ images with ~~*spatial locality*~~
  - RNNs → ~~*ordered*~~ sequences
- Properties of Graphs
  - No obvious ordering
  - Not fixed sizes
  - Non-obvious or non-existent spatial locality

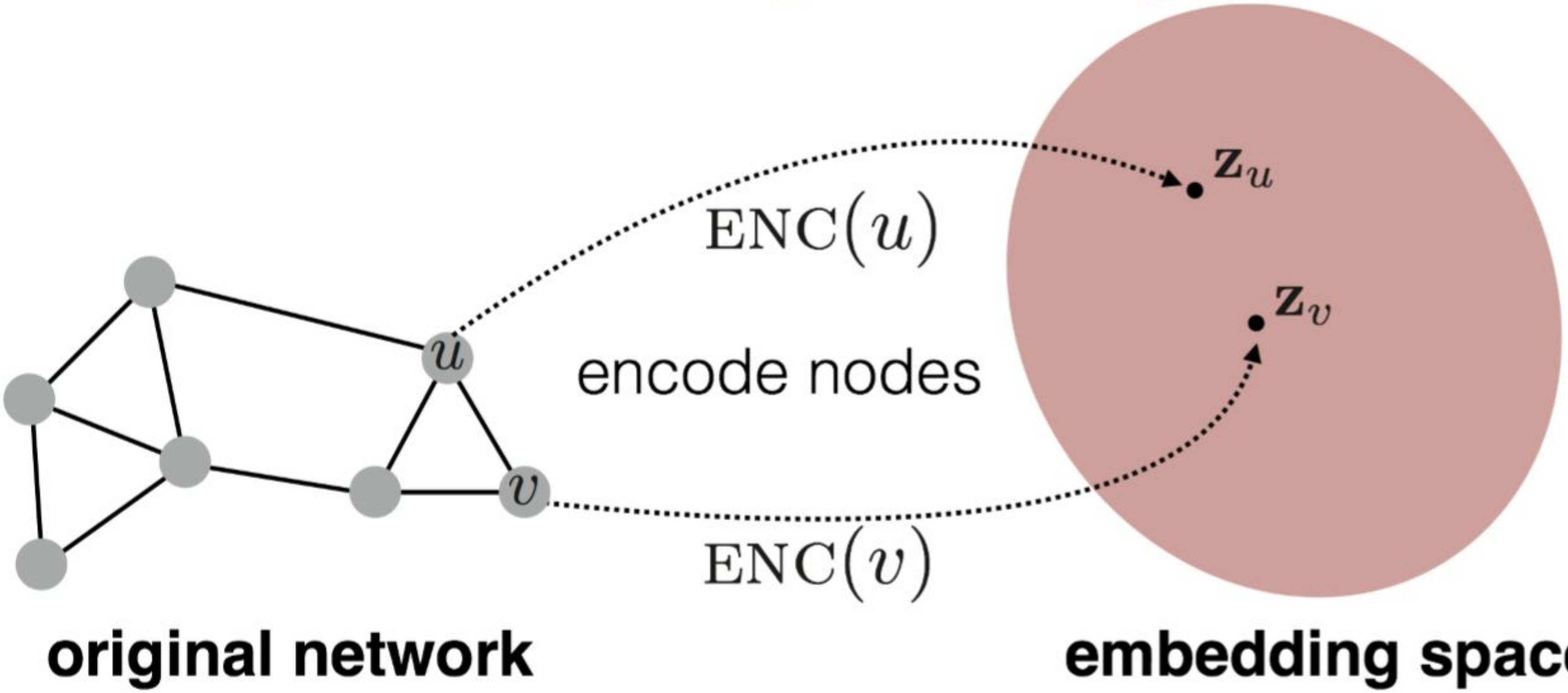
Primer on Graph NNs

- Building a Graph NN (focus on embedding)
  - Need an encoder
    - Such that similarity in original graph is preserved in embedded space

Primer on Graph NNs

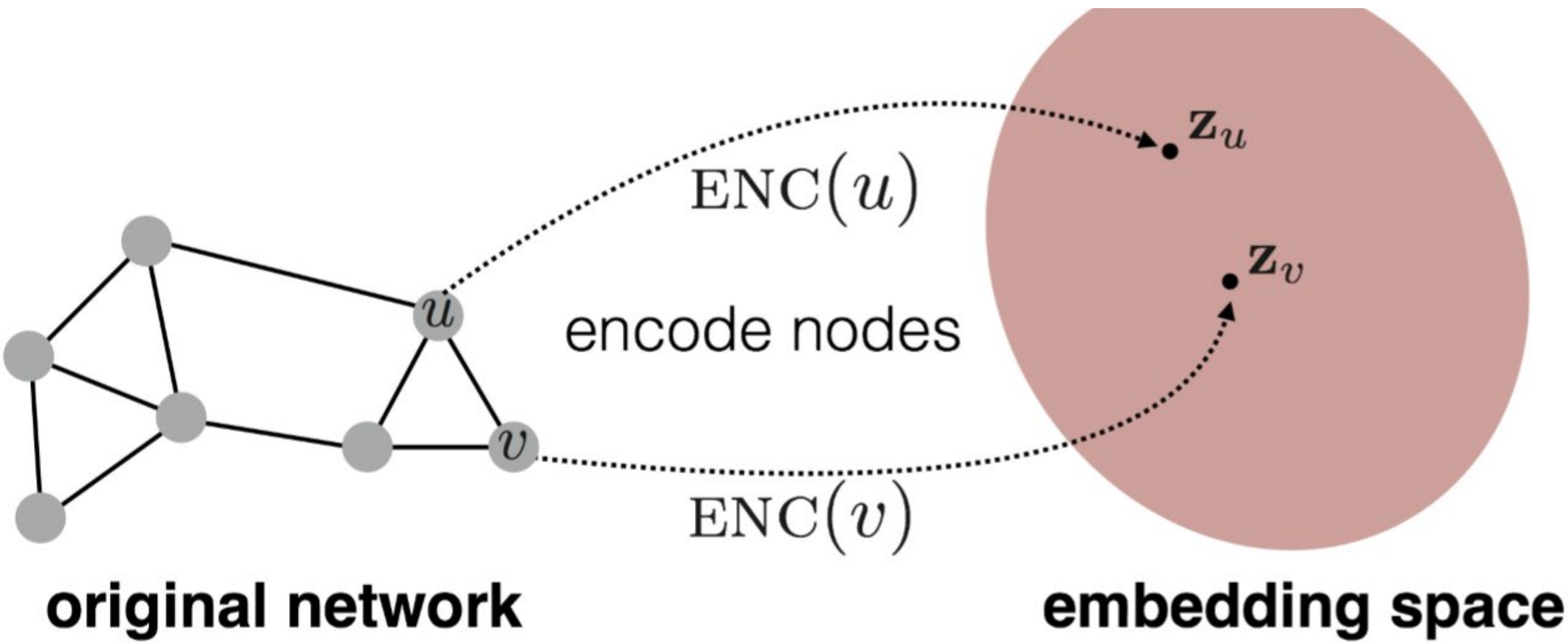
- Building a Graph NN (focus on embedding)
  - Need an encoder
    - Such that similarity in original graph is preserved in embedded space
  - Need a similarity metric
  - Learning → minimizing the distance between similar nodes

Primer on Graph NNs



Primer on Graph NNs

$$\text{similarity}(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u$$



Primer on Graph NNs

- For now, shallow encoding
    - Each node has a unique vector ("embedding-lookup")

Primer on Graph NNs

- For now, shallow encoding
  - Each node has a unique vector ("embedding-lookup")
- Similarity
  - Connected? or Share Neighbors?

