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

Xiaojun Xu
Shanghai Jiao Tong University
xuxj@apex.sjtu.edu.cn

Chang Liu
University of California, Berkeley
liuchang@eecs.berkeley.edu

Qian Feng
Samsung Research America
qian.feng1@samsung.com

Heng Yin
University of California, Riverside
heng@cs.ucr.edu

Le Song
Georgia Institute of Technology
lsong@cc.gatech.edu

Dawn Song
University of California, Berkeley
dawnsong@cs.berkeley.edu

Presented By:
Andrew Calvano

Binary Code Function Similarity Problem

- Binary code is output during compilation and is contained in an executable or library file.
- Binary code is organized into functions
  - Usually one source code function corresponds to one binary code function
- Similarity task is to compute measure of similarity between binary functions
  - Used to determine if function A is directly or indirectly related to function B
- Particularly difficult to compute similarity of functions across different architectures

Example

- Function compiled from same source file in two different binaries
- First binary
  - Compiled by GCC v7.3.0 (x86_64) with no optimizations
- Second binary
  - Compiled by GCC v5.5.0 (Aarch64) with -O3
- The goal is to determine that the two different functions are the same

Applications

- Anything that requires comparing of code from syntactic and semantic point of view
  - Vulnerability Detection
  - Malware Analysis
  - Reverse Engineering
  - Plagiarism Detection, Copyright Violations, Patent Infringement, etc.
  - Authorship Attribution

Reverse Engineering

- When looking at executables or firmware images usually do not have access to debug symbols or source code
- Useful to identity semantically relevant functions to figure out what code is doing
  - Identify printf,malloc,free,etc.
  - Label code identified from previous reverse engineering efforts
- Binary code similarity can be used to help with this task
  - Used to automatically label code in an unseen binary
- Once known functions are labelled appropriately analysis becomes easier

Malware Analysis

- Determine if binary function is similar to known malicious functions
  - Develop library of malicious signatures for matching
  - Any suspicious code be examined for matches against signature database
- Use similarity metrics to perform attribution
  - Use binary similarity metrics to collect evidence of specific malicious actors responsible
- For malware that does not sit nicely on disk static binary similarity can be problematic
  - Obfuscation through packers and cryptors can disrupt analysis tools
  - Can prevent binary code similarity from working at all
  - Dynamic binary similarity likely better for this use case

Plagiarism, Copyright, Patent Infringement

- Detect plagiarism by computing similarity of student code base against other submitted code bases
  - Can be done at function level or by computing threshold over entire code base
- Copyright violations and patent infringements may be detected by comparing similarity of protected code against suspected competitors
  - May be used as evidence that an infringement or violation exists
  - Alternatively, can be used by competitor to ensure that they are not violating any existing protections.

Vulnerability Detection

- Assessing a security patch from a closed-source vendor such as Microsoft or Adobe
    - Need to identify differences in similar functions between unpatched binary and patched binary
- Use binary function similarity to identify functions with high similarity scores
    - Creates prioritized list of similar functions with minor differences
    - Differences can be used to identify semantic changes in code to determine if patch fixes vulnerability

Vulnerability Detection

- Use binary function similarity to determine if vulnerability exists across supported software versions
  - Vulnerability in Windows 7 may be present in Windows 8.1 or Windows 10
- Locate known vulnerable function in one version and compute similarity in differing versions
  - Prioritize analysis of matches to determine if vulnerability exists

Vulnerability Detection

- IoT devices containing similar vulnerable code may be compiled for different architectures and for varying hardware
- Goal is to identify vulnerable function in some firmware image and see if it exists in other firmware images
- Compare known vulnerable function from firmware image with functions in other firmware images

Previous Binary Similarity Strategies

- Pairwise Graph Matching and Heuristics
  - Bindiff/Diaphora
  - Bipartite graph matching
- Dynamic Analysis
  - Compare inputs/outputs of similar function candidates
  - “Blanket execution paper”
    - Synthesize environments for code to execute in and map in fake pages for unmapped accesses
    - Collect features on execution characteristics per function
    - Use SVM to optimize execution feature weights for similarity
- Graph embedding using Codebook/centroiding
  - “Genius” paper

Genius Paper

- Generate code book (centroids) of Attributed Control Flow Graphs (ACFGs) to be used in comparison operations
  - Raw Feature Similarity
    - Bipartite graph matching using cost function computed from ACFG features (statistical and structural)
  - Clustering
    - Spectral clustering to create codebook (n == 16 in practice)
  - Feature Encoding
    - Maps input ACFG to higher dimensional space (size n)
    - Each dimension is distance to centroid in codebook
    - Bag of features vs VLAD encoding
  - Online Search
      - Locality sensitive hashing

Attributed Control Flow Graphs

- Functions are organized into graphs based on control flow
- Control flow graphs:
  - Basic blocks
    - Instructions that execute sequentially without deviation in control flow
  - Edges between basic blocks
    - Represents possible control flow transfers based on conditional and unconditional jumps
- “Attributed Control Flow Graphs” (ACFGs)
  - Basic blocks annotated with attributes or features
    - Basic Block Level Features:
      - Number of types of instructions and constants referenced, etc.
    - Inter Basic Block Features:
      - Number of offspring and betweenness score, etc.

Diagram of ACFG from paper

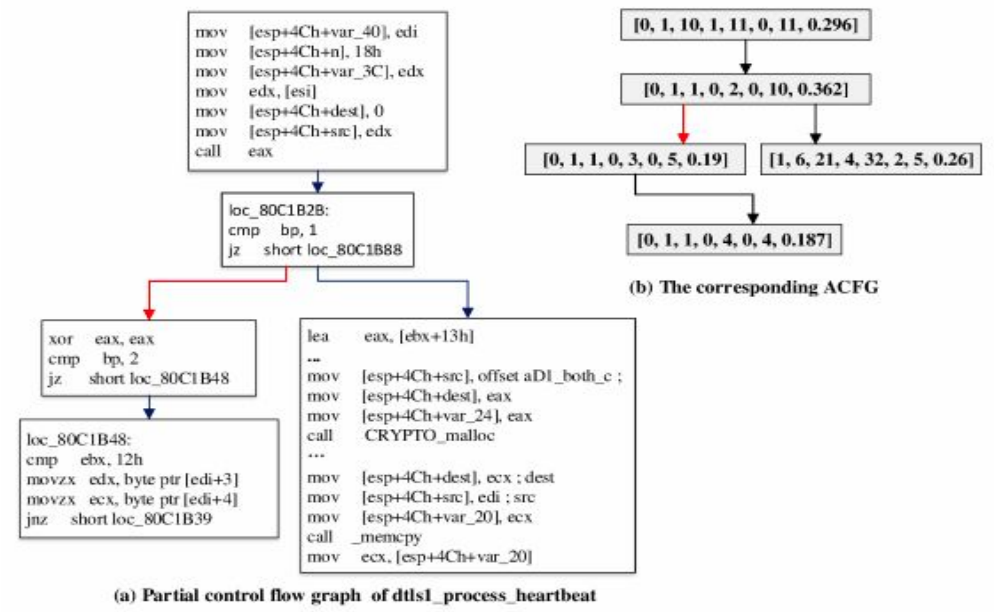


Figure 2: An example of a code graph on Function `dtls1_process_heartbeat` **(Heartbleed vulnerability)**

