Learning Program Dependencies with ML

Dongdong She
Kexin Pei
Abhisheek Shah
input_byte = read("foo.txt")
if input_byte[0] == '1':
    // divide by zero error
else:
    // exit gracefully
- 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
- 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
Problem

Symbolic execution has many limitations.

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

\[
\text{Constraints} = S_1 \land N_1 \land N_2 \land S_2
\]
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

```c
1 int main (...) {
2     if (strlen(filename) > 1 && filename[0]==‘-’) {
3         exit(1)
4     }
5     copy_data(...);
6 }
7 void copy_data(..., int *file ...) {
8     static double data[4096], value;
9     read_double_value(file, ...);
10    value = fabs(data[0]);
11    for(i=0; i<4096; i++)
12        if(file[i] == 0.0) count++;
13        data[1] /= (value+count-3);    // CVP: Divide-by-zero
14    ...
15 }
```
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

```c
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
    ...
}
```
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.

```c
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
1. Reachability constraints:
   \( \text{strlen(filename)} \leq 1 \)
   \( \lor \text{filename} \neq '-' \)
   \( \land \)

   \( N: \text{infile} \rightarrow (\text{value}, \text{count}) \)

2. Vulnerability condition:
   \( \text{value} + \text{count} - 3 = 0 \)

```c
1 int main (...) {
2     if (strlen(filename)>1 && filename[0]=='\'-')
3         exit(1)
4     copy_data(...);
5     ...
6 }
7 void copy_data(..., int *file,...) {
8     static double data[4096], value;
9     read_double_value(file, ...);
10    value = fabs (data [0]);
11    for(i=0; i<4096; i++)
12        if(file[i] == 0.0) count++;
13     data[1] /= (value+count-3); // CVP: Divide-by-zero
14     ...
15 }
```
Constraints

Constraint Solving

1. Reachability constraints:
   \[
   
   \text{strlen(} \text{filename} \text{)} \leq 1 \\
   \lor \text{filename} \neq ' - '
   \]
   
   \[
   \land
   \]

   \[
   N: \text{infile} \rightarrow (\text{value, count})
   \]

2. Vulnerability condition:
   \[
   \text{value + count} - 3 = 0
   \]

Purely symbolic constraints:
No variable shared with neural constraints

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

\[ N: \text{infile} \rightarrow (\text{value}, \text{count}) \land \text{value} + \text{count} - 3 = 0 \]

Symbolic constraint

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

One possible encoding:
\[ L = \text{abs}(\text{value} + \text{count} - 3) \]
### Constraints => Loss

<table>
<thead>
<tr>
<th>Symbolic Constraint</th>
<th>Loss Function ((L))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 ::= a &lt; b )</td>
<td>( L = \max(a - b + \alpha, 0) )</td>
</tr>
<tr>
<td>( S_1 ::= a &gt; b )</td>
<td>( L = \max(b - a + \alpha, 0) )</td>
</tr>
<tr>
<td>( S_1 ::= a \leq b )</td>
<td>( L = \max(a - b, 0) )</td>
</tr>
<tr>
<td>( S_1 ::= a \geq b )</td>
<td>( L = \max(b - a, 0) )</td>
</tr>
<tr>
<td>( S_1 ::= a = b )</td>
<td>( L = \text{abs}(a - b) )</td>
</tr>
<tr>
<td>( S_1 ::= a \neq b )</td>
<td>( L = \max(-1, -\text{abs}(a - b + \beta)) )</td>
</tr>
<tr>
<td>( S_1 \land S_2 )</td>
<td>( L = L_{S_1} + L_{S_2} )</td>
</tr>
<tr>
<td>( S_1 \lor S_2 )</td>
<td>( L = \min(L_{S_1}, L_{S_2}) )</td>
</tr>
</tbody>
</table>
Solving Mixed Constraints via Gradient Descent

Gradient: $\nabla_{file} L$

file $\rightarrow$ count, value

<table>
<thead>
<tr>
<th>count</th>
<th>value</th>
<th>loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>20</td>
<td>274</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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

# of bottlenecks: 61
- 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 25% higher than vanilla 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
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

Hex-rays

```c
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

Descriptive names for functions and variables
Stripped Binaries

Assembly

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

Debug symbols

Non-descriptive names

Can we recover the debug symbols?

Hex-rays
Challenges

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

```
<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
```

Stores the value of a semantic variable

Computes $1 + 2 + \ldots + n$

Stores intermediate (non-semantic) value
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

Assembly
<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

Debug information

Assembly
<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

Debug information

<table>
<thead>
<tr>
<th>Location</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>res</td>
<td>int</td>
</tr>
</tbody>
</table>
Design Choice

How will you do this?

