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

Introduction

- 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

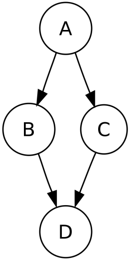


Figure: Control Flow Graph

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

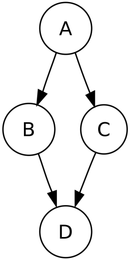


Figure: Control Flow Graph

Gemini

Graph Neural Networks for Binary Code Similarity

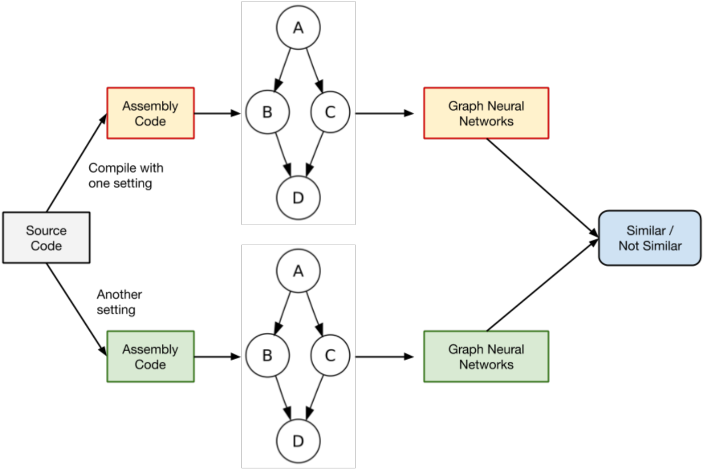


Figure: **Training** Gemini - *Siamese* Graph Neural Network Architecture

Gemini

Graph Neural Networks for Binary Code Similarity

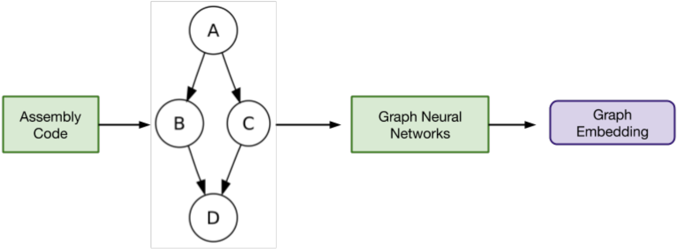


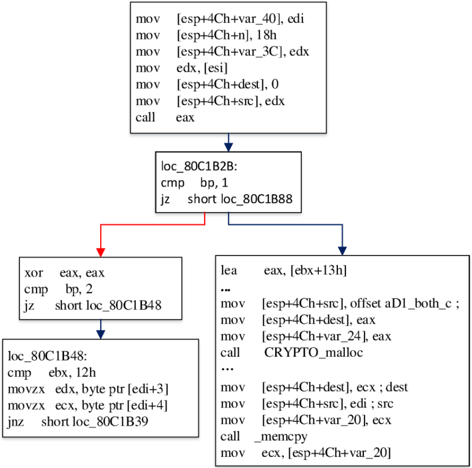
Figure: Gemini **Testing** - *Siamese* Graph Neural Network Architecture

Algorithm 1 Graph embedding generation

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

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

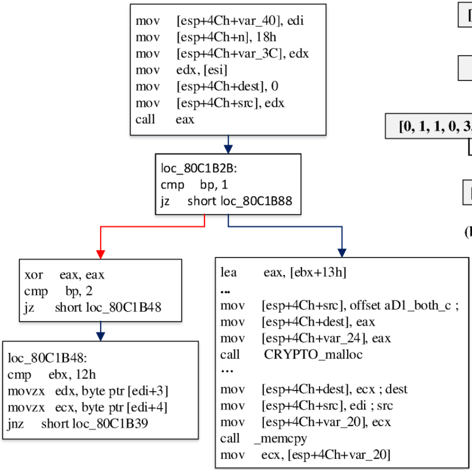
Gemini
Node Attributes



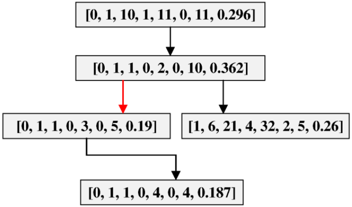
(a) Partial control flow graph of dtls1_process_heartbeat