Genius Approach

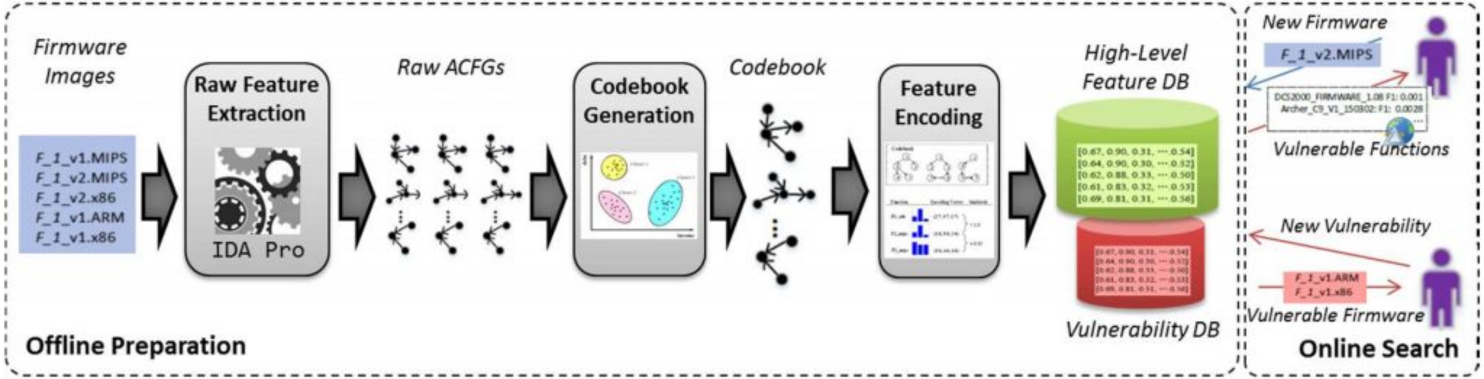
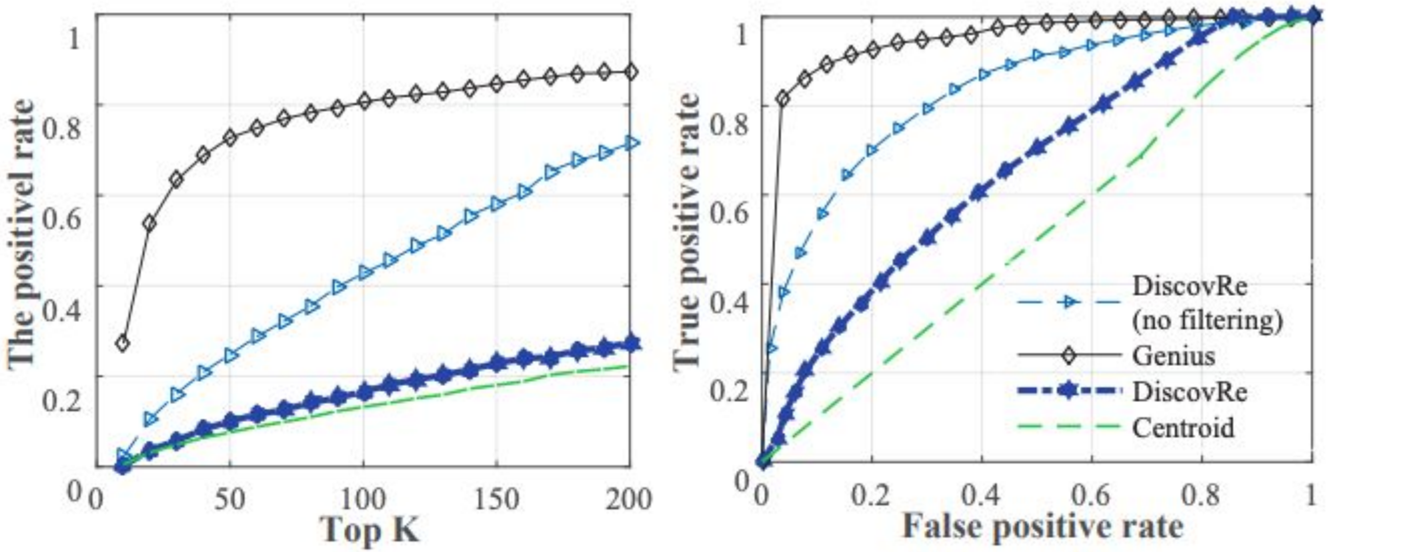


Figure 1: **The approach overview**

Genius Evaluation Results



a) Recall rates across different threshold K

b) ROC curves for different approaches

Figure 4: **Baseline comparison for accuracy on Dataset I.** K **means that we consider retrieved candidates on top K as positives** Two figures share the same legends.

Genius Limitations

- Codebook generation very slow
- ACFG extraction slow
    - Requires betweenness centrality metric
- Recall
    - Top-K results count as positive matches
    - For small values of K recall is not good at all
    - Only 27% of the time in eval is correct match rated 1 in top-K query results

Gemini Overview

- Improve state of the art in binary function similarity detection using deep learning
    - Specifically, be able to apply to compare functions across different platforms and architectures (x86,ARM,MIPS,etc.)
- Feed-forward neural network to learn graph embeddings
    - Training data is pairs of similar and dissimilar binary functions
    - Can retrain on demand to incorporate expert supplied data
- Create graph embeddings for unseen functions for use in comparison operations
- Evaluate computed graph embedding network with test set for similarity analysis

