I C A N N 2 1

30th International Conference on Artificial Neural Networks

TUM

HawkEye: Cross-Platform Malware Detection with Representation Learning on Graphs

Peng Xu[1], Youyi Zhang[2], Claudia Eckert[1], Apostolis Zarras[2]

{**Peng**,eckert}sec.in.tum.de

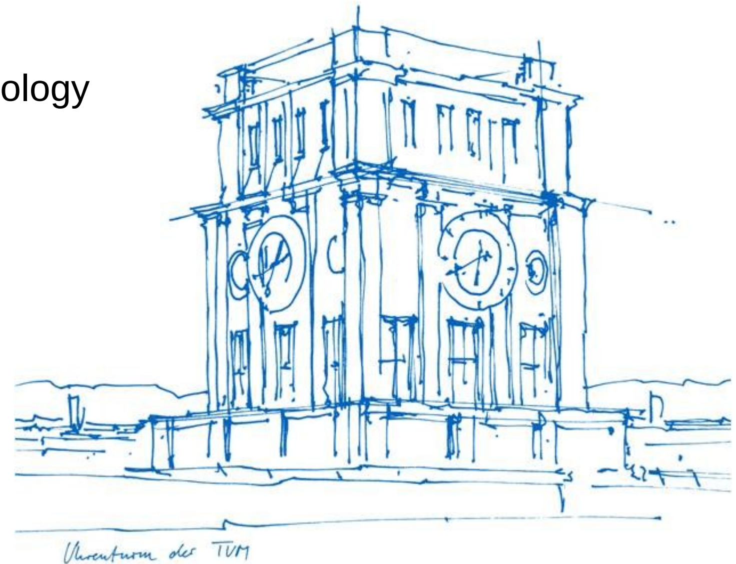
zhangyouyi@yahoo.com

a.zarras@tudelft.nl

[1] Technical University of Munich

[2] Tongji University

[3] Delft University of Technology



Agenda

- Introduction
- Related Work
- System Design and Implementation
- Evaluation
- Conclusion

Introduction

Problems

- ✓ Developed systems can **prevent** malware
  - ➢ they frequently target **one specific platform/architecture**, and thus, they cannot be ubiquitous.
- ✓ Malware authors use **obfuscation** technique
  - ➢ code obfuscation techniques used by malware authors can **negatively influence** detecting and preventing performance.

Introduction

Motivation

✓ **Cross-platfrom Detection**

- The **native libraries** of **Android** apps, including the malicious native code, are cross platform (i.e., x86, ARMv7, ARMv8).
- Modern ICT environments need **mixture** malware detection system on Router, Terminals, Workstation and PC
- **Control Flow Graphs** (CFG) based method is a solution in the right direction because all programs have CFGs

Introduction

Contribution

- ✓ design and implement **HawkEye**, a **cross-platform** malware **detection** framework
- ✓ a hybrid **Control Flow Graph** and **Graph Neural Networks** and is inspired by Natural Language Processing.
- ✓ includes **three primary** components: (i) a graph generator; (ii) a graph-neural-network-based graph embedding method (iii) a machine-learning classification
- ✓ **outperforms** numerous malware detection solutions.

Related Work

✓ Feature-code-based methods

Related Work

- ✓ Feature-code-based methods
- ✓ Machine/Deep learning-based methods
  - ➢ **Malicious API**(Mainfest file) based methods

Related Work

- ✓ Feature-code-based methods
- ✓ Machine/Deep learning-based methods
  - ➢ **Malicious API**(Mainfest file) based methods
  - ➢ **Permission**-based methods

Related Work

- ✓ Feature-code-based methods
- ✓ Machine/Deep learning-based methods
  - ➢ **Malicious API**(Mainfest file) based methods
  - ➢ **Permission**-based methods
  - ➢ **Network-traffic**-based-methods

Related Work

- ✓ Feature-code-based methods
- ✓ Machine/Deep learning-based methods
  - ➢ **Malicious API**(Mainfest file) based methods
  - ➢ **Permission**-based methods
  - ➢ **Network-traffic**-based-methods
  - ➢ **Program-code-based methods**
    - **Control flow graph**-based
    - **Function-call**/**API-call** graph-based
    - Executable-file pattern-based