Primer on Graph NNs

- For now, shallow encoding
  - Each node has a unique vector ("embedding-lookup")
- Similarity
  - Connected? or Share Neighbors?
  - One Idea: dot products between node embeddings ~ edge existence

Primer on Graph NNs

- For now, shallow encoding
  - Each node has a unique vector ("embedding-lookup")
- Similarity
  - Connected? or Share Neighbors?
  - One Idea: dot products between node embeddings ~ edge existence
  - Adjacency Matrix defines ground truth for edge existence
  - Take the difference between the two

Primer on Graph NNs

- Similarity

$$\boxed{\mathcal{L}} = \quad \| \boxed{\mathbf{z}_u^\top \mathbf{z}_v} - \boxed{\mathbf{A}}_{u,v} \|^2$$

loss (what w
want to minim

ll

s

embedding
similarity

(weighted)
adjacency matrix
for the graph

Primer on Graph NNs

- Similarity

$$\boxed{\mathcal{L}} = \boxed{\sum_{(u,v)\in V\times V}} \|\boxed{\mathbf{z}_u^\top \mathbf{z}_v} - \boxed{\mathbf{A}}_{u,v}\|^2$$

loss (what we want to minimize)

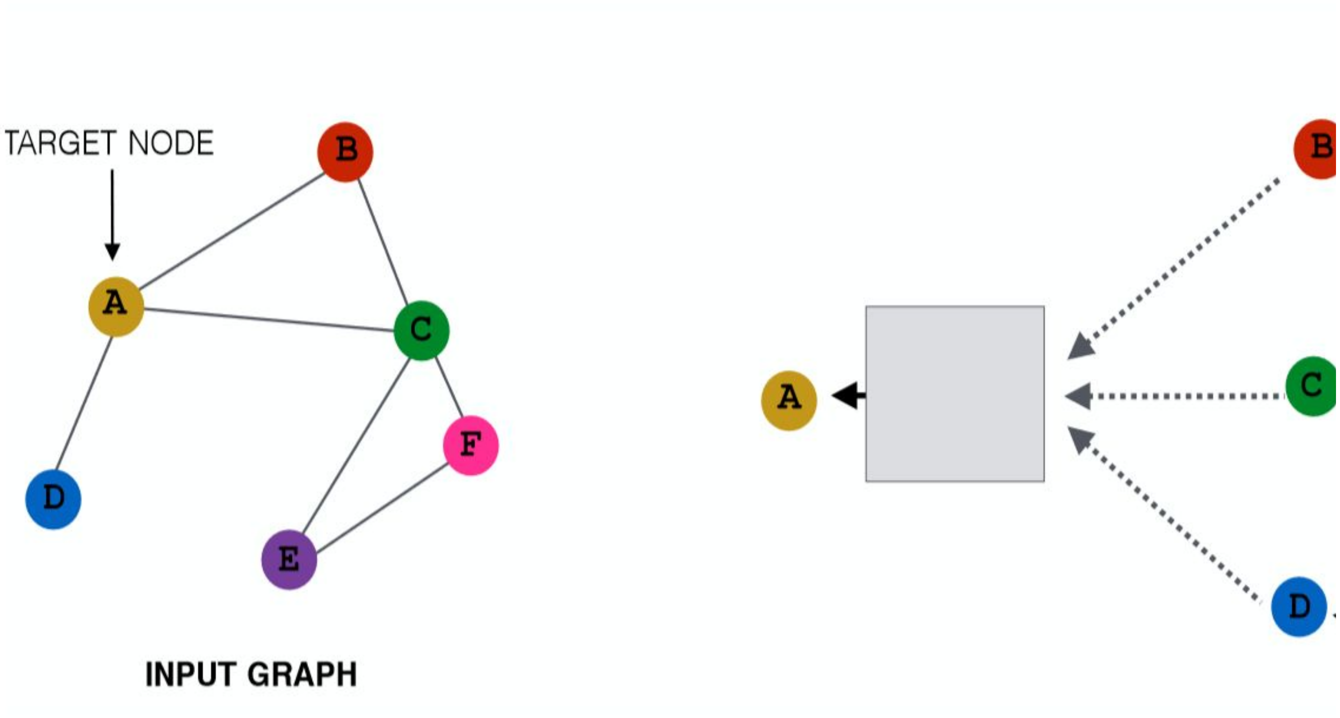
sum over all node pairs

embedding similarity

(weighted) adjacency matrix for the graph

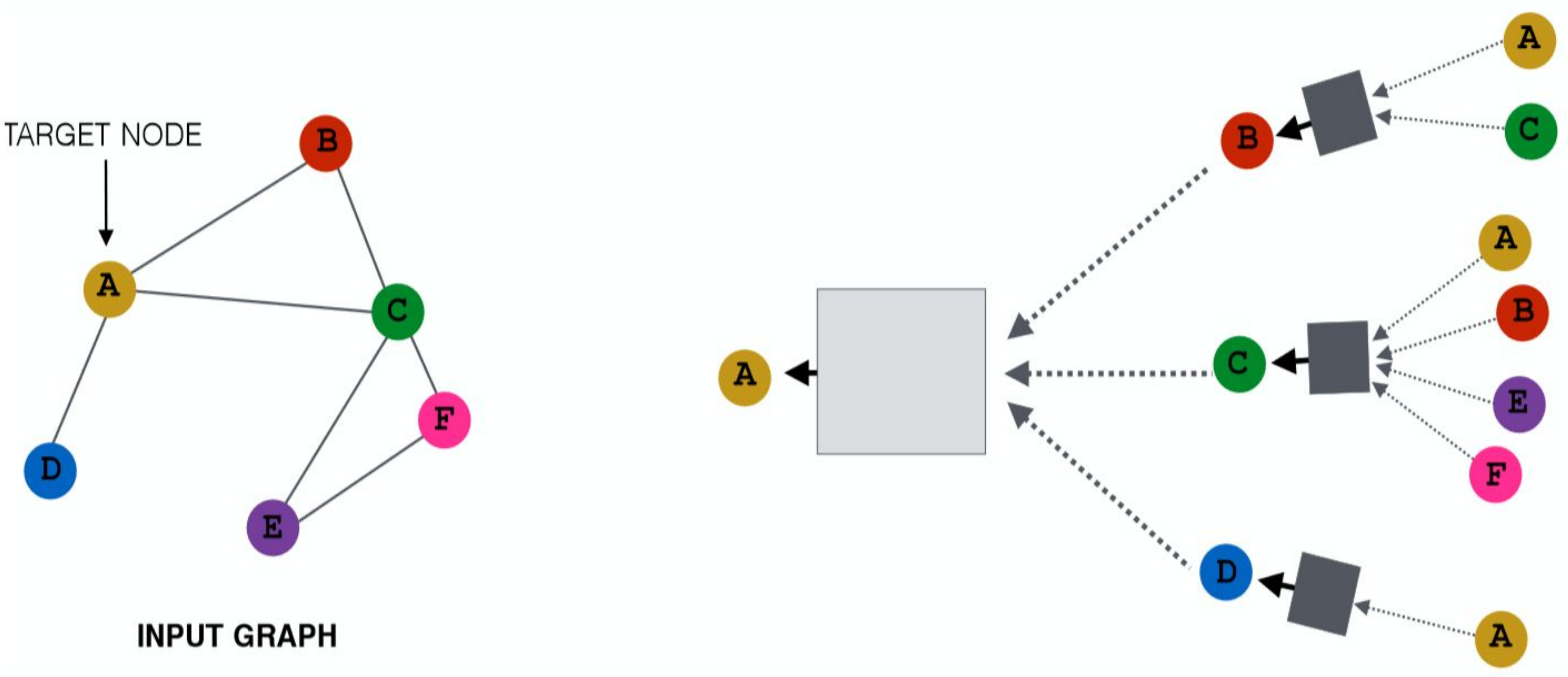
Primer on Graph NNs

- Encoder
  - Main insight: generate node embeddings based on local neighborhoods



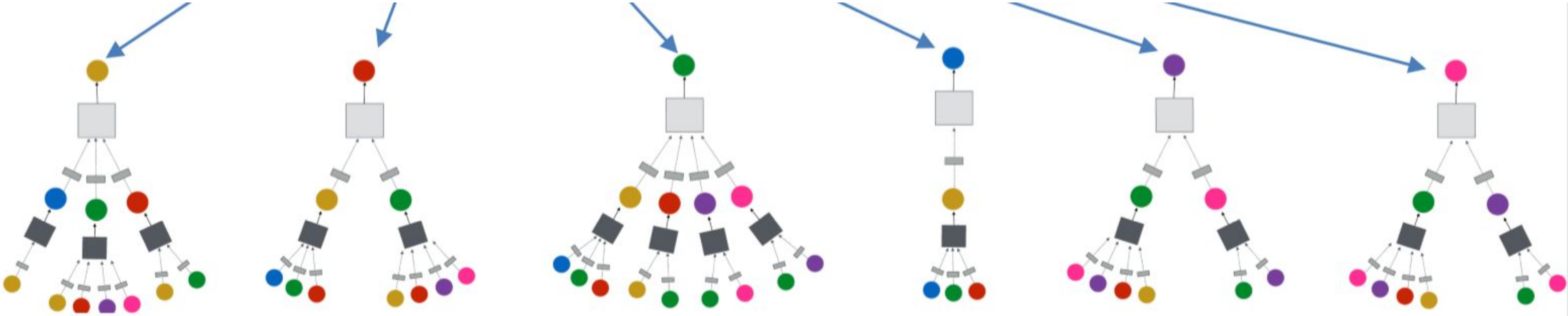