Gemini Workflow

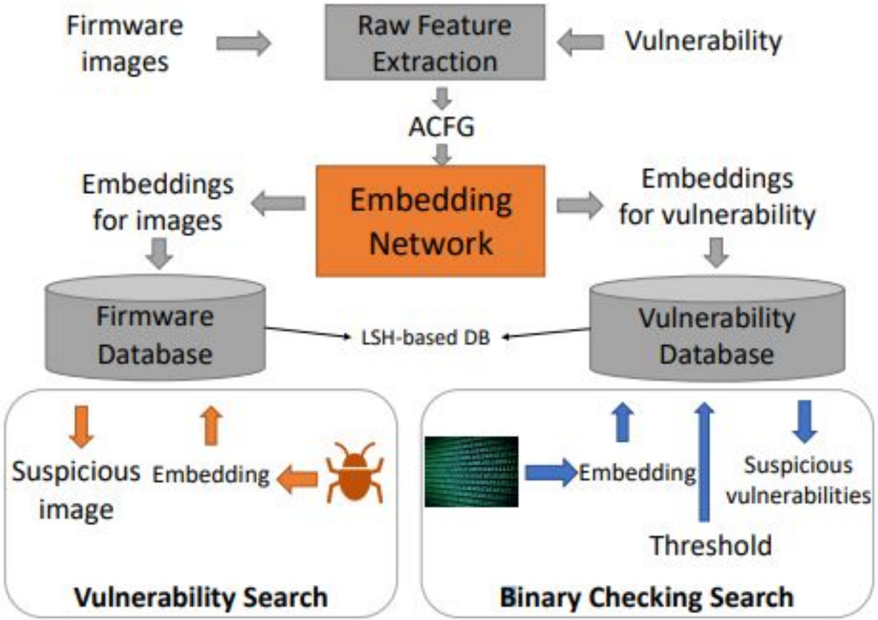


Figure 1: Cross-platform Binary Code Search Workflow

Graph Embeddings

- A numerical vector encapsulating raw features of graph
- Useful to compare with instead of comparing raw features directly which may be expensive
- Embeddings can be quickly indexed and stored for comparison
- Embeddings created from different ACFGs can be compared quickly using methods such as cosine or euclidean distance
- As Example:
  - Take 7 features from ACFG and embed into an embedding vector of size 64

Adapted structure2vec embedding network

- Neural network model to create embeddings from a graph structure
  - Incorporates knowledge of graph topology in embedding creation
- Each vertex in graph has a vector of raw features and an embedding vector
- Network propagates vertex features across vertex embeddings based on graph topology
- During embedding process:
  - Vertex embedding vector updates includes neighbor vertex embedding vectors
  - Runs for T iterations where T is “number of hops” in graph
- Entire graph embedding is created by using aggregation operation over all vertex embedding vectors at the end of T iterations

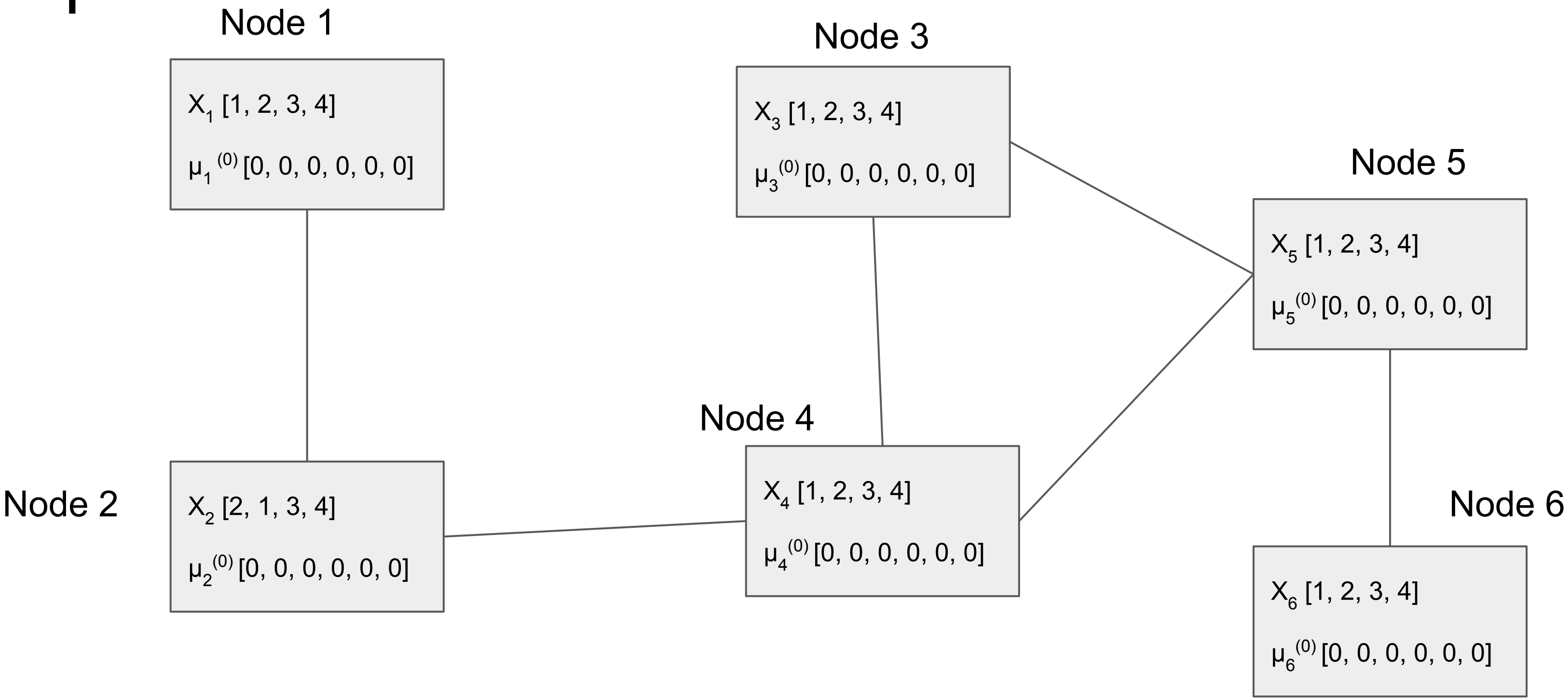
Dimensions
X_v = 1 x d
W_1 = d x p
μ = 1 x p
P_1 ... P_N = p x p
W_2 = p x p

Algorithm 1 **Graph embedding generation**

Input: ACFG $g = \langle \mathcal{V}, \mathcal{E}, \overline{x} \rangle$
Initialize $\mu_v^{(0)} = \overline{\mathbf{0}}$, 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 $\phi(g) := W_2(\sum_{v \in \mathcal{V}} \mu_v^{(T)})$