Related Work

Table 1: Comparsion with previous works

	Approaches	Description
CFG	Gemini [15], MAGIN [16] Adagio [4]	*ACFG + Manually indicated features* *CFG + Manual one-hot embedding features*
Byte Sequences	MalConv [10] Ember [3]	*Convolutional + trainable embedding* *Gradient boosted decision tree + LightGBM*
CFG+NLP	Pektaş et. al. [9] `HawkEye`	*Malware Detection + call graph + graph embedding* *Cross-platforms + Representation learning on graph*

Design and Implementation

✓ Overview

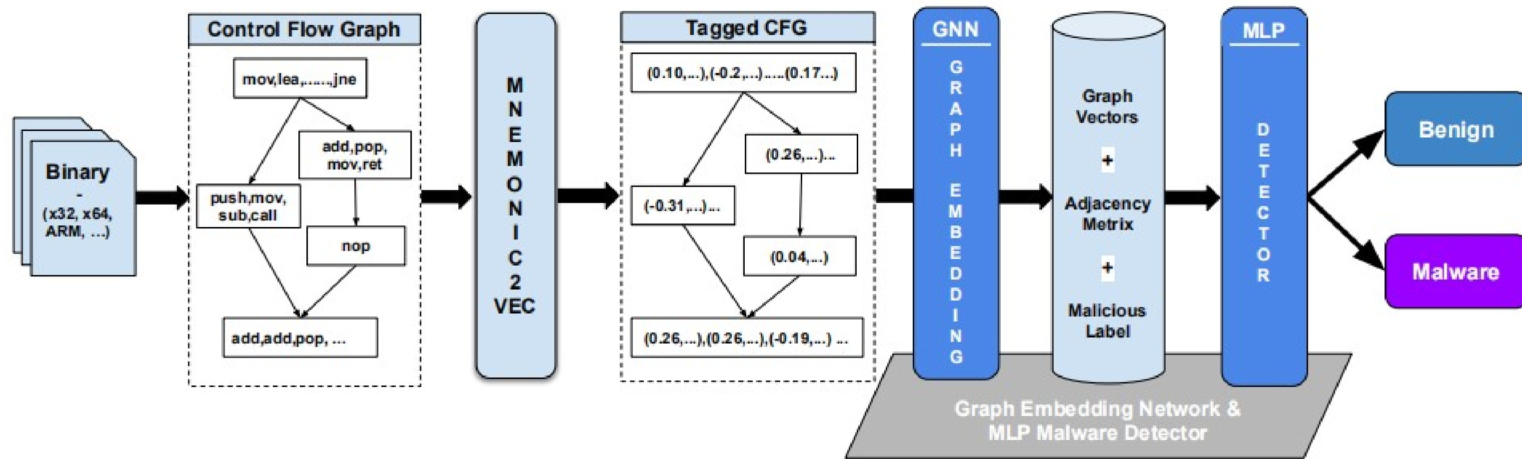


Fig. 1: System architecture

$$loss = \min \sum_{i=1}^{n} \lambda(f(g_i(V, X, A)) + \delta w(f)) - y_i$$

$G^D(V^D, E^D)$, D presents the number of the graph $g_i(V, X, A)$

HawkEye G^D to Y^D, $f : G \rightarrow Y$ to predict whether an executable file is malicious