Assembly

<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

Debug information

Assembly

<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

Debug information

<table>
<thead>
<tr>
<th>Location</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>res</td>
<td>int</td>
</tr>
</tbody>
</table>
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?

**Learning phase**

- Binary with debug symbols
- Variable recovery model
- Names/types model

**Prediction phase**

- Assembly:
  ```assembly
  start:
  mov 4(%esp), %ecx
  mov $0, %eax
  mov $1, %edx
  add %edx, %eax
  ```

- Debug symbols

- Assembly:
  ```assembly
  start: sum int
  4(%esp) n uint
  $eax res int
  $edx i uint
  ```
Step 1: Recovering Variables

Binaries with debug symbols

>8K binaries

>10K distinct features

>2M vectors

Extracted features

Binary feature vectors

Ensemble of trees

100 decision trees

Feature templates

plus[Reg][Val]
inst[Op][Reg]
dep[Reg][Reg]
...
Step 1: Recovering Variables

Variable recovery

Register

%edx.2

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

::

Features

0010010101010001

Feature vector \( \nu \)

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

\( \text{sem} \)

(DeBIN will predict name and type)

\( \text{tmp} \)

(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

Known elements
Unknown elements
Binary features
Factors
## Pairwise Feature functions

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

Factors:
- All nodes that appear in the same $\phi$ expression of BAP-IR
- Function node of a call and its arguments
- Elements that are accessed in the same statement
Learning to predict

Binary features and factors

Dependency graphs

Actual graphs have >1K nodes

Name 1  Name 2  Weight
f1    i     n    0.4
f2    p     s    0.5
f3    a     b    0.2
f4    i     i    0.3
f5    i     j    0.6
f6    p     p    0.4

3-factor  Weight
1     i     i     0.4
1     j     i     0.2
1     p     p     0.1

4-factor  Weight
1     i     i     k     0.3
1     j     i     a     0.5
1     p     p     v     0.2

Find weights that maximize $P(\vec{U} = \vec{u} | \vec{K} = \vec{k}_i)$ for all training samples $(\vec{u}_i, \vec{k}_i)$

> 8,000 binaries

Binaries with debug symbols

(f_{unary}, Op, Var)
(f_{var-dep}, Var_1, Var_2)
...

Feature templates

23 templates

Static analysis

Train model
End-to-end recovery of debug information

```assembly
<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

**Semantic variables**
- EDX.2
- EDX.3
- ECX.1

**Temporary**
- EDX.1

**Known elements**
- 0
- 1
- mov

```
```

**Graph representation**
- EDX.3
  - ECX.1
- EDX.2
  - 1
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

- cond-NE-EDX-ECX

- dep-EDX-EDX

- 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

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

Debug information

<table>
<thead>
<tr>
<th>Loc</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>uint</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>res</td>
<td></td>
<td>int</td>
</tr>
</tbody>
</table>
Implementation

Static analysis: BAP

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

Learning and inference

http://scikit-learn.org

http://nice2predict.org

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

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

Results

<table>
<thead>
<tr>
<th>Arch</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>87.1%</td>
</tr>
<tr>
<td>x64</td>
<td>88.9%</td>
</tr>
<tr>
<td>ARM</td>
<td>90.6%</td>
</tr>
</tbody>
</table>

Predicted as semantic registers and memory offsets
Name and type prediction accuracy

Correctly predicted names and types

Precision = \frac{|CP|}{|PN|} = \frac{\text{Correct Predictions (CP)}}{\text{Predicted names and types (PN)}}

Recall = \frac{|CP|}{|P|} = \frac{\text{Correct Predictions (CP)}}{\text{Total names and types (P)}}

F1 = \frac{2\times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
## Evaluation of name and type prediction

<table>
<thead>
<tr>
<th>Arch</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x86</td>
<td>62.6</td>
<td>62.5</td>
<td>62.5</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>63.7</td>
<td>63.7</td>
</tr>
<tr>
<td></td>
<td>Overall</td>
<td>63.1</td>
<td>63.1</td>
</tr>
<tr>
<td>x64</td>
<td>63.5</td>
<td>63.1</td>
<td>63.3</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>74.1</td>
<td>73.4</td>
</tr>
<tr>
<td></td>
<td>Overall</td>
<td>68.8</td>
<td>68.3</td>
</tr>
<tr>
<td>ARM</td>
<td>61.6</td>
<td>61.3</td>
<td>61.5</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>66.8</td>
<td>68.0</td>
</tr>
<tr>
<td></td>
<td>Overall</td>
<td>64.2</td>
<td>64.7</td>
</tr>
</tbody>
</table>

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK
Malware inspection

Inspected 35 x86 malware from VirusShare

Manipulating DNS settings

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

```c
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

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

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

Predicting Debug Information in Stripped Binaries

DEBIN uses machine learning to recover debug information in stripped binaries (e.g., ADs, AABs). This technique helps with the analysis of binary files such as disassembling, reverse engineering, and similarity.