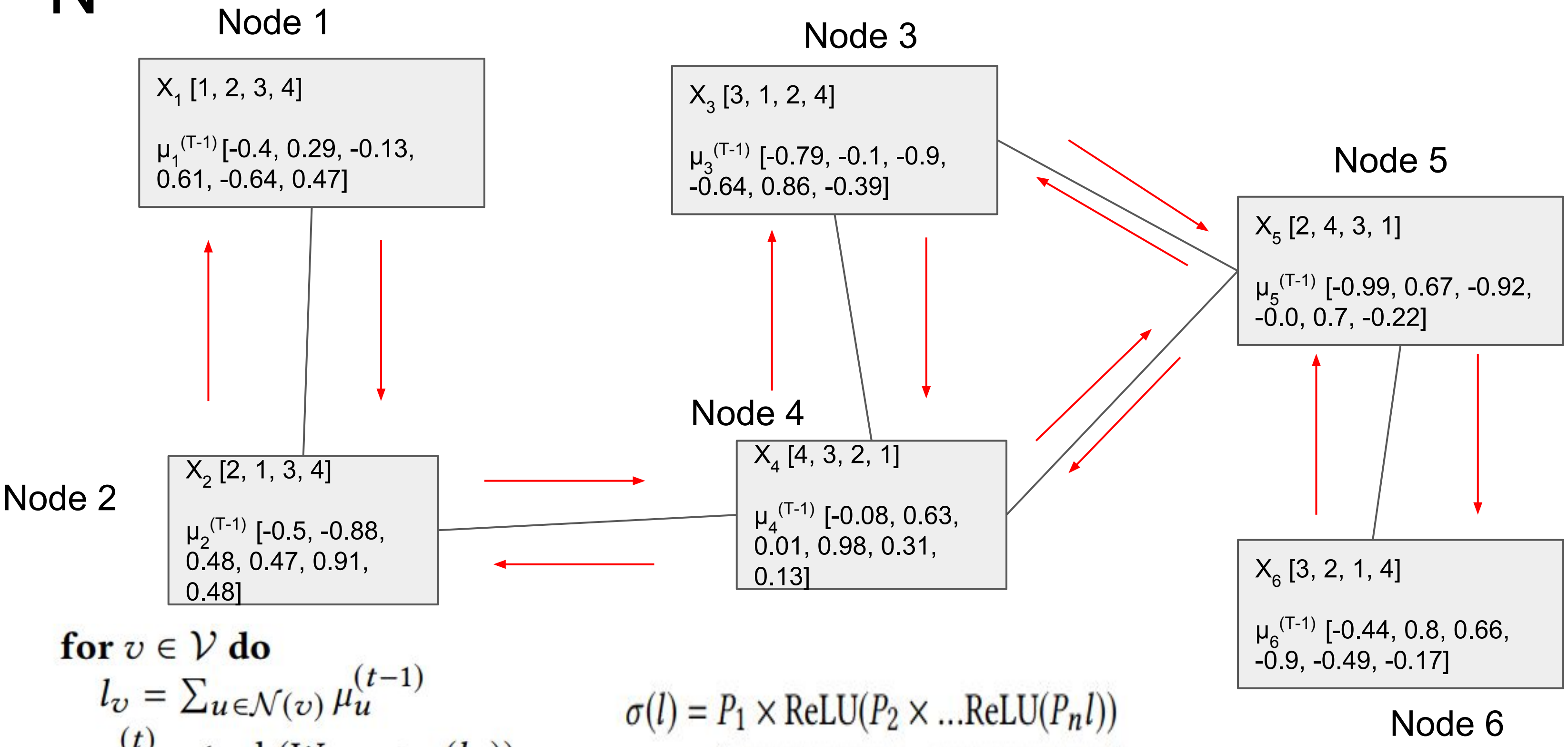
$$\sigma(l) = \underbrace{P_1 \times \mathrm{ReLU}(P_2 \times \ldots \mathrm{ReLU}(P_n l))}_{n \text{ levels}}$$

T = 1



Node 1
X1 [1, 2, 3, 4]
μ1 (0) [0, 0, 0, 0, 0, 0]
Node 2
X2 [2, 1, 3, 4]
μ2(0) [0, 0, 0, 0, 0, 0]
Node 3
X3 [1, 2, 3, 4]
μ3(0) [0, 0, 0, 0, 0, 0]
Node 4
X4 [1, 2, 3, 4]
μ4(0) [0, 0, 0, 0, 0, 0]
Node 5
X5 [1, 2, 3, 4]
μ5(0) [0, 0, 0, 0, 0, 0]
Node 6
X6 [1, 2, 3, 4]
μ6(0) [0, 0, 0, 0, 0, 0]

T = N



for $v \in \mathcal{V}$ **do**

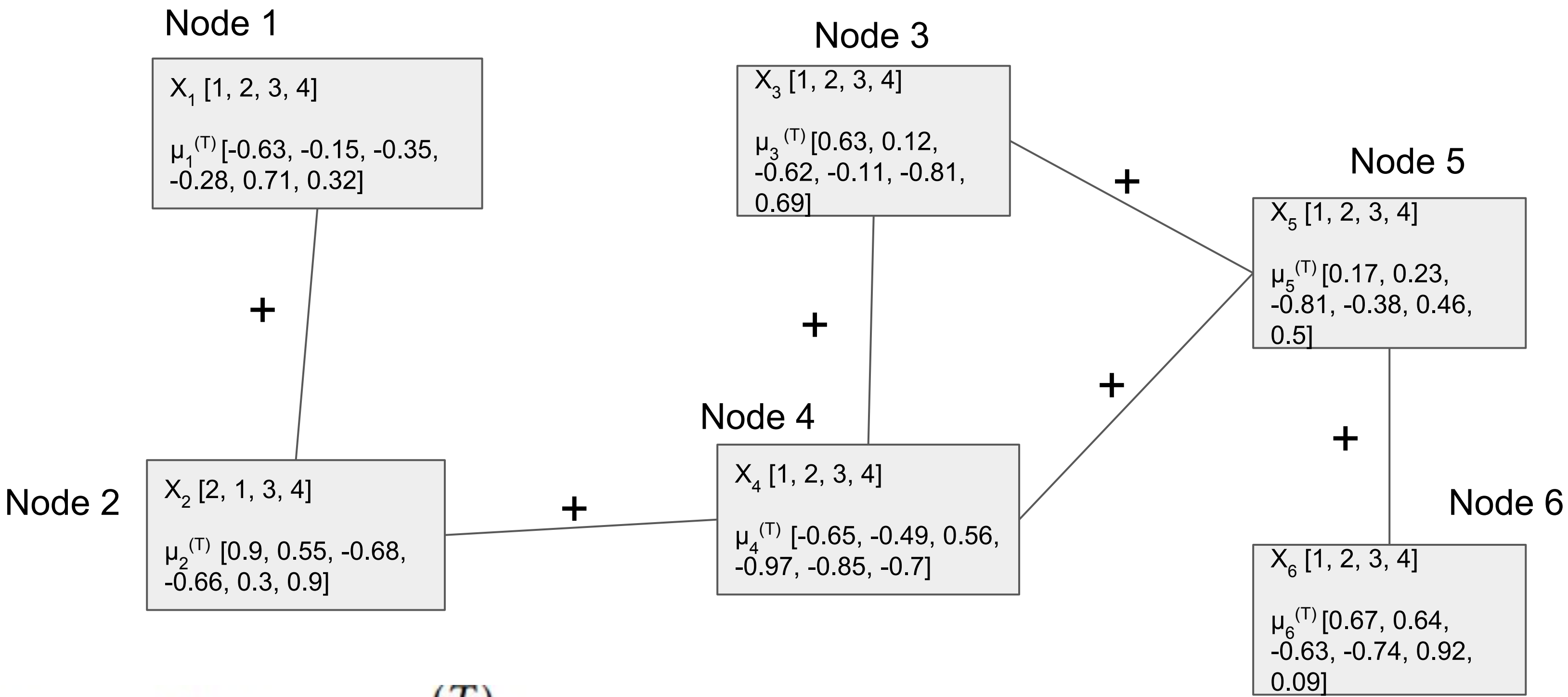
$$l_v = \sum_{u \in \mathcal{N}(v)} \mu_u^{(t-1)}$$

$$\mu_v^{(t)} = \tanh(W_1 x_v + \sigma(l_v))$$

end for

$$\sigma(l) = \underbrace{P_1 \times \mathrm{ReLU}(P_2 \times \ldots \mathrm{ReLU}(P_n l))}_{n \text{ levels}}$$