Primer on Graph NNs

- Encoder
  - Main insight: generate node embeddings based on local neighborhoods
  - NNs to aggregate information (msg) per layer



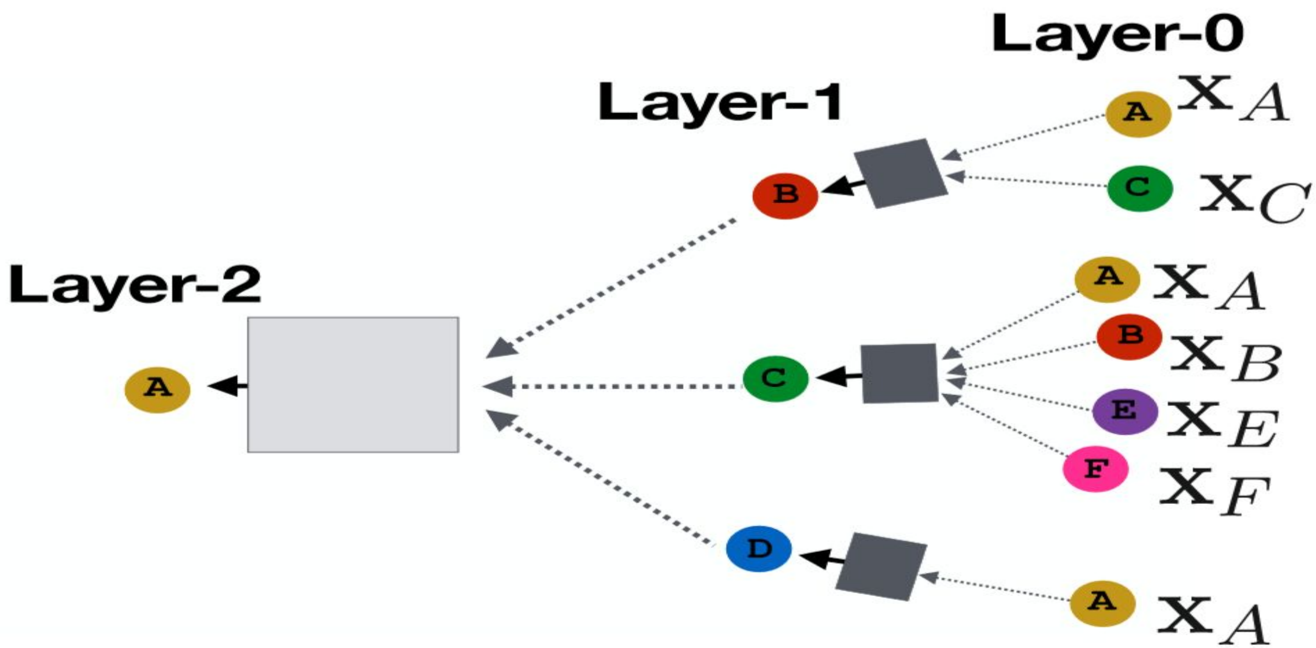
Primer on Graph NNs

- "Deep" Encoder
  - Main insight: generate node embeddings based on local neighborhoods
  - NNs to aggregate information (msg) per layer
  - Each node has unique computation graph



Primer on Graph NNs

- Setup
  - Graph G = (V, A, X)
    - V → Vertex Set
    - A → Adjacency Matrix
    - X → matrix of node features
      - Name, id, relationship status
  - Layer 0 embedding → input feature vector



COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Primer on Graph NNs

- **Basic approach:** Average neighbor messages and apply a neural network.

$$\boxed{\mathbf{h}_v^0 = \mathbf{x}_v}$$

Initial "layer 0" embeddings are equal to node features

Primer on Graph NNs

- **Basic approach:** Average neighbor messages and apply a neural network.

Initial "layer 0" embeddings are equal to node features

$$\boxed{\mathbf{h}_v^0 = \mathbf{x}_v}$$

$$\boxed{\mathbf{h}_v^k} = \boxed{\sigma}\Bigg($$

kth layer embedding of v

non-linearity (e.g., ReLU or tanh)

Primer on Graph NNs

- **Basic approach:** Average neighbor messages and apply a neural network.

