Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection

Author: Xiaojun Xu

Shanghai Jiao Tong University

Presenter: Faizan Ahmad

https://qdata.github.io/deep2Read
Outline

1. Introduction
2. Related Work
3. Gemini
4. Evaluation and Results
5. Takeaways
Binary Code $\rightarrow$ Decompiled assembly code

Code Similarity $\rightarrow$ Comparing two *functions semantically*

Why cross-platform $\rightarrow$ Plethora of platforms these days - differences in compilation
  - Different operating systems
  - Different compilers
  - Different optimization techniques

Why Binary? $\rightarrow$ Source code is seldom available, hence the tendency towards binary analysis
Related Work

Graph Matching

- Pairwise Graph Matching [1] [2]
  - Convert functions into control flow graphs (CFG)
  - Match two graphs using graph matching algorithms
  - **Problems?** Ineffective and computationally very expensive

Figure: Control Flow Graph
Graph Embeddings - **(Genius)** [3]
- Convert each function into a CFG
- Train graph neural networks.
- **Problems?** How to get labeled data for similar codes?

**Figure:** Control Flow Graph
Figure: **Training** Gemini - *Siamese* Graph Neural Network Architecture
Figure: Gemini Testing - Siamese Graph Neural Network Architecture
Gemini
Structure2vec GNN Model

Algorithm 1 Graph embedding generation

1: **Input:** ACFG $g = \langle \mathcal{V}, \mathcal{E}, \mathcal{x} \rangle$
2: Initialize $\mu_v^{(0)} = \overline{0}$, for all $v \in \mathcal{V}$
3: **for** $t = 1$ **to** $T$ **do**
4:     **for** $v \in \mathcal{V}$ **do**
5:         $l_v = \sum_{u \in N(v)} \mu_u^{(t-1)}$
6:         $\mu_v^{(t)} = \tanh(W_1 x_v + \sigma(l_v))$
7:     **end for**
8: **end for**{fixed point equation update}
9: return $\phi(g) := W_2(\sum_{v \in \mathcal{V}} \mu_v^{(T)})$

**Figure:** Gemini uses Structure2vec [3] as the GNN model
Gemini
Node Attributes

(a) Partial control flow graph of dTLS1_process_heartbeat
(a) Partial control flow graph of `dltls1_process_heartbeat`

(b) The corresponding ACFG

```
mov [esp+4Ch+var_40], edi  
mov [esp+4Ch+n], 18h  
mov [esp+4Ch+var_3C], edx  
mov edx, [esi]  
mov [esp+4Ch+dest], 0  
call eax

loc_80C1B2B:  
cmp bp, 1  
jz short loc_80C1B88

xor eax, eax  
cmp bp, 2  
jz short loc_80C1B48

loc_80C1B48:  
cmp ebx, 12h  
movzx edx, byte ptr [edi+3]  
movzx ecx, byte ptr [edi+4]  
jnz short loc_80C1B39

lea eax, [ebx+13h]  
...

mov [esp+4Ch+src], offset aD1_both_c  
mov [esp+4Ch+dest], eax  
mov [esp+4Ch+var_24], eax  
call CRYPTO_malloc  
...

mov [esp+4Ch+dest], ecx; dest  
mov [esp+4Ch+src], edi; src  
mov [esp+4Ch+var_20], ecx  
call _memcpy  
mov ecx, [esp+4Ch+var_20]
```
Node Attributes

- Function $\rightarrow$ basic blocks (node)
- Node features are extracted from basic blocks

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block-level attributes</td>
<td>String Constants</td>
</tr>
<tr>
<td></td>
<td>Numeric Constants</td>
</tr>
<tr>
<td></td>
<td>No. of Transfer Instructions</td>
</tr>
<tr>
<td></td>
<td>No. of Calls</td>
</tr>
<tr>
<td></td>
<td>No. of Instructions</td>
</tr>
<tr>
<td></td>
<td>No. of Arithmetic Instructions</td>
</tr>
<tr>
<td>Inter-block attributes</td>
<td>No. of offspring</td>
</tr>
<tr>
<td></td>
<td>Betweenness</td>
</tr>
</tbody>
</table>

Table 1: Basic-block attributes
Evaluation and Results

Dataset Creation

- Complete source code of OpenSSL
- Compiled with three architectures
  - x86
  - MIPS
  - ARM
- 129,365 control flow graphs

<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Validation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>30,994</td>
<td>3,868</td>
<td>3,973</td>
</tr>
<tr>
<td>MIPS</td>
<td>41,477</td>
<td>5,181</td>
<td>5,209</td>
</tr>
<tr>
<td>ARM</td>
<td>30,892</td>
<td>3,805</td>
<td>3,966</td>
</tr>
<tr>
<td>Total</td>
<td>103,363</td>
<td>12,854</td>
<td>13,148</td>
</tr>
</tbody>
</table>

Figure: CFGs in data set
Evaluation and Results

Comparison Methods

- Bipartite Graph Matching (BGM)
- Genius - Embeddings based on GNNs [3]
  - Labels are created based on graph matching - **not good!**
- Gemini (Uses Structure2Vec [3] as the GNN model)
Does our labeling methodology work for all tasks? NO!
- Vulnerability detection - we want the semantics to match
- Plagiarism detection - we want to syntax to match too

Solution?
- Pretrain on larger data set
- Retrain on a smaller fine grained data set
Results

ROC Curves

Figure 5: ROC curves for different approaches evaluated on the testing similarity dataset.
Results

Ablation Analysis

(d) ROC versus embedding size $p$.

(e) ROC versus ACFG attributes.

(f) ROC versus no. of iterations $T$. 
Figure 8: Visualizing the embeddings of the different functions using t-SNE. Each color indicates one source functions. The legend provides the source function names.
Results
Retraining

- Pretrain on a large data set
- Retrain on a vulnerable code dataset
- Test on a held-out set of vulnerable codes
  - 50 or 100 most similar functions based on code similarity

Results $\rightarrow$ 80% precision as compared with 35% from Genius.
Takeaways

- Graph based approaches for program analysis often work well
- Pretraining before retraining is a nice way around data scarcity
- Again, huge implications for vulnerability analysis