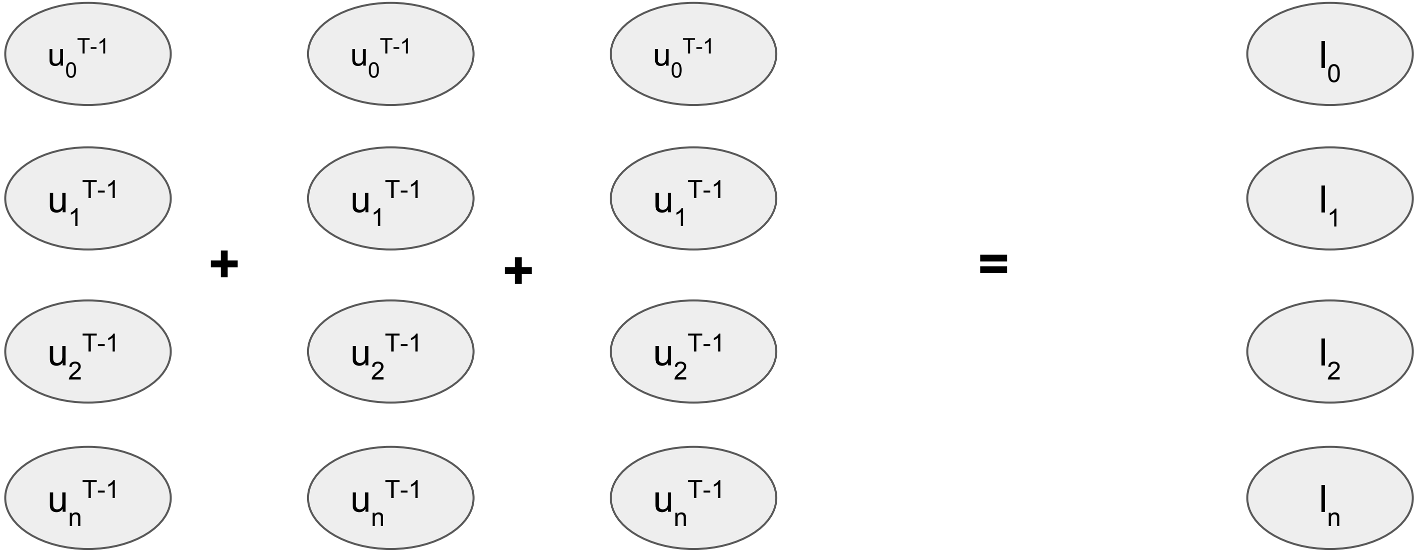
Final



$$\phi(g) := W_2(\textstyle\sum_{v \in \mathcal{V}} \mu_v^{(T)})$$

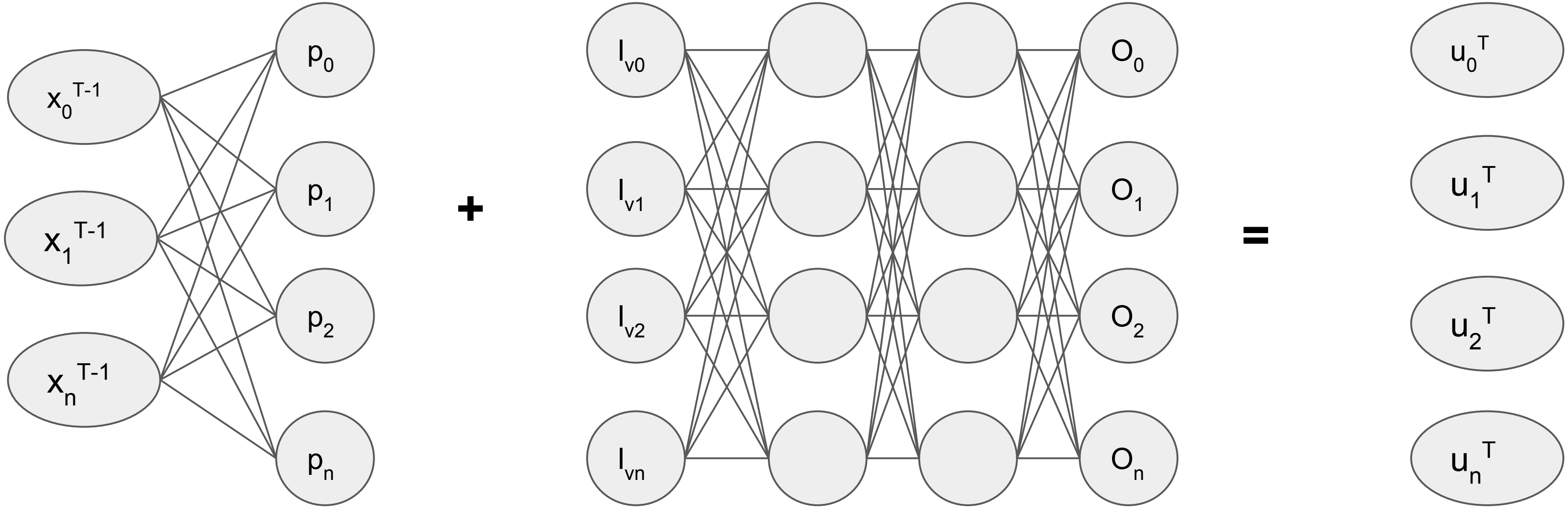
Embedding Network

For each T
For each vertex
For each neighbor

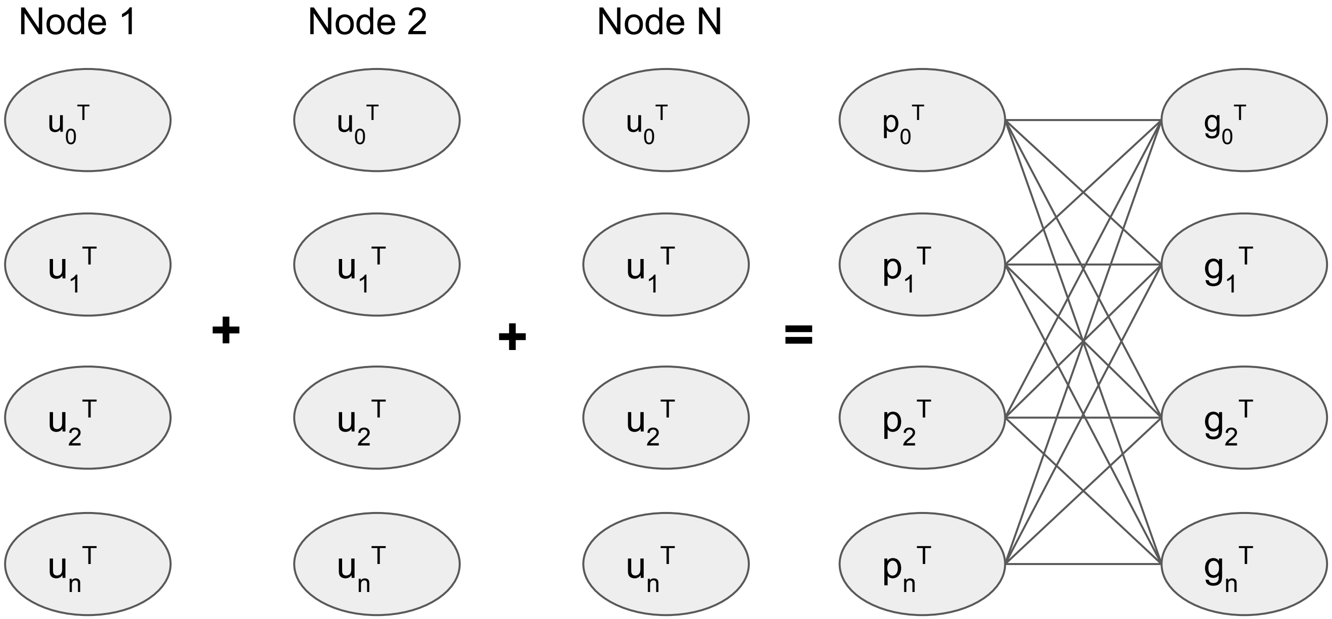


Embedding Network

For each T
    For each vertex



Embedding Network



Task Independent Pre-Training

- Refers to initial training of embedding network
- Learns weights for structure2vec network
- Training data conforms to “default policy”:
  - Each ACFG paired with random similar ACFG and random dissimilar ACFG
  - Similar function pairs (Ground truth +1)
    - Functions compiled from the exact same source code
  - Dissimilar function pairs (Ground truth -1)
    - Functions compiled from different source code
- Training occurs in Siamese neural network architecture joined by cosine similarity function
  - Optimize such that cosine distance is large (close to 1) for similar and small (close to -1) for dissimilar

Similarity Optimization

$$\mathrm{Sim}(g, g') = \cos(\phi(g), \phi(g')) = \frac{\langle \phi(g), \phi(g') \rangle}{||\phi(g)|| \cdot ||\phi(g')||}$$

$$\min_{W_1, P_1, \ldots, P_n, W_2} \sum_{i=1}^{K} (\mathrm{Sim}(g_i, g_i') - y_i)^2.$$

Siamese Network Architecture

- Taken from computer vision field
  - Creates high dimensional image embeddings used to compare images
  - Easier to compare then comparing raw features directly
- Uses two neural networks that share weights during training
- Joins the outputs of the network using a distance function such as cosine or euclidean distance
- Training data consists of similar pairs and dissimilar pairs
  - Feedback based on distance measurement of the two network outputs
  - Close to 1 typically similar, close to -1 typically dissimilar

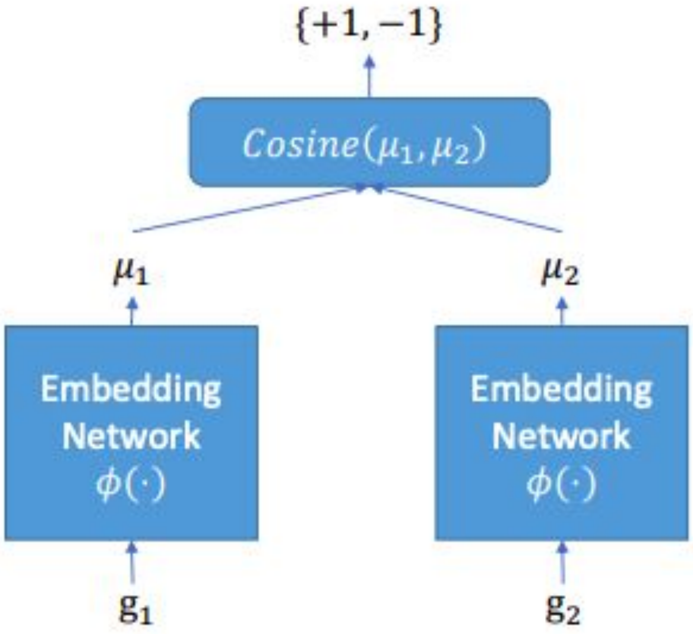


Figure 4: Siamese Architecture

Learning the embedding network parameters

- Structure2vec network weights must be learned to created good embeddings
  - W1 (node weights)
