Graph Based Malware Analysis Using NLP Approaches

Chao Xie

Department of Computer Science
University of Wisconsin-Madison, USA
Email: cxie@cs.wisc.edu

Abstract

An increasing prevalence of attacks and intrusions has accompanied the continued growth and diversification of the Internet. Traditional malicious code detection technologies that based on byte- or instruction-level signatures fail to detect modern malicious code due to advanced software obfuscation technologies. In the project, we use graph based approaches to discover the internal structural characteristics of malicious code and adopt graph mining techniques to find the frequent graph patterns. We build the feature vectors using above graph patterns and apply Naive Bayes and SVM to classify these vectors in a vector space.

Introduction

The continued growth and diversification of the Internet is accompanied by the increasing prevalence of attacks and intrusions. There is a significant change in motivation for malicious activity has taken place over the past several years: from vandalism and recognition in the hacker community, to attacks and intrusions for financial gain. Malware, which is software for malicious attacks and intrusions, has also evolved a great deal. The evolution of malware can be seen both in terms of variants of existing tools and in the relatively frequent emergence of completely new codebases.

Most of the malware samples received daily at anti-virus laboratories are variants of previously seen samples. Malware authors develop huge numbers of variants as a way to bypass anti-virus signatures and saturate the anti-virus labs. These new samples share the same code with minor differences, so if we were able to analyze them focusing on their internal structure, identifying the shared code between them, we could group members of the same family together and therefore automatically identify and classify new samples received. Previous research to identify changes among similar pieces of code has been focused on byte- or instruction-level comparison. Although this approach has worked in the past, nowadays it is not effective any more because of aggressive compiler optimizations, instruction reordering, modifications in register allocation, branch inversion, obfuscation methods etc.

Representing code as graphs provides an abstraction that allows us to identify pieces of code shared between binaries, or with minor changes, without having to match instructions as a sequence of lines and applying a sequence-comparison algorithm. In the project, we use graph based approaches to discover the internal structural characteristics of malicious code and adopt graph mining techniques to find the frequent graph patterns. We build the feature vectors using above graph patterns. Then we apply Naive Bayes and SVM to train the malicious code classifier.

System Architecture

Figure 1 illustrates our system architecture. In the following, we will introduce each component of the system in details.

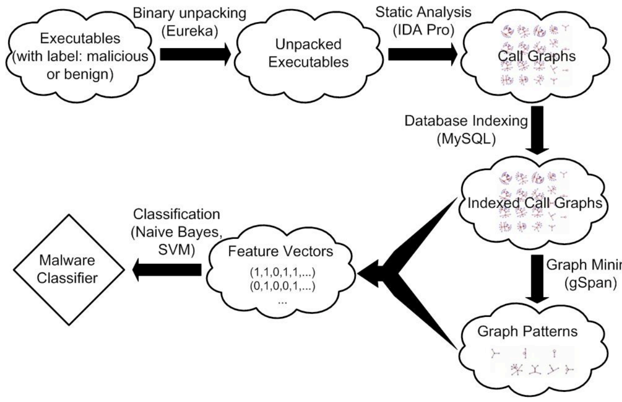

Executables (with label: malicious or benign)
Binary unpacking (Eureka)
Unpacked Executables
Static Analysis (IDA Pro)
Call Graphs
Database Indexing (MySQL)
Indexed Call Graphs
Graph Mining (gSpan)
Graph Patterns
Feature Vectors
(1,1,0,1,1,...)
(0,1,0,0,1,...)
...
Classification (Naive Bayes, SVM)
Malware Classifier

Figure 1: System Architecture

Binary Unpacking

Unpacking is an indispensable pre-processing step for binary code analysis. Packers and executable protectors are often used to automatically add several layers of protection to malware executables. Recent packers and protectors also incorporate API obfuscations that make it hard for analyzers to identify system calls or calls to Windows APIs.

We use the Eureka binary unpacking system to pre-process the executables. Eureka adopts both heuristics-based unpacking and statistics-based unpacking and is very effective.

Static Analysis and Call Graph

Executables could be represented as call graphs, which is a collection of nodes connected by a group of edges, where the nodes represent the disassembly of the functions and the edges represent the relations (function calls) among these functions. For example, an edge from f1 to f2 implies that f1 contains a function call to f2 but not vice versa. A example of call graph is given in Figure 2.