Select Binary File
Upload
or try samples: android libart libraries

Two-stage prediction process

DE BIN

Try online: https://debin.ai

High precision and accuracy

≈ 65%
How can we improve?
Learning To Represent Programs with Graphs

Abhishek Shah
Problem

Neural Networks have understood:

- Images
- Speech
- Language
- Source Code?
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
- 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?
- 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
- Building a Graph NN (focus on embedding)
  - Need an encoder
    - Such that similarity in original graph is preserved in embedded space
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 $\rightarrow$ minimizing the distance between similar nodes
Primer on Graph NNs

original network

encode nodes

ENC(u)

ENC(v)

embedding space

z_u

z_v
Primer on Graph NNs

\[
similarity(u, v) \approx z_u^\top z_v
\]
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?
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 $\sim$ edge existence
  - Adjacency Matrix defines ground truth for edge existence
  - Take the difference between the two
Primer on Graph NNs

- Similarity

\[ \mathcal{L} = \| Z_u Z_v \|^2 - \| A_{u,v} \|^2 \]

loss (what we want to minimize)

embedding similarity

(weighted) adjacency matrix for the graph
Primer on Graph NNs

- Similarity

\[ \mathcal{L} = \sum_{(u, v) \in V \times V} \| z_u^T z_v \| - A_{u,v} \]

- 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 $G = (V, A, X)$
    - $V \rightarrow$ Vertex Set
    - $A \rightarrow$ Adjacency Matrix
    - $X \rightarrow$ matrix of node features
      - Name, id, relationship status
  - Layer 0 embedding $\rightarrow$ input feature vector
**Primer on Graph NNs**

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

\[ h_v^0 = x_v \]

Initial “layer 0” embeddings are equal to node features.
**Basic approach:** Average neighbor messages and apply a neural network.

Initial "layer 0" embeddings are equal to node features.

\[ h_0^v = x_v \]

kth layer embedding of v

non-linearity (e.g., ReLU or tanh)
**Basic approach:** Average neighbor messages and apply a neural network.

Initial “layer 0” embeddings are equal to node features.

$k$th layer embedding of $v$

Average of neighbor’s previous layer embeddings.

Non-linearity (e.g., ReLU or tanh)
Primer on Graph NNs

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

\[
\begin{align*}
    h_v^0 &= x_v \\
    h_v^k &= \sigma \left( \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|} + B_k h_v^{k-1} \right), \quad \forall k > 0
\end{align*}
\]

- Initial "layer 0" embeddings are equal to node features
- kth layer embedding of \( v \)
- non-linearity (e.g., ReLU or tanh)
- average of neighbor's previous layer embeddings
- previous layer embedding of \( v \)
Primer on Graph NNs

\[ h^0_v = x_v \]

\[ h^k_v = \sigma \left( \sum_{u \in N(v)} \frac{h^{k-1}_u}{|N(v)|} W^k + B^k h^{k-1}_v \right), \ \forall k \in \{1, \ldots, K\} \]

(trainable matrices
(i.e., what we learn)
Primer on Graph NNs

\[ h^0_v = x_v \]

\[ h^k_v = \sigma \left( \sum_{u \in N(v)} \frac{h^{k-1}_u}{|N(v)|} + B_k h^{k-1}_v \right), \quad \forall k \in \{1, \ldots, K\} \]

\[ z_v = h^K_v \]

- 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
Primer on Graph NNs

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

same neural network across layers
Primer on Graph NNs

- **Idea 2:** Recurrent state update.
**Intuition:** Neighborhood aggregation with RNN state update.

1. Get “message” from neighbors at step $k$:

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

aggregation function does not depend on $k$
Gated Graph NN

**Intuition:** Neighborhood aggregation with RNN state update.

1. Get “message” from neighbors at step $k$:
   
   $$m^k_v = W \sum_{u \in N(v)} h^{k-1}_u$$

   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:
   
   $$h^k_v = \text{GRU}(h^{k-1}_v, m^k_v)$$
Gated Graph NN

Feed-forward unit

Simple recurrent unit

Gated recurrent unit (GRU)
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?
Syntactic Edges

Blue → Children
Black → NextToken

- Order saved
Converting Programs to Graphs

Semantic Edges

\[
\begin{align*}
x, y &= \text{Foo}(); \\
\text{while } (x > 0) & \quad x = x + y;
\end{align*}
\]