  - P1-PN(neighbor activation)
  - W2 (entire graph)
- Training data set consists of known similar function pairs and known dissimilar function pairs
  - Randomly chosen pairs of similar/dissimilar function
- Pairs fed to network and typical back propagation using stochastic gradient descent from cosine distance calculation
- End of training weights are learned
  - Network can be used to compute unseen function’s embedding

Task Specific Retraining

- Retrains graph embedding network by updating with human expert supplied knowledge
  - Additional samples of similar/dissimilar function pair ground truth data
- Create ACFGs from function pairs with ground truth data
  - Use specified ACFGs as additional training data
- Train the network for a small number of epochs more
  - Sample ACFGs from new human supplied data many times more than existing data
    - Unclear what they mean by this notion of sampling
- After retraining new parameters can be used for embedding creation

Evaluation

- Gemini evaluated with respect to search accuracy and efficiency
  - Evaluates pre-trained model on known ground truth
  - Evaluate re-trained model on real world data
- Baselines for comparisons
  - Bipartite Graph Matching
  - Genius solution
  - ACFG extractor implemented as IDA Pro script
- Graph embedding network implemented in Tensorflow

Evaluation Datasets

- Dataset 1 (For pre-training and testing embedding network)
  - Binary functions compiled from source code with ground truth similarity information
    - Compiled for different architectures and with different optimizations
  - Total of 129,365 ACFGs extracted from 18,269 binary files produces from various OpenSSL builds
    - Split into training, validation, and testing sets.
- Dataset 2 (Firmware images from Genius)
  - 33,045 firmware images with 8,128 being in scope
  - Obtained from 26 different IoT vendors
- Dataset 3
  - Functions with variable size ACFGs from random 16 firmware images
- Dataset 4
  - 154 vulnerable functions