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

Initial “layer 0” embeddings are equal to node features

previous layer embedding of v

$$\mathbf{h}_v^k = \sigma$$

kth layer embedding of v

non-linearity (e.g., ReLU or tanh)

average of neighbor’s previous layer embeddings

Primer on Graph NNs

- **Basic approach:** Average neighbor messages and apply a neural network.

Initial "layer 0" embeddings are equal to node features

$$\boxed{\mathbf{h}_v^0 = \mathbf{x}_v}$$

previous layer embedding of *v*

$$\boxed{\mathbf{h}_v^k} = \boxed{\sigma}\left(\mathbf{W}_k \boxed{\sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}} + \mathbf{B}_k \boxed{\mathbf{h}_v^{k-1}}\right), \quad \forall k > 0$$

kth layer embedding of *v*

non-linearity (e.g., ReLU or tanh)

average of neighbor's previous layer embeddings

Primer on Graph NNs

trainable matrices
(i.e., what we learn)

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

$$\mathbf{h}_v^k = \sigma\left(\boxed{\mathbf{W}_k} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \boxed{\mathbf{B}_k} \mathbf{h}_v^{k-1}\right), \quad \forall k \in \{1, \ldots, K\}$$

Primer on Graph NNs

trainable matrices
(i.e., what we learn)

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

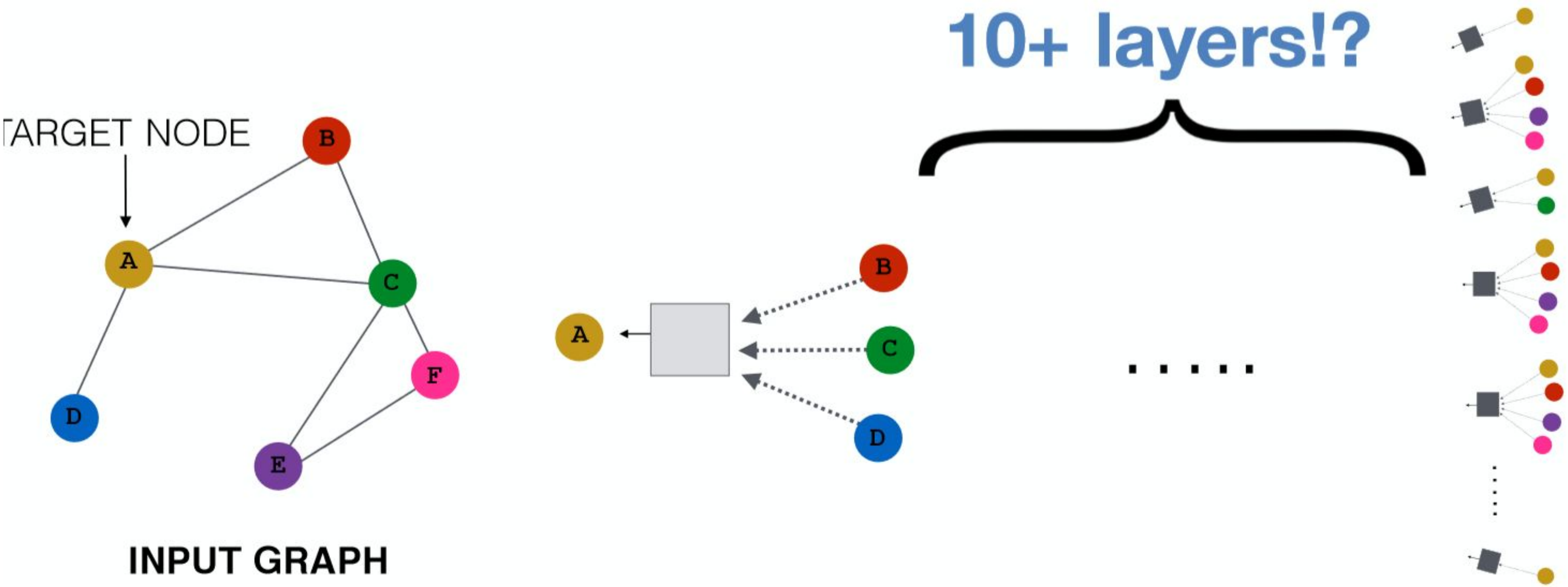
$$\mathbf{h}_v^k = \sigma\left(\boxed{\mathbf{W}_k} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \boxed{\mathbf{B}_k} \mathbf{h}_v^{k-1}\right), \quad \forall k \in \{1, \ldots, K\}$$

$$\boxed{\mathbf{z}_v = \mathbf{h}_v^K}$$

- After K-layers of neighborhood aggregation, we get output embeddings for each node.

Primer on Graph NNs

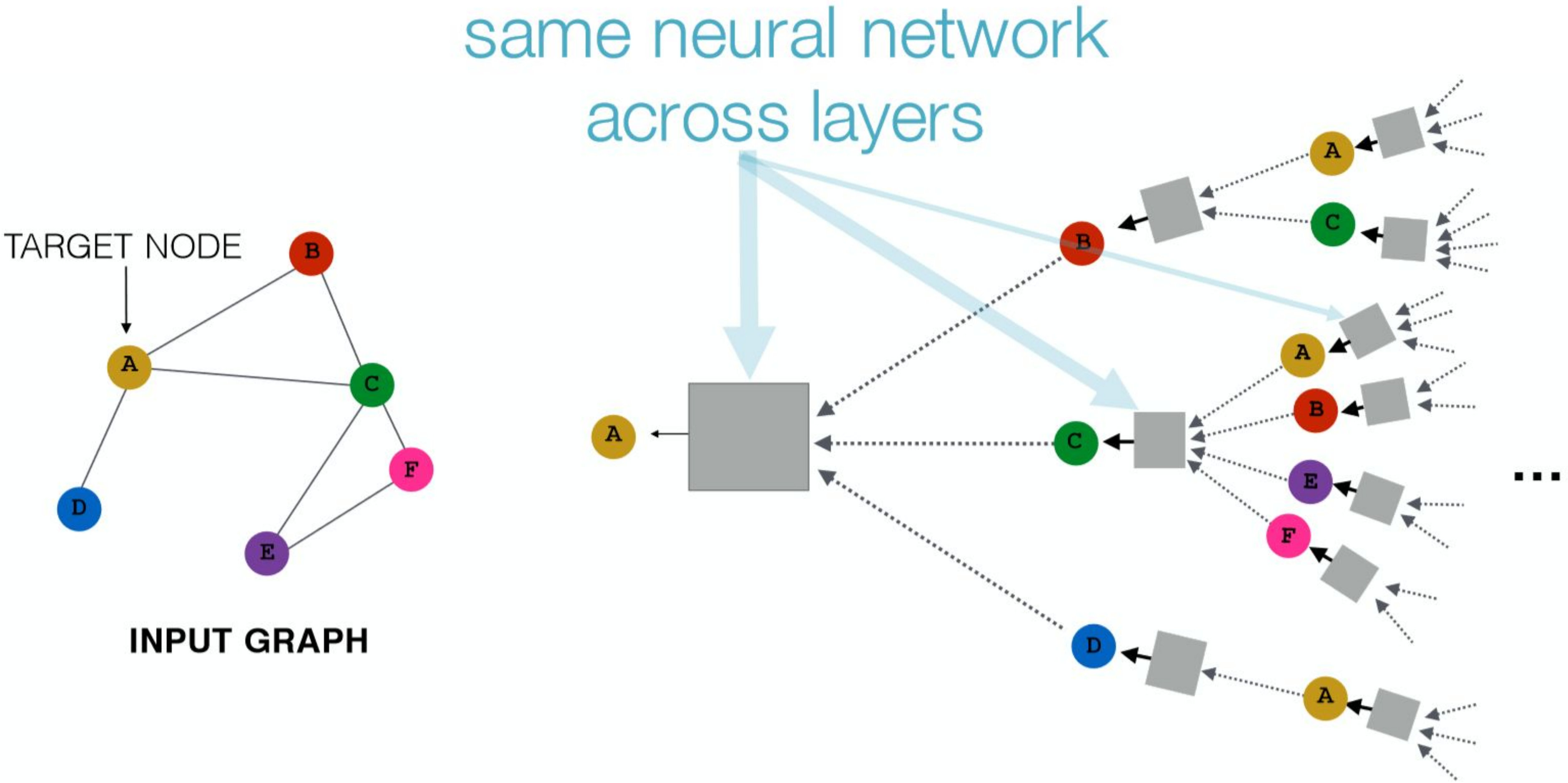
- What if we want to go deeper?
  - Overfitting from parameters



COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

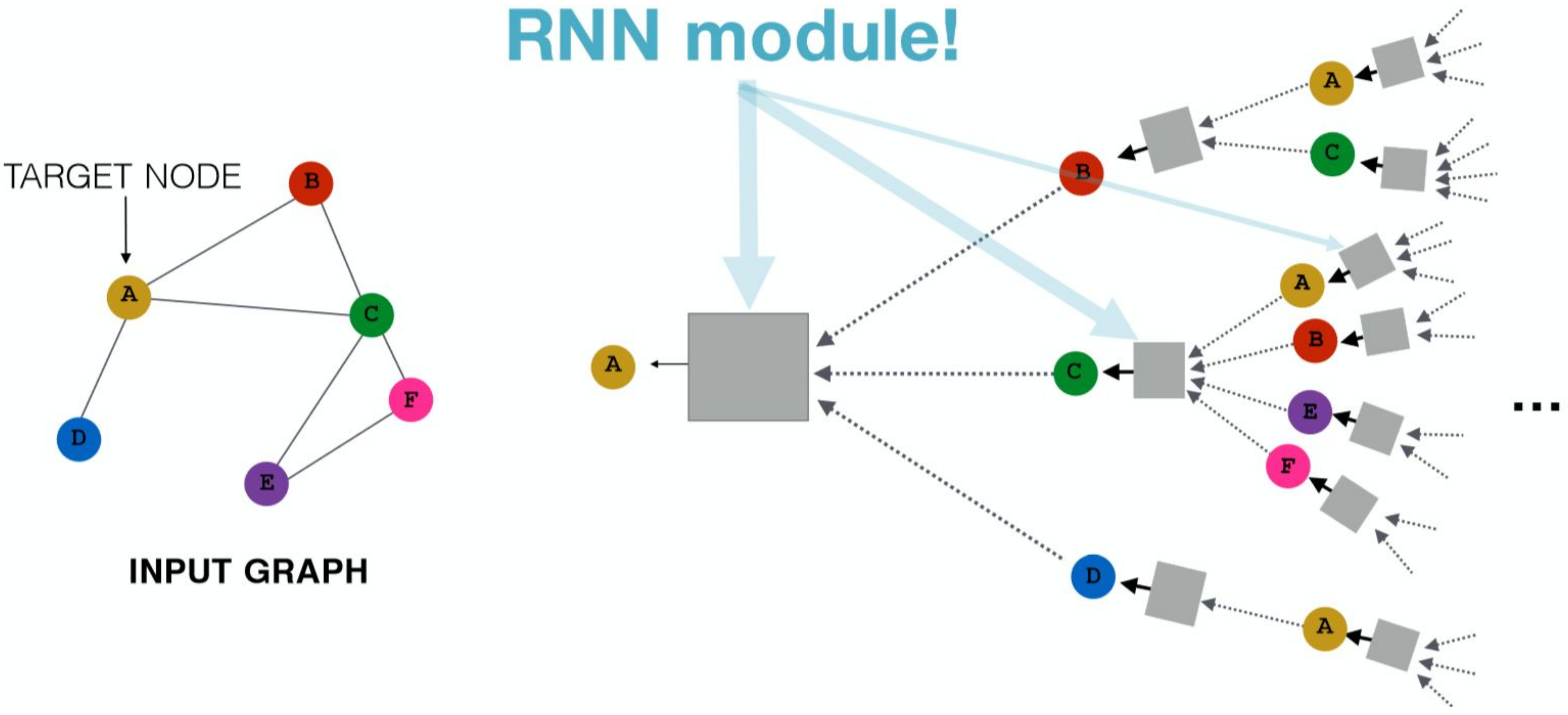
Primer on Graph NNs

- **Idea 1:** Parameter sharing across layers.



Primer on Graph NNs

- **Idea 2:** Recurrent state update.



Gated Graph NN

Intuition: Neighborhood aggregation with RNN state update.

1. Get "message" from neighbors at step k:

$$\mathbf{m}_v^k = \boxed{\mathbf{W} \sum_{u \in N(v)} \mathbf{h}_u^{k-1}}$$

aggregation function does not depend on k

Gated Graph NN

Intuition: Neighborhood aggregation with RNN state update.