Design and Implementation

✓ Overview

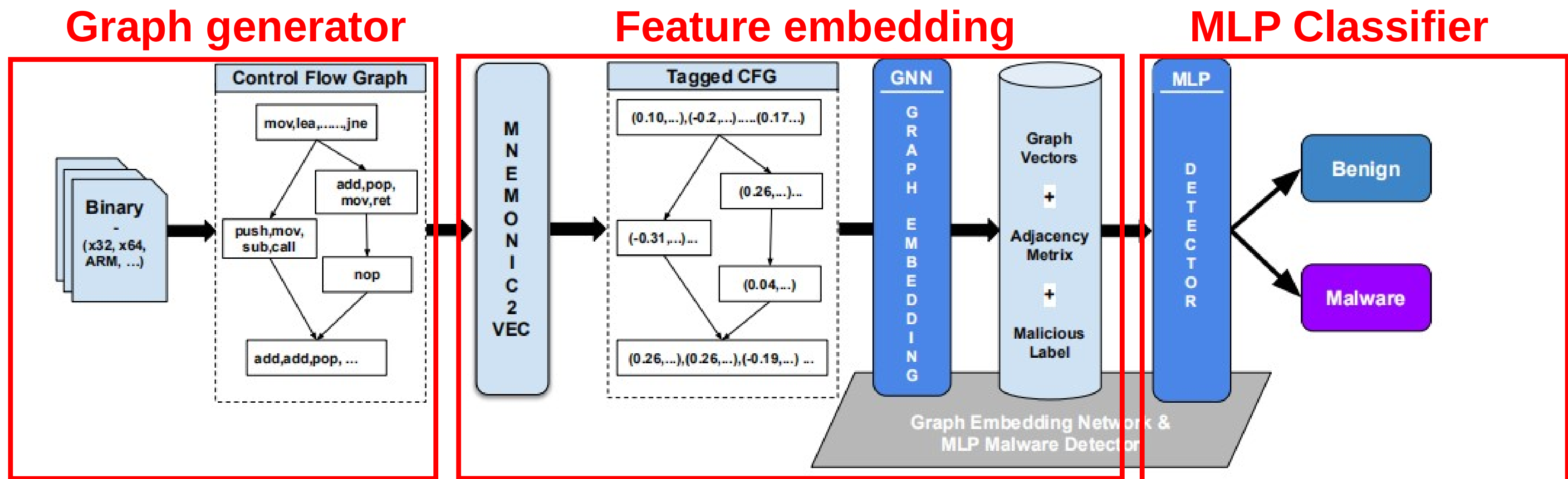


Fig. 1: System architecture

Graph generator, Feature embedding and MLP-based classifier

Design and Implementation

- ✓ Overview
- ✓ **Graph Generator**
  - ➢ Extract flow graphs from the executable binaries
    - Types
      - static&dynamic control flow graph, function call graph
    - Expression
      - $G = (N, E)$, N: function/basic block, E: caller/callee
    - The dCFG and the fCG are **reduced graph** of the sCFG

Design and Implementation

- ✓ Overview
- ✓ Graph Generator
- ✓ **Feature Embedding**

  - **Mnemonic embedding**
    - **Machine Instruction:** *label:[mnemonics][operands][comment].*
    - **Mnemonic :** *mov*, *add*, and *sub*.
    - **Operands :** immediate number, register, and memory,
    - **Mnemonic2vec:** skip gram sampleing word2vec

Mnemonic will **lost operands** infomration, operands have their **inherent variability**

Design and Implementation

- ✓ Overview
- ✓ Graph Generator
- ✓ **Feature Embedding**
  - ➢ Mnemonic embedding
  - ➢ **Block embedding**
    - **Normalization:** $x_{normalization} = \frac{x - Min}{Max - Min}$

Max and **Min** are the maximal and minimal values among all embedding vectors in the basic block. The **first** element of vectors is picked up to get the maximal and minimal values

Design and Implementation

- ✓ Overview
- ✓ Graph Generator
- ✓ **Feature Embedding**
  - ➢ Mnemonic embedding
  - ➢ Block embedding
  - ➢ **Graph embedding**
    - **Structure2vec:** $G = (N, E)$
    - **Vertices**: Functions/Basic Block
    - **Edges**: Caller/callee, jump/return/jne instruction

Design and Implementation

- ✓ Overview
- ✓ Graph Generator
- ✓ Feature Embedding
- ✓ **MLP Classifier**
  - **MLP Structure:** one input layer, two hidden layers with 32 units, and one output layer
  - **Loss function:** hinge loss

Evaluation

- ✓ Experimental Setup
  - ✓ Platform
    - ➢ Linux X86-64
    - ➢ 128 GB RAM and 16 GB GPU
  - ✓ Software
    - ➢ Tensorflow 2.0.0-beta0
    - ➢ Angr: angr-utils and bingraphvis
    - ➢ Keras 2.2.4
    - ➢ Sklearn 0.20.0
    - ➢ numpy 1.16.4
    - ➢ matplotlib 3.1.1

Evaluation

✓ Experimental Setup

✓ Datasets

Table 2: The number of samples in different datasets

Platforms	Training		Validation		Testing		Total	
	Malware	Benign	Malware	Benign	Malware	Benign	Malware	Benign
Windows-x86	17,910	19,043	5,970	6,347	5,970	6,346	29,850	31,736
Android	15,331	15,000	5,111	5,000	5,111	5,000	25,553	25,000
Linux-x86	5,501	5,693	1,834	1,898	1834	1,897	9,169	9,488
Linux-x64	319	923	106	307	106	307	533	1,539
Linux-ARM32	434	446	144	148	144	148	724	744