Gemini

Node Attributes



(a) Partial control flow graph of dtls1_process_heartbeat



(b) The corresponding ACFG

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

Type	Attribute name
Block-level attributes	String Constants
	Numeric Constants
	No. of Transfer Instructions
	No. of Calls
	No. of Instructions
	No. of Arithmetic Instructions
Inter-block attributes	No. of offspring
	Betweenness

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

	Training	Validation	Testing
x86	30,994	3,868	3,973
MIPS	41,477	5,181	5,209
ARM	30,892	3,805	3,966
Total	103,363	12,854	13,148

Figure: CFGs in data set

- 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!

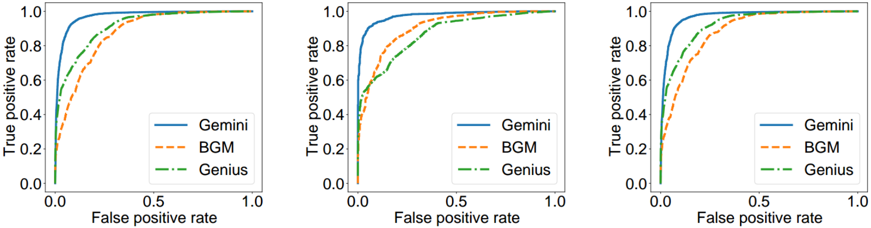
- Vulnerability detection - we want the semantics to match
- Plagiarism detection - we want to syntax to match too

Solution?

- **Pretrain** on larger data set
- **Retrain** on a smaller fine grained data set

Results

ROC Curves

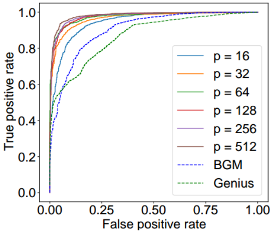


(a) Results on the similarity testing set (b) Results on the large-graph subset (c) Results on the small-graph subset

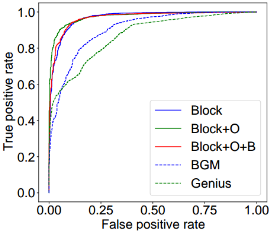
Figure 5: ROC curves for different approaches evaluated on the testing similarity dataset.

Results

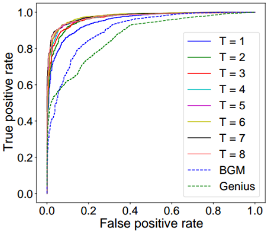
Ablation Analysis



(d) ROC versus embedding size *p*.



(e) ROC versus ACFG attributes.



(f) ROC versus no. of iterations *T*.

Results

Embeddings Visualization

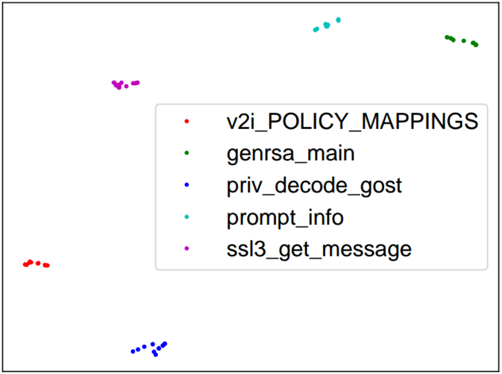


Figure 8: Visualizing the embeddings of the different functions using t-SNE. Each color indicates one source functions. The legend provides the source function names.

- 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 → 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

J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, pp. 709–724, IEEE, 2015.

S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code.," in *NDSS*, 2016.

Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 480–491, ACM, 2016.