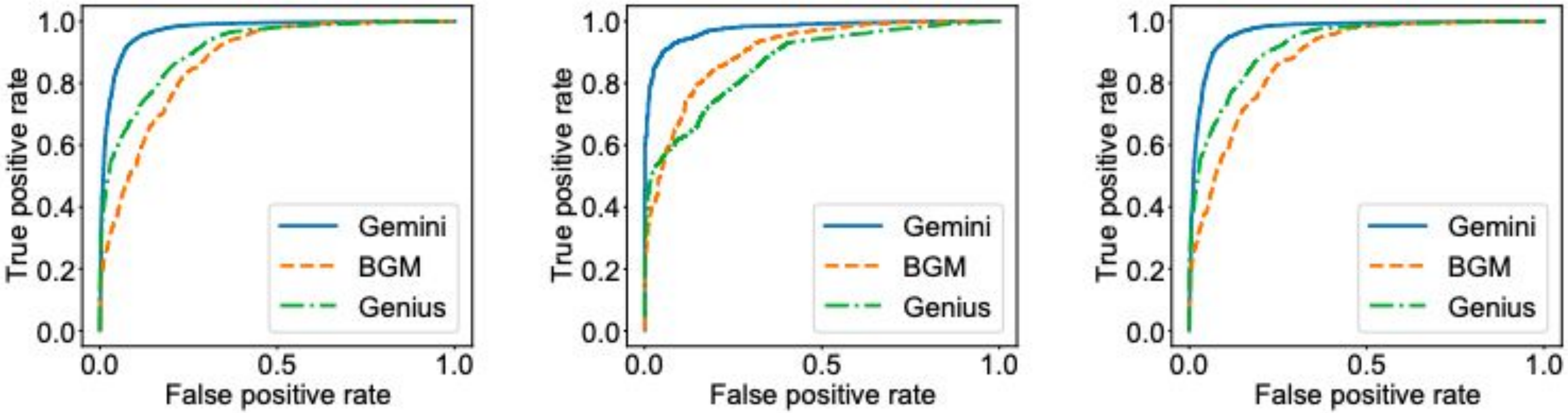
Task Independent Pre-training

- Trained using training set derived from Dataset 1
- Training data
  - At each epoch training data is randomly selected
  - For each ACFG in training set
    - One similar and dissimilar ACFG is randomly selected and assigned ground truth
    - Training data randomly shuffled before being fed to siamese model
- Training details
  - Adam optimization algorithm using learning rate of .0001
  - Siamese model trained for 100 epochs
  - Mini-batch size of 10 ACFG pairs
  - T = 5 and Embedding Depth (p) = 64
  - After every epoch loss vs validation set is calculated
  - Over 100 epochs the model that achieves the lowest loss is selected

Task Independent Pre-training Evaluation

- Evaluated using dataset 1 derived testing set
- Training set and testing set contain exclusive sets of functions
  - E.g. If function A is in training set no version of function A was in testing set
- Similarity testing set constructed as:
  - For each ACFG in testing set
    - Randomly select 1 similar ACFG in testing set
    - Randomly select 1 dissimilar ACFG in testing set
  - Total sets of ACFG triples in testing set: 26,265
- Similarity testing set also split
  - Large graph subset (At least 10 vertices)
  - Small graph subset
- General AUC claim of .971 when eval on testing set

Task Independent Pre-training Evaluation Results



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

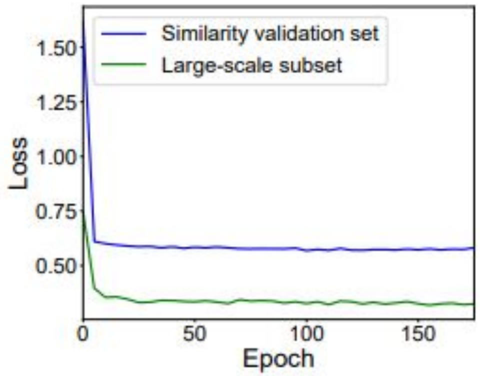
Figure 5: ROC curves for different approaches evaluated on the testing similarity dataset.

Embedding Visualization

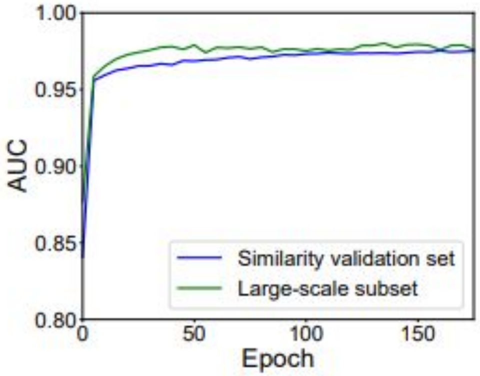
Figure 8: Visualizing the embeddings of the different functions using t-SNE. Each color indicates one source functions. The legend provides the source function names.

Hyperparameter Evaluation

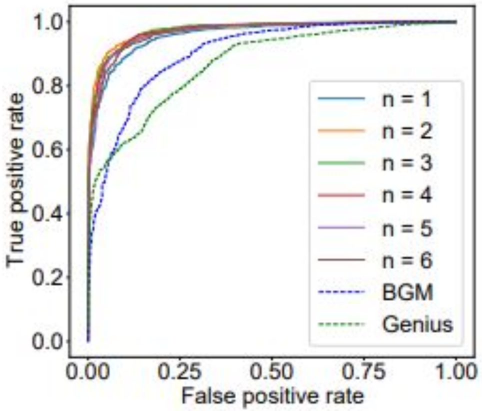
- Hyper parameters evaluated by experimentation
- Number of epochs
  - Good performance on validation set after training for 5 epochs
  - Lowest loss on validation set after 100 epochs
- Embedding depth
  - Depth of embedding neural network best using 2 layers
- Embedding size
  - Best performance at 512 but very good performance at size 64
- ACFG Attributes
  - Best performance using block level attributes + number of offspring
- Number of iterations in embedding algorithm (T)
  - Best performance observed when T is 5



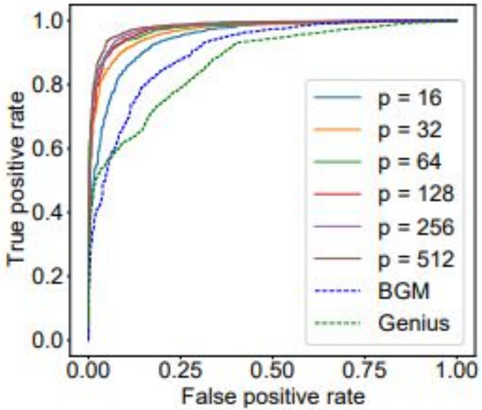
(a) Loss versus no. of epochs.



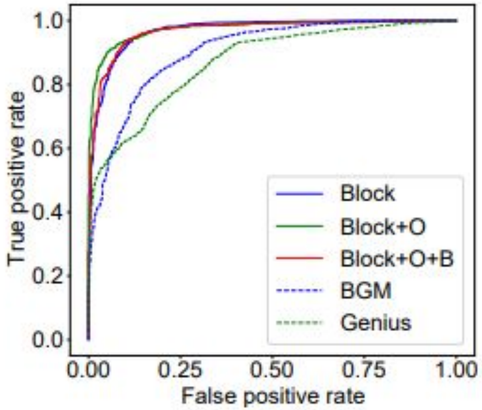
(b) AUC versus no. of epochs.