1. Get "message" from neighbors at step k:

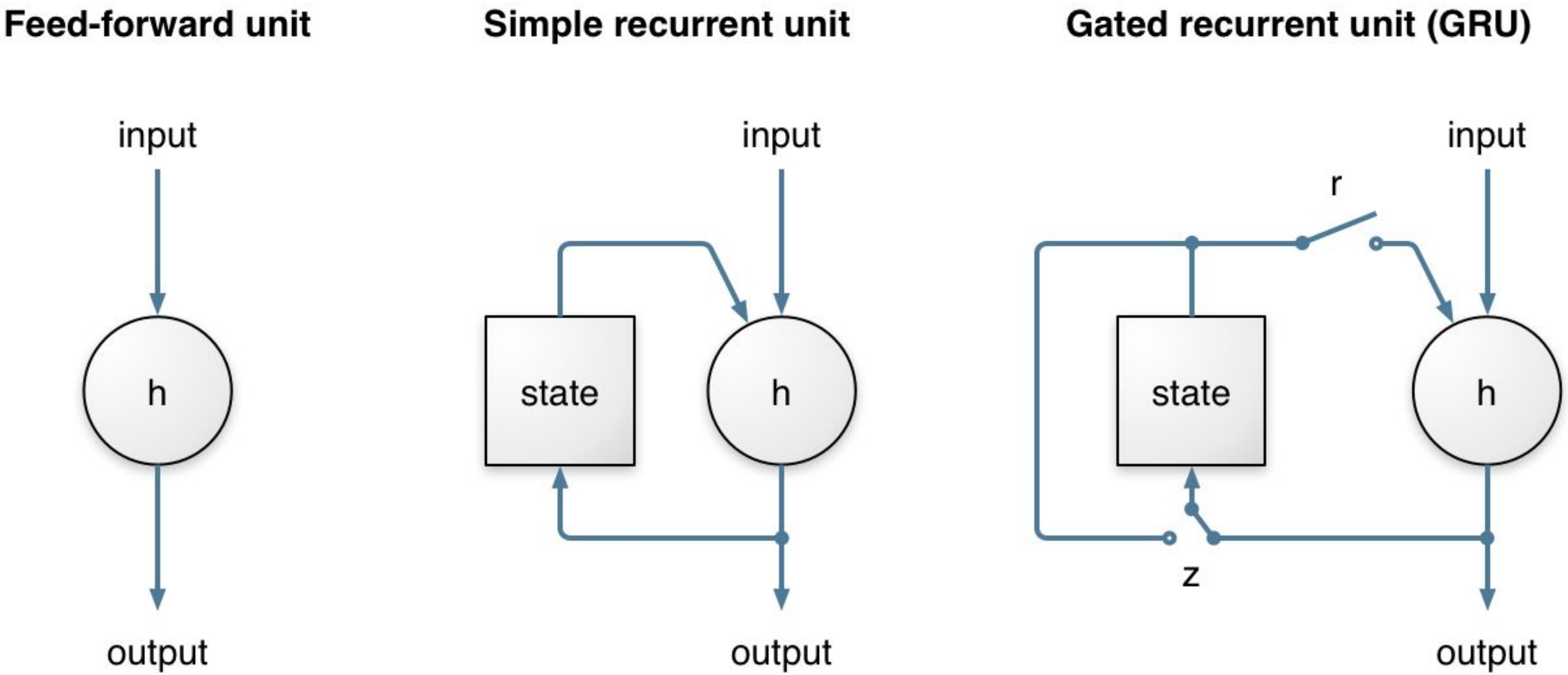
$$\mathbf{m}_v^k = \boxed{\mathbf{W} \sum_{u \in N(v)} \mathbf{h}_u^{k-1}}$$

aggregation function does not depend on k

2. Update node "state" using Gated Recurrent Unit (GRU). New node state depends on the old state and the message from neighbors:

$$\mathbf{h}_v^k = \mathrm{GRU}(\mathbf{h}_v^{k-1}, \mathbf{m}_v^k)$$

Gated Graph NN



Outline

- Primer on Graph Neural Networks
- Converting Programs to Graphs
- Learning with Graph NNs
- Downstream Tasks
- Evaluation

Converting Programs to Graphs

Key Insight:

- Expose semantics to NN via a Graph
  - Avoid shallow, textual structure by using data flow and type information

Published as a conference paper at ICLR 2018

LEARNING TO REPRESENT PROGRAMS WITH GRAPHS

Miltiadis Allamanis
Microsoft Research
Cambridge, UK
miallama@microsoft.com

Marc Brockschmidt
Microsoft Research
Cambridge, UK
mabrocks@microsoft.com

Mahmoud Khademi*
Simon Fraser University
Burnaby, BC, Canada
mkhademi@sfu.ca

Converting Programs to Graphs

Graph: (V, E, X)

- V (AST nodes)
  - Grammar-Rule-Named Nonterminals
  - Named Program Tokens
- E
  - Syntactic
  - Semantic
- Discussion:
  - What are examples of syntactic and semantic edges?

