Graph Convolutional Network for Classifying Binaries with Control Flow Graph Data

Micah Flack, Rita Foster, and Shengjie Xu

Abstract—**The ability to precisely identify the threats that unknown binaries propose to an environment is an ongoing issue within cyber security. Malware analysis is typically performed through two general methods, static analysis and dynamic analysis. Through static analysis control flow graphs can be generated to represent the flow of a program between different code-blocks. Data can be further generated to represent individual instructions as VEX commands. By organizing this data into StellarGraph objects, we can use GraphSAGE models to generate low-dimensional embeddings for familial attribution with greater resistance to unseen samples causing mis/overfitting.**

I. Introduction

Because current intrusion detection systems (IDS) and intrusion prevention systems (IPS) rely on static, signature-based solutions, the most minute changes to code can cause the same signatures to be targeted or misclassify malicious binaries as otherwise benign. Malware analysis is typically performed through two general methods, static analysis and dynamic analysis. Static analysis is done without running the binary and the same methods can be performed against source code as well. Whereas dynamic analysis requires running the binary and then inspecting different elements about the binary itself, it's memory, or the environment it is running in [1].

Where this research diverges from traditional methods is the type of static features extracted, their represented format, and the algorithms used to assess familial inferences. In doing so we hope to reduce the number of false positives produced by misfit models classifying unknown binaries; in short, reducing the number of networks or environments compromised by emerging malware or advanced persistent threats (APTs).

II. Inductive Representation Learning on Large Graphs

Executables can have thousands of functions or code-blocks represented within a control flow graph. Because of this, it is important to determine whether there are any valid ways to efficiently generate embeddings and inferences from the nodes within those graphs. [2] presents Graph Sample and Aggregate (GraphSAGE) from the StellarGraph library [3], an inductive framework that leverages node attribute information to generate embeddings via forward propagation for previously undigested data.

M. Flack and R. Foster are with the Idaho National Laboratory, Idaho Falls, Idaho 83415. S. Xu is with The Beacom College of Computer and Cyber Sciences, Dakota State University, Madison, SD 57042. E-mail: micah.flack@inl.gov, rita.foster@inl.gov, shengjie.xu@dsu.edu.

III. Metamorphic Malware Detection using Control Flow Graph Mining

There are different ways that executables can be represented, but the three most common are: source code, disassembly, and control flow graphs. Each of these representations allows us to derive different information or indicators which will be the primary data source for our models. For instance, with source code we know exactly what the code originally looked like before the executable was compiled. We can see the libraries that were imported, the functions called from those libraries, as well as the structure of any methods or data types.

We can disassemble an executable and use the raw assembly or opcodes produced from that disassembler to count opcode frequency, instruction count, or to even track references to addresses within memory. Similarly, we can use that same disassembly to create decompilations which are a form of pseudo-source code meant to improve the understanding of a program. Depending on the language an executable was written in, decompilation will produce wildly different results. For instance, the difference between the decompiler used by dnSpy for .NET code versus the results produced by IDA with Hex-Rays and any C or C++ programs [5]. [4] discusses how we can use the third representation type, control flow graphs (CFG), to gather further data as features for our datasets.

Control flow is indicated by instructions using an unconditional jump jmp, a conditional jump jcc, a function call call, or a function return ret operator. Using this intuition, we can build CFGs defined by the entrypoint address of a function to where any of the previously indicated instructions are made. This will produce CFGs consisting of multiple blocks interconnected by arrows; an example of such a graph is shown in Figure 3.

IV. Proposed Solution

The phases for this proposal are as defined by Figure 4. Under the Data Generation Phase malicious portable executable (PE32) samples will be gathered from the shown sources: *VirusShare*, *HybridAnalysis*, and *VirusTotal*. The benign samples will be gathered from the same sources as well as from verifiable Microsoft Windows images. The exact distribution for this sampling is 991 malicious and 991 benign samples.

A. Data Extraction

The CFG data will be extracted from the samples using angr, a multi-architecture binary analysis toolkit, with the ability to perform dynamic symbolic execution, and various static analyses on binaries. For the purpose of this proposed solution, we will use a static CFG (CFGFast) to generate a CFG for each