- Let’s focus on \( y \) at line 3
Converting Programs to Graphs

Semantic Edges

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

Discussion: any flaws?
Variable Type Information
- Map variable type to max of set of supertypes
- Boolean $\rightarrow \max(\{\text{Boolean}, \text{Any}\}) \rightarrow \text{Any}$
- Scalar $\rightarrow \max(\{\text{Scalar}, \text{Any}\}) \rightarrow \text{Any}$
Converting Programs to Graphs

Variable Type Information

- Use dropout mechanisms: randomly select subset
- Boolean $\rightarrow \max(\{\text{Any}\}) \rightarrow \text{Any}$
- Scalar $\rightarrow \max(\{\text{Scalar}\}) \rightarrow \text{Scalar}$
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?
private void DownstreamTask1() {
    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

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

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

Assert.AreEqual("string", first.Properties["Name"].Name);
Assert.IsFalse(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”)

Columbia University
In the City of New York
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 $\rightarrow$ context (i.e. $c(t)$)
    - Connect it with node $v$-dependent edges $\rightarrow$ 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 $\rightarrow$ 2 layer bidirectional GRU
    - AVGB1RNN $\rightarrow$ 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)
- 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.

<table>
<thead>
<tr>
<th>VarMisuse</th>
<th>SeenProjTest</th>
<th>UnseenProjTest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Loc</td>
<td>AvGLBL</td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>15.8</td>
<td>—</td>
</tr>
<tr>
<td>PR AUC</td>
<td>0.363</td>
<td>—</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th></th>
<th>SeenProjTest</th>
<th></th>
<th></th>
<th></th>
<th>UnseenProjTest</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Avglbl</td>
<td>Avgbirnn</td>
<td>Ggnn</td>
<td>LOC</td>
<td>Avglbl</td>
<td>Avgbirnn</td>
<td>Ggnn</td>
</tr>
<tr>
<td>VarMisuse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>15.8</td>
<td>—</td>
<td>73.5</td>
<td>82.1</td>
<td>13.8</td>
<td>—</td>
<td>59.7</td>
<td>68.6</td>
</tr>
<tr>
<td>PR AUC</td>
<td>0.363</td>
<td>—</td>
<td>0.931</td>
<td>0.963</td>
<td>0.363</td>
<td>—</td>
<td>0.891</td>
<td>0.909</td>
</tr>
<tr>
<td>VarNaming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>22.0</td>
<td>25.5</td>
<td>30.7</td>
<td></td>
<td>15.3</td>
<td>15.9</td>
<td>19.4</td>
<td></td>
</tr>
<tr>
<td>F1 (%)</td>
<td>36.1</td>
<td>42.9</td>
<td>54.6</td>
<td></td>
<td>22.7</td>
<td>23.4</td>
<td>30.5</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Ablation Description</th>
<th>VARMISUSE Accuracy (%)</th>
<th>VARNAMING Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Model (reported in Table 1)</td>
<td>82.1</td>
<td>30.7</td>
</tr>
<tr>
<td>Only NextToken, Child, LastUse, LastWrite edges</td>
<td>79.0</td>
<td>15.4</td>
</tr>
<tr>
<td>Only semantic edges (all but NextToken, Child)</td>
<td>74.3</td>
<td>29.7</td>
</tr>
<tr>
<td>Only syntax edges (NextToken, Child)</td>
<td>49.6</td>
<td>20.5</td>
</tr>
</tbody>
</table>
Evaluation

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

<table>
<thead>
<tr>
<th>Ablation Description</th>
<th>VARMisUSE Accuracy (%)</th>
<th>VARNaming Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Model (reported in Table 1)</td>
<td>82.1</td>
<td>30.7</td>
</tr>
<tr>
<td>Only NextToken, Child, LastUse, LastWrite edges</td>
<td>79.0</td>
<td>15.4</td>
</tr>
<tr>
<td>Only semantic edges (all but NextToken, Child)</td>
<td>74.3</td>
<td>29.7</td>
</tr>
<tr>
<td>Only syntax edges (NextToken, Child)</td>
<td>49.6</td>
<td>20.5</td>
</tr>
<tr>
<td>Node Labels: Tokens instead of subtokens</td>
<td>82.1</td>
<td>16.8</td>
</tr>
<tr>
<td>Node Labels: Disabled</td>
<td>80.0</td>
<td>14.7</td>
</tr>
</tbody>
</table>
Contributions

- VarMisuse tasks and its practicality
- Learning Program Representations over Graphs
- References
  - http://snap.stanford.edu/proj/embeddings-www/