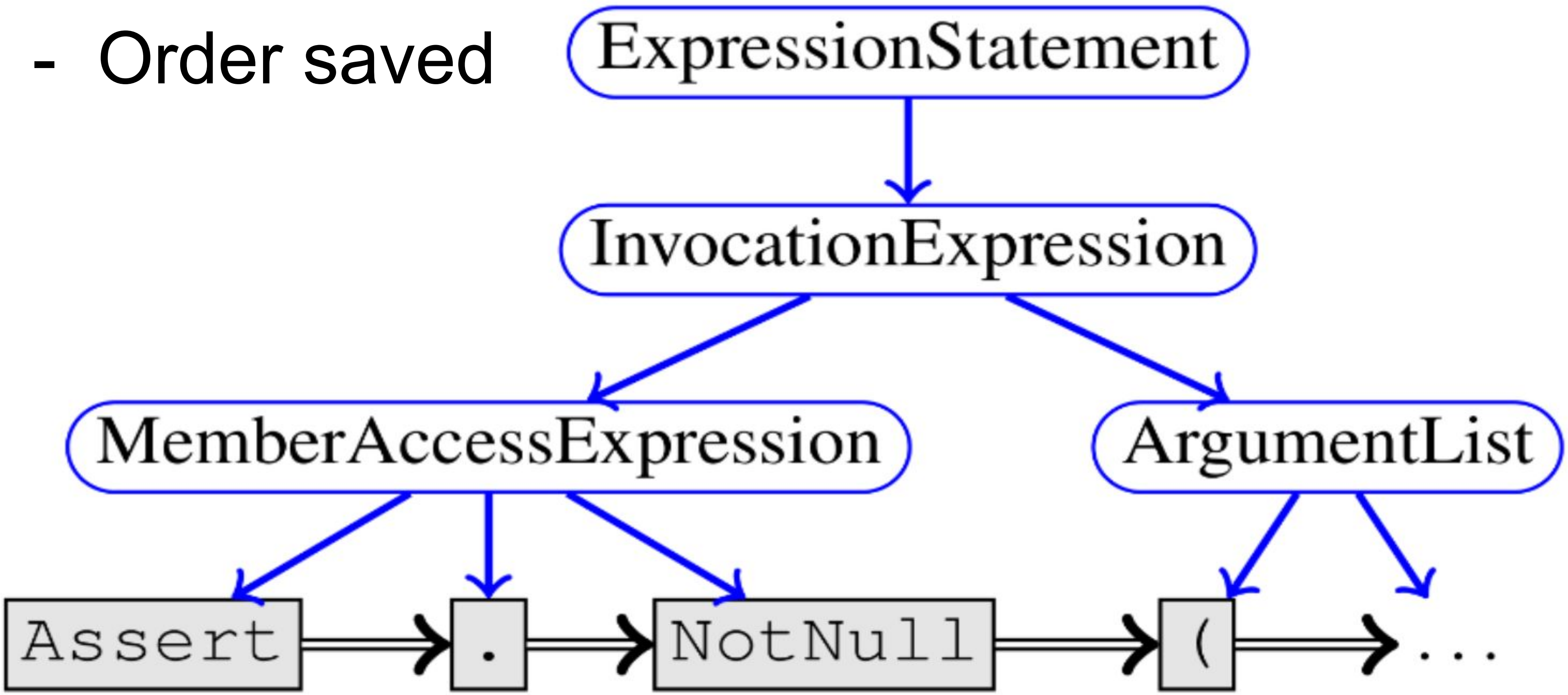
Converting Programs to Graphs

Syntactic Edges

Blue → Children

Black → NextToken

- Order saved



Converting Programs to Graphs

Semantic Edges

```
x, y = Foo();
while (x > 0)
    x = x + y;
```

- Let's focus on y at line 3

Converting Programs to Graphs

Semantic Edges

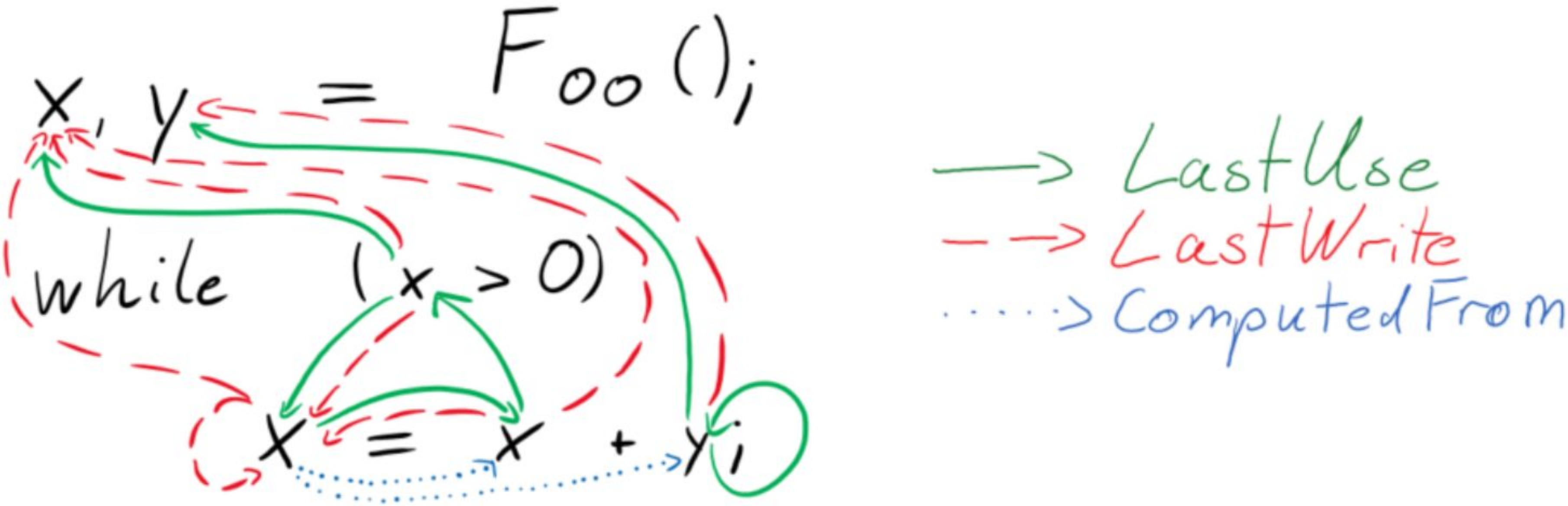
```
x, y = Foo();
while (x > 0)
    x = x + y;
```

- LastUse/Read(y_3) → Line {1, 3}
  - Line 3 due to loop
- LastWrite(y_3) → Line 1

Converting Programs to Graphs

Semantic Edges

```
x, y = Foo();
while (x > 0)
    x = x + y;
```



Converting Programs to Graphs

- Other Edges
  - Can use any other program analysis
    - Points-to analysis
    - Formal Parameter <-->Argument Match
    - Conditional Guards
    - ReturnsTo

Converting Programs to Graphs

Variable Type Information

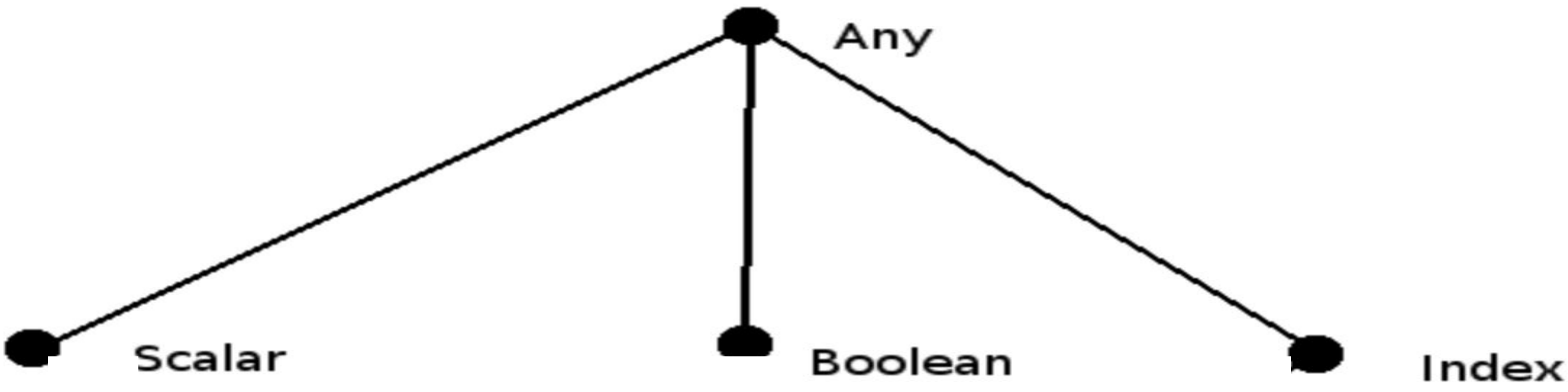
- Map variable type to max of set of supertypes
- List<int> → max({List<int>, List<K>})

Discussion: any flaws?

Converting Programs to Graphs

Variable Type Information

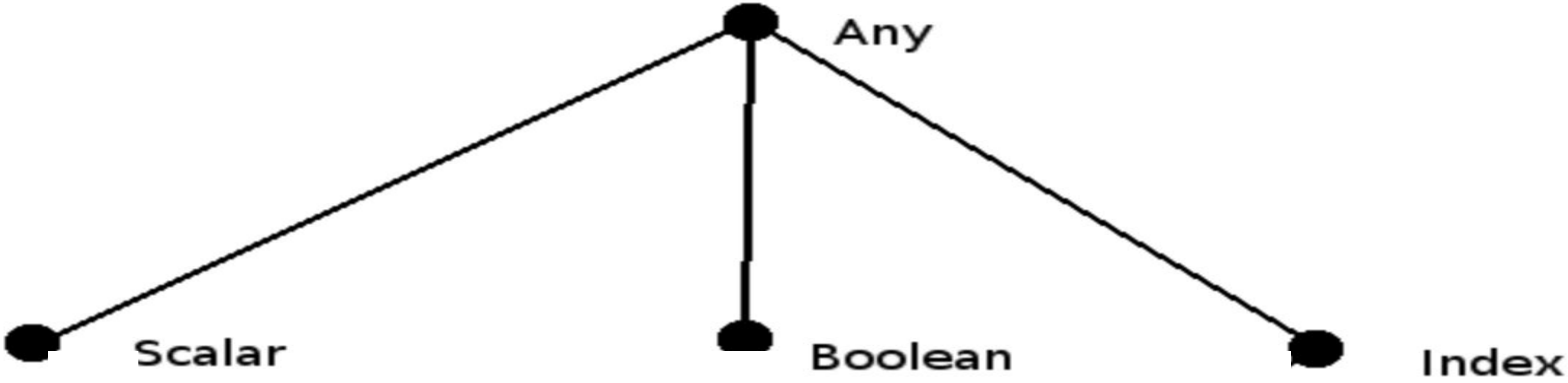
- Map variable type to max of set of supertypes
- Boolean → max({Boolean, Any}) → Any
- Scalar → max({Scalar, Any}) → Any



Converting Programs to Graphs

Variable Type Information

- Use dropout mechanisms: randomly select subset
- Boolean → max({Any}) → Any
- Scalar → max({Scalar}) → Scalar



Learning with Graph NNs

- T = 0 (Initial Node Representation)
  - Concatenate Name with Type string embedding
- Run Gated Graph NN propagation for 8 steps
  - 8 was experimentally determined

Downstream Tasks

- We have an embedding… now what?