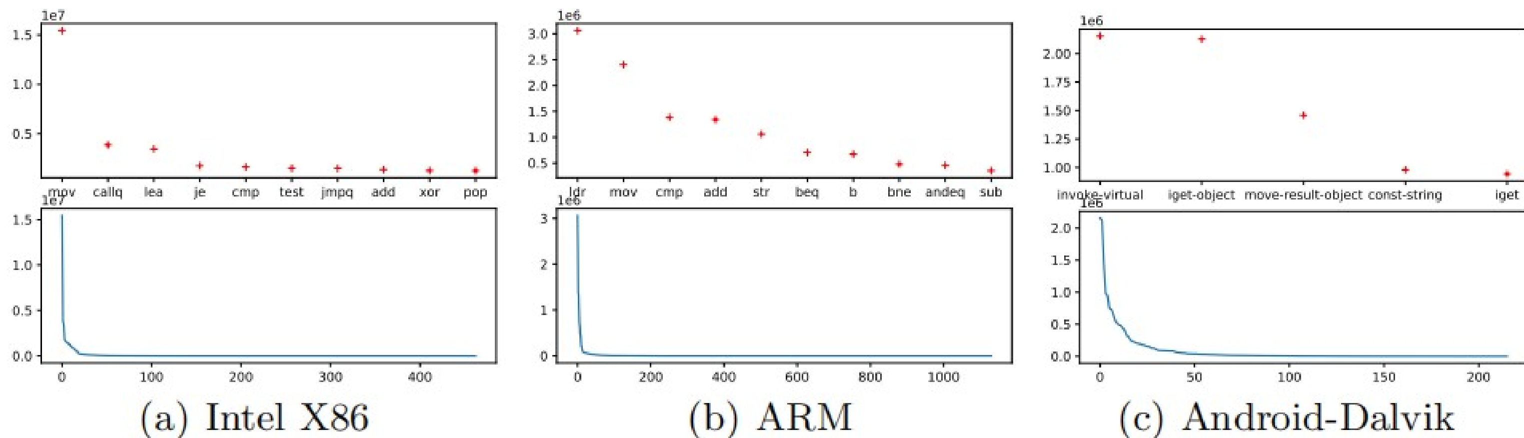
Split the dataset with 60% for training, 20% for validation, and the rest 20% for testing

Evaluation

✓ Experimental Setup

✓ Datasets

✓ Power Law and Opcode Embedding



(a) Intel X86 (b) ARM (c) Android-Dalvik

Fig. 2: Power-law Distribution for Intel, ARM and Dalivk Opcodes.

Evaluation

✓ Experimental Setup

✓ Datasets

✓ Power Law and Opcode Embedding

✓ Results Comparison - Malware Detection Performance

Table 3: Performance comparison with other approaches

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	AUC: (%)
WIN-Ge	93.39	94.79	**97.74**	**96.24**	94.61
WIN-MalConv [10]	90.77	**98.88**	34.34	50.97	82.43
WIN-Ember [3]	**98.23**	97.47	89.72	93.43	**96.67**
WIN-MAGIC [16]	82.46	86.63	82.46	81.96	84.78
ANDROID-Ge	**99.85**	**99.74**	**99.74**	**99.74**	**99.57**
ANDROID-Adagio [4]	95.00	91.07	94.0	95.32	91.07

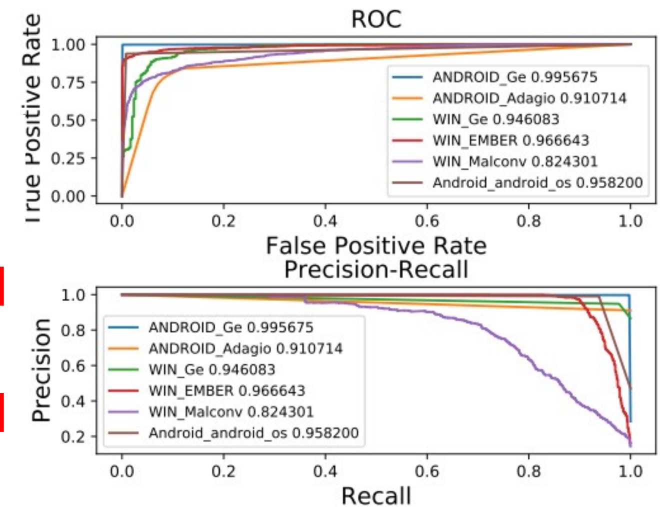


Fig. 3: ROC and precision-recall on Windows-x86 and Android

Evaluation

✓ Experimental Setup
✓ Datasets
✓ Power Law and Opcode Embedding
✓ Results Comparison - Hyperparameters