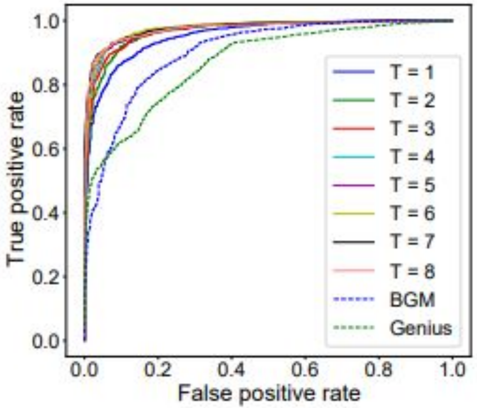
(c) ROC versus embedding depth n.



(d) ROC versus embedding size p.



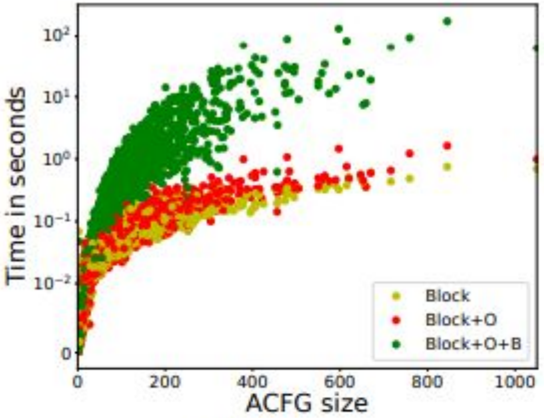
(e) ROC versus ACFG attributes.



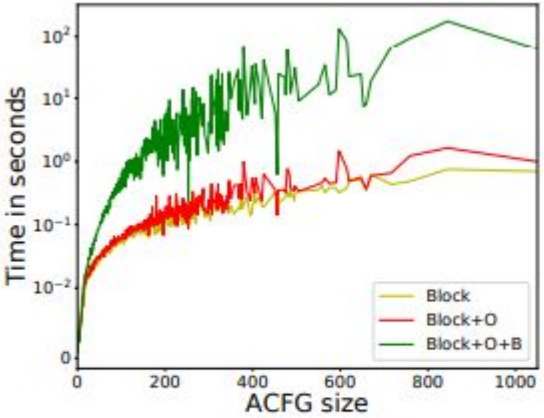
(f) ROC versus no. of iterations T.

Efficiency Evaluation

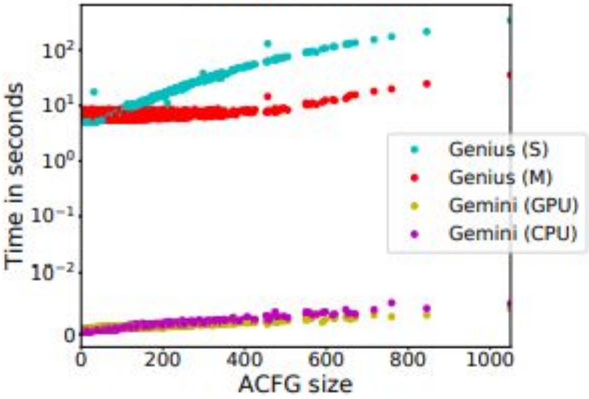
- Evaluated using data set 3
- Results
  - ACFG extraction time
    - Improves 8x on average over Genius
      - Exclude betweenness feature
  - Embedding generation time
    - 2400x to 16000x faster than Genius
    - No graph matching needed
    - Gemini implemented using parallelizable matrix operations
  - Overall latency of embedding generation
    - Average 386.4x faster than Genius



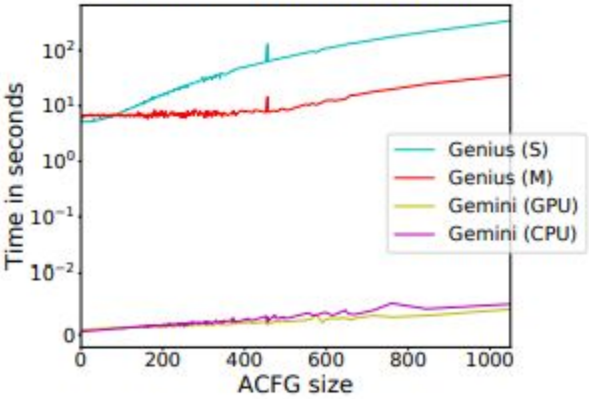
(a) ACFG Extraction time



(b) Average ACFG extraction time



(c) Feature vector generation time



(d) Average feature vector generation time

Training Time Evaluation

- Evaluated due to retraining use case
    - New firmware images pushed by vendors
        - Need to incorporate into network model
- Each epoch trains on 206,000 examples
    - Takes 5 minutes
- Performance surpasses Genius after training for 5 epochs
    - Around 30 minute training time
- Best performance after training for 100 epochs
    - Around 8.5 hours

Task Specific Re-training Evaluation

- Extract all ACFGs from Data Set 2 (420,558,702 functions)
- Select two vulnerable functions from Data Set 4 (same used in Genius)
- Retrain pre trained (Data Set 1) embedding network
    - Compute embeddings of all functions in Data Set 2
    - For each vulnerability query
        - Retrieve Top K results
        - Manually assign ground truth data to all top k results
            - Paper claims 2 hours of manual investigation time for k = 50
    - Retrain using Top K ground truth similarity pairs
    - After each retraining iteration compute new embeddings of 10% of data set 2 and repeat
- Evaluate (1 retraining iteration) using same vulnerabilities as Genius
    - Gemini 84% positive (84 of top-100 results) vs Genius 28% positive (14 matches of top-50)

Limitations

- They only evaluated pre trained model on code from same code base (OpenSSL)
- Manual investigation required for retraining
- Limited to comparing similarity at the function level
  - Would struggle with inlined code
  - Not possible to compare subgraph of functions
- They do not take into account data flow information
  - Only control flow information is examined which is only one component of what the code is doing.
- Requires complete and correct recovered control flow graph
  - They looked at C code bases, for C++ this can be much more difficult

Future Work

- Incorporate intraprocedural data flow information
  - Train either exclusively with data flow embedding or combine with control flow embedding
  - Could use as additional feature in existing graph embedding using basic block scoped data flow information
    - Number of uses, number of defs, etc.
  - Use dataflow relationships to consider additional propagation vertex neighbors
- Another paper used unsupervised learning to learn ACFG features rather than manually selecting them
  - Boosted performance by 2% positive matches on same data set
- Asm2vec (IEEE S&P 2019)
  - Applies word2vec approaches to assembly code
  - Approach and results not released yet?

Additional References

- https://www.zynamics.com/bindiff.html
- https://github.com/joxeankoret/diaphora
- https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-egele.pdf
- https://www.cs.ucr.edu/~heng/pubs/genius-ccs16.pdf
- https://www.rsaconference.com/writable/presentations/file_upload/ht-t10-bruh_-do-you-even-diff-diffing-microsoft-patches-to-find-vulnerabilities.pdf
- https://googleprojectzero.blogspot.com/2017/10/using-binary-diffing-to-discover.html
- https://github.com/xiaojunxu/dnn-binary-code-similarity
- https://arxiv.org/pdf/1708.06525.pdf
- https://ghidra-sre.org/