Downstream Task 1 - VarNaming

```
private void [redacted] () {
  String vertexShader = "literal_1";
  String fragmentShader = "literal_2";
  shader = new ShaderProgram(vertexShader,
      fragmentShader);
  if(shader.isCompiled() == false)
    throw new IllegalArgumentException(
        "literal_3" + shader.getLog());
}
```

Downstream Task 1 - VarNaming

- Goal: predict correct name at slot t
- Edit Graph
    - Insert new node at slot t (“hole”)

Downstream Task 1 - VarNaming

- Goal: predict correct name at slot t
- Edit Graph
  - Insert new node at slot t (“hole”)
  - Run Gated Graph NN for 8 steps
  - Feed representation into trained GRU to predict name as a sequence

Downstream Task 2 - VarMisuse

- Found several real-world bugs

```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

Downstream Task 2 - VarMisuse

- Goal: predict correct token at slot t
  - Only type-correct tokens allowed at slot t
- Edit Graph
  - Insert new node at slot (“hole”)

Downstream Task 2 - VarMisuse

- Goal: predict correct token at slot t
  - Only type-correct tokens allowed at slot t
- Edit Graph
  - Insert new node at slot (“hole”)
    - Connect it without node v-dependent edges → *context (i.e. c(t))*

Downstream Task 2 - VarMisuse

- Goal: predict correct token at slot t
    - Only type-correct tokens allowed at slot t
- Edit Graph
    - Insert new node at slot ("hole")
        - Connect it without node v-dependent edges → *context (i.e. c(t))*
        - Connect it with node v-dependent edges → *usage representation (i.e. u(t, v))*
            - Edges include LastUse and LastWrite
            - Add usage node per type-correct variable

Downstream Task 2 - VarMisuse

- Goal: predict correct token at slot t
  - Only type-correct tokens allowed at slot t
- Edit Graph
  - Insert new node at slot ("hole")
    - Connect it without node v-dependent edges → *context (i.e. c(t))*
    - Connect it with node v-dependent edges → *usage representation (i.e. u(t, v))*
      - Edges include LastUse and LastWrite
      - Add usage node per type-correct variable
  - Run Gated Graph NN for 8 steps
- Correct Variable Usage
  - Node v that maximizes trained W(*c(t)*, *u(t, v)*)

Evaluation

- Dataset
  - 29 C# projects (~3 million lines of code)
  - Graphs on average: ~2300 nodes, ~8400 edges
- Baseline
  - VarMisuse (predict variable usage)
    - LOC → 2 layer bidirectional GRU
    - AVGB1RNN → LOC + simple variable usage dataflow

Evaluation

- Dataset
  - 29 C# projects (~3 million lines of code)
  - Graphs on average: ~2300 nodes, ~8400 edges
- Baseline
  - VarMisuse (predict variable usage)
    - LOC → 2 layer bidirectional GRU
    - AVGB1RNN → LOC + simple variable usage dataflow
  - VarNaming (predict name)
    - AVGLBL → Log-bilinear model (NLP-inspired)
    - AVGB1RNN (birectional RNN)

Evaluation

- LOC → captures little information
- AVGLBL/AVGB1RNN → captures some info
- Generalization --> unknown types/vocabulary

Table 1: Evaluation of models. UNSEENPROJTEST refers to projects that have no files in the training data, SEENPROJTEST refers to the test set containing projects that have files in the training set.

	SEENPROJTEST				UNSEENPROJTEST			
	LOC	AVGLBL	AVGBIRNN	GGNN	LOC	AVGLBL	AVGBIRNN	GGNN
VARMISUSE								
Accuracy (%)	15.8	—	73.5	**82.1**	13.8	—	59.7	**68.6**
PR AUC	0.363	—	0.931	**0.963**	0.363	—	0.891	**0.909**

Evaluation

- LOC → captures little information
- AVGLBL/AVGB1RNN → captures some info
- Generalization --> unknown types/vocabulary

Table 1: Evaluation of models. UNSEENPROJTEST refers to projects that have no files in the training data, SEENPROJTEST refers to the test set containing projects that have files in the training set.

	SEENPROJTEST				UNSEENPROJTEST			
	LOC	AVGLBL	AVGBIRNN	GGNN	LOC	AVGLBL	AVGBIRNN	GGNN
VARMISUSE								
Accuracy (%)	15.8	—	73.5	**82.1**	13.8	—	59.7	**68.6**
PR AUC	0.363	—	0.931	**0.963**	0.363	—	0.891	**0.909**
VARNAMING								
Accuracy (%)	—	22.0	25.5	**30.7**	—	15.3	15.9	**19.4**
F1 (%)	—	36.1	42.9	**54.6**	—	22.7	23.4	**30.5**

Evaluation

- Lacking semantic info hurts both
- Lacking syntactic info hurts VarMisuse

Table 2: Ablation study for the GGNN model on SeenProjTest for the two tasks.

Ablation Description	VarMisuse Accuracy (%)	VarNaming Accuracy (%)
Standard Model (reported in Table 1)	82.1	30.7
Only NextToken, Child, LastUse, LastWrite edges	79.0	15.4
Only semantic edges (all but NextToken, Child)	74.3	29.7
Only syntax edges (NextToken, Child)	49.6	20.5

Evaluation

- Only syntactic info impacts both
- Only semantic info impacts VarMisuse
- Node initial labeling impacts VarNaming

Table 2: Ablation study for the GGNN model on SEENPROJTEST for the two tasks.

Ablation Description	VARMISUSE Accuracy (%)	VARNAMING Accuracy (%)
Standard Model (reported in Table 1)	82.1	30.7
Only NextToken, Child, LastUse, LastWrite edges	79.0	15.4
Only semantic edges (all but NextToken, Child)	74.3	29.7
Only syntax edges (NextToken, Child)	49.6	20.5
Node Labels: Tokens instead of subtokens	82.1	16.8
Node Labels: Disabled	80.0	14.7

Contributions

- VarMisuse tasks and its practicality
- Learning Program Representations over Graphs

Questions/Discussion

- References
  - http://snap.stanford.edu/proj/embeddings-www/