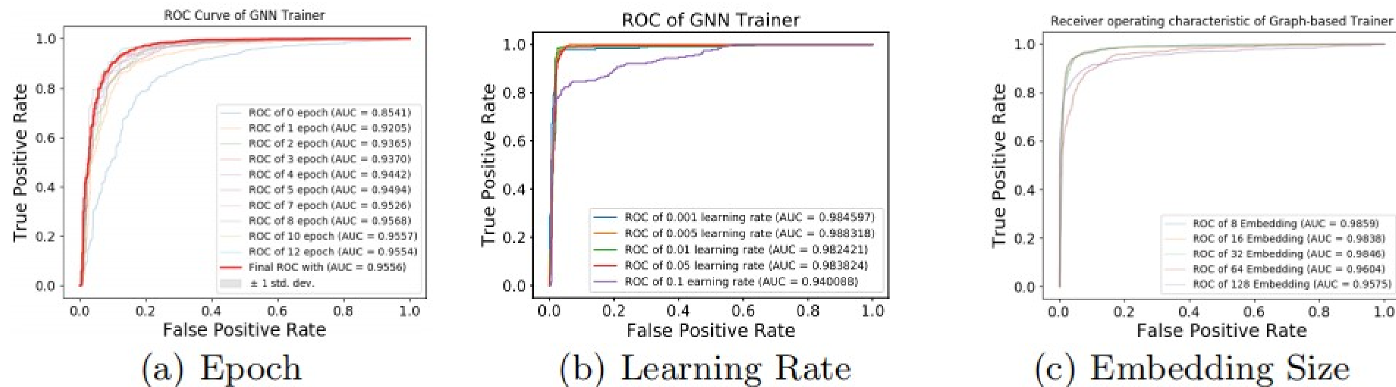


Fig. 4: ROC results with different iteration

- After 12 epochs, the ROC value will be maintained at a certain level.
- When learning rate equals to 0.005, HawkEye gets the best of AUC with 98.83%.
- The embedding size in a specific range does not impact the performance.

Evaluation

- ✓ Experimental Setup
- ✓ Datasets
- ✓ Power Law and Opcode Embedding
- ✓ Results Comparison - Obfuscated Samples

Table 4: Detection rate of obfuscated APK

	ClassEnc.	StrEnc.	Refl.	Triv.	Triv.-Str.	Triv.-Ref.-Str.	Triv.-Ref.-Str.-Class.
PRAGuard[5]	38.0	64.0	96	90.0	50.0	44.0	32.0
Drebin	99.12	98.99	86.58	98.32	98.99	99.32	96.98
Our framework	99.33	98.99	86.58	98.32	98.99	99.32	96.98

Evaluation

- ✓ Experimental Setup
- ✓ Datasets
- ✓ Power Law and Opcode Embedding
- ✓ Results Comparison - Obfuscated Samples

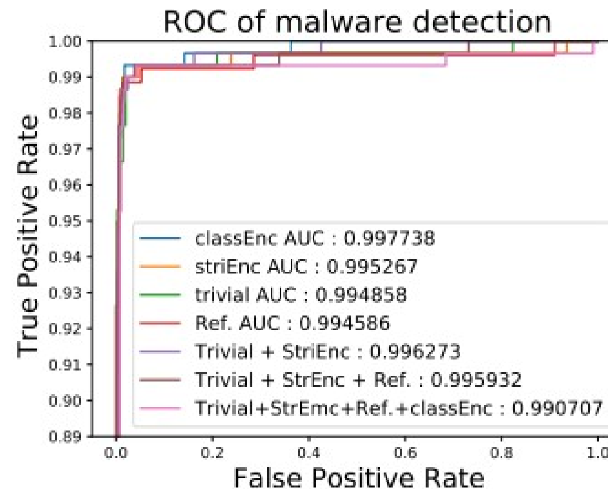


Fig. 5: ROC of obfuscated APK

Conclusion

- ✓ design and implement **HawkEye**, a **cross-platform** malware **detection** framework
- ✓ a hybrid **Control Flow Graph** and **Graph Neural Networks** and is inspired by Natural Language Processing.
- ✓ includes **three primary** components: (i) a graph generator; (ii) a graph-neural-network-based graph embedding method (iii) a machine-learning classification
- ✓ **outperforms** numerous malware detection solutions.

Thank you !!!

Questions?

Comments?