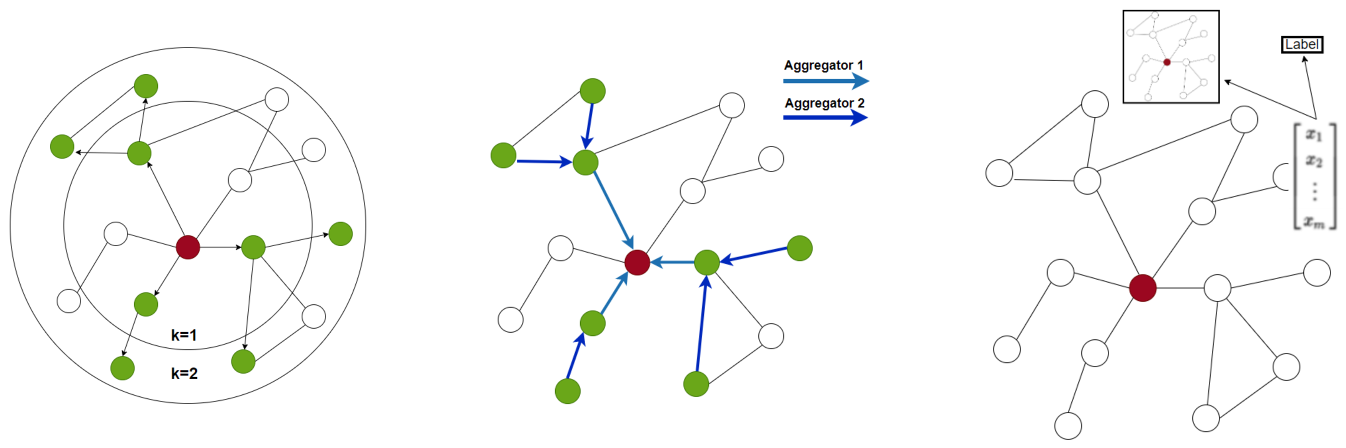

Fig. 1: Visual demonstration of GraphSAGE sampling and aggregation.

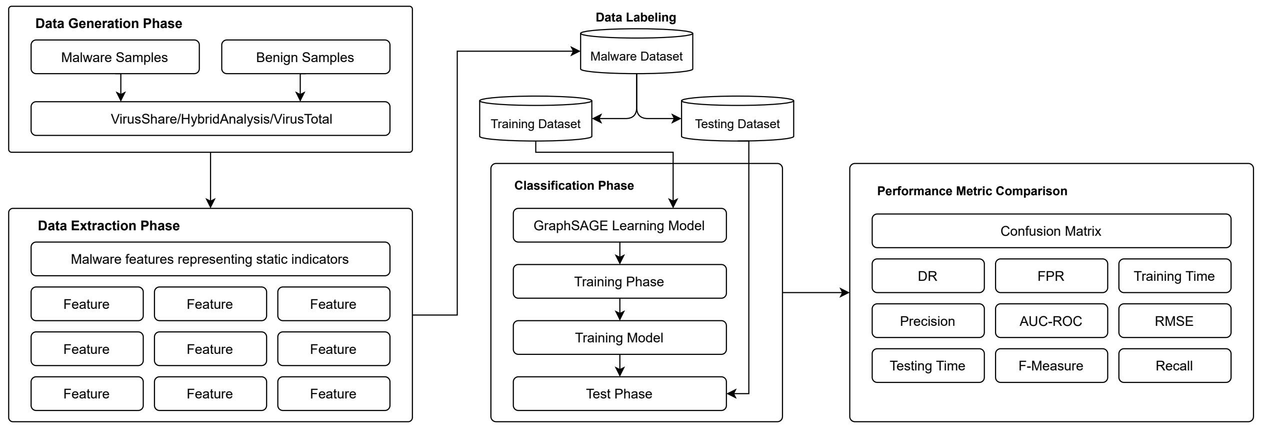

Fig. 2: Four main phases of the proposed solution.

sample. The analytical data for these CFGs are stored using the VEX IR, an architecture-neutral intermediate representation, of each node or block within Neo4j [6].

B. Data Labeling

Through the use of angr, the VEX commands chosen to identify code blocks within a given sample are the following:

[table 1]

These 38 features represent the different nodes features for each block within Stellargraph. Brief descriptions of what each feature represents can be found at [7].

C. Classification

The GraphSAGE framework from Stellargraph is currently the preferred method for supervised learning and node classification of the CFGs and their node features generated by angr. Using this framework will allow typical splits of 70% training and 30% testing dataset. After several runs of training the models with a mix of benign and malicious PE32 binaries, the model will be used to predict the family and class of the binaries. The output of the classification phase is a confusion matrix and other details not limited to training/testing time.

D. Performance Metric Computation

The performance metric computation phase will calculate the necessary performance metrics as indicated in Figure 4. The matrix will be used to calculate False Positive Rate (FPR), Precision, Accuracy, Root Means Square Error (RMSE), F-Measure, and AUC-ROC. Finally, a table of the performance metrics will be created containing the values from the proposed metrics.

V. Conclusion and Future Work

In this paper, we discuss the limits of current detection methods and potential for current machine learning models to identify familial relations within malware samples. We propose a solution with control flow graph data gathered using angr and generative convolutional networks to attribute the malware samples. Future contributions aim to refine the proposed implementation by increasing overall detection rate and time to train through dimensionality reduction; a form of exploratory data analysis not considered at the time of testing.

References

[1] M. Sikorski, A. Honig, and R. Bejtlich, Practical malware analysis: the hands-on guide to dissecting malicious software. San Francisco, CA: No Starch Press, 2012.

[2] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," 31st Conference on Neural Information Processing Systems (NIPS 2017).

[3] CSIRO's Data61, "StellarGraph Machine Learning Library," Github Repository, 2018.

[4] M. Eskandari and S. Hashemi, "Metamorphic Malware Detection using Control Flow Graph Mining," IJCSNS International Journal of Computer Science and Network Security, vol. 11, no. 12, 2011.

[5] Hex Rays. [Online]. Available: https://www.hex-rays.com/products/decompiler/compare/. [Accessed: 05-Nov-2020].

[6] "Intermediate Representation," angr Documentation, 2018. [Online]. Available: https://docs.angr.io/advanced-topics/ir. [Accessed: 05-Nov-2020].

[7] "VEX IR" intermediary representation description, 2015. [Online]. Available: https://github.com/angr/vex/blob/dev/pub/libvex_ir.h