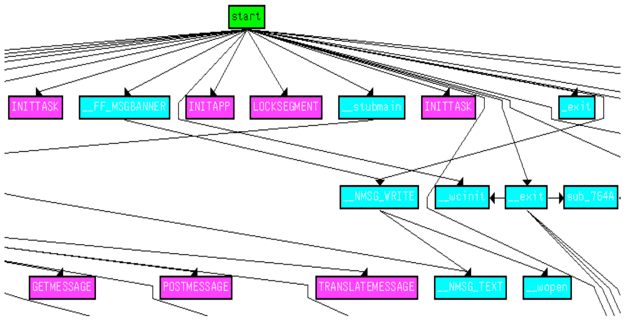
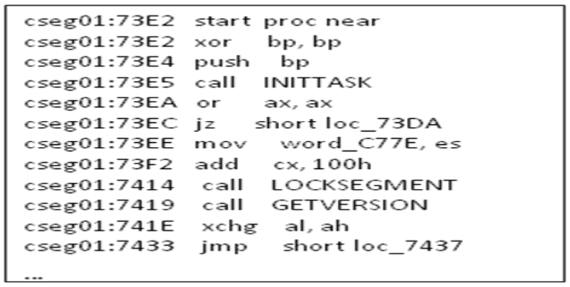


Figure 2: A call graph example

The functions in a program can be classified into three categories, which are represented by different colors in Figure 2. The cyan-colored nodes represent the statically-linked library functions, which are library functions statically linked into the final distributed binary, such as Libc and MFC. The pink nodes represent the dynamically-imported functions that are linked at run time, e.g., windows DLL functions. The green nodes represent the local functions that are written by software authors.

To extract call graphs, we employ the IDA-Pro disassembler to disassemble the unpacked executables. Import and library functions are standard routines, so their names are consistent throughout all the programs. We are use the function name as their node label in the call graph. However, for local functions, their names are in general unavailable because most executables do not come with their symbol table. We use two different approaches to get the label of the local function nodes.



```
cseg01:73E2  start proc near
cseg01:73E2  xor    bp, bp
cseg01:73E4  push   bp
cseg01:73E5  call   INITTASK
cseg01:73EA  or     ax, ax
cseg01:73EC  jz     short loc_73DA
cseg01:73EE  mov    word_C77E, es
cseg01:73F2  add    cx, 100h
cseg01:7414  call   LOCKSEGMENT
cseg01:7419  call   GETVERSION
cseg01:741E  xchg   al, ah
cseg01:7433  jmp    short loc_7437
...
```

Figure 3: Instruction sequence in function "start"

Bag of Word Presentation (BWP). Each local function consists of a sequence of assembly instructions. We treat each instruction, such as xor and push, as a "word type" and count its occurrence. By this way, we can build a unigram vector for each local function and use it as the label.

High Level Function Only (HLFO). We assign a unique label to all the local function nodes. By this way, we treat omit the difference between local functions and only concentrate on high level Import and library functions.

Frequent Graph Pattern Mining

Because a call graph represents the internal structure of an executable, we can suppose that malware samples in the same family should generate similar graphs, or there exists at least graph isomorphism at certain level. Moreover, there could be certain patterns shared by malware samples of the same family. We can suppose that these patterns are preserved in the graphs and use the famous gSpan graph mining algorithm to discover the frequent graph patterns.

Malware Classification

We transform each call graph into a feature vector x = {x_1, x_2, …, x_n}, where x_i=1 if the i-th pattern is contained in that graph; otherwise, x_i=0. Each vector is associated with a class label y, where y=1 if the executable is malicious and y=0 if it is benign. Then we apply Naive Bayes and SVM to classify these vectors in a vector space.

Experiment and Results

We used 1000 malicious executables and 1000 benign executables for the experiment and used 10-fold cross validation.

The confusion matrices of the classification are given below.

Table 1: Confusion Matrix Using Bag of Word Presentation

	Predicted (Naive Bayes)			Predicted (SVM)		
Actual		Mal	Benign		Mal	Benign
	Mal	719	267	Mal	712	296
	Benign	281	733	Benign	288	704

Table 2: Confusion Matrix Using High Level Function Only

	Predicted (Naive Bayes)			Predicted (SVM)		
Actual		Mal	Benign		Mal	Benign
	Mal	782	243	Mal	751	262
	Benign	218	757	Benign	249	738

We can see that Naive Bayes achieves slightly better results in this malware classification task. We also observe that BWP will result in a better classification than HLFO. This may be because of that graphs and feature vectors based on the low-level assembly instructions potentially bring a lot of noise or chaos. However, the accuracy of the malware classification is not desirable at all. We plan to integrate other feature with structure feature to for future improvement.

References

Realms, S. *Armadillo protector*. http://www.woodmann.com/crackz/Packers.htm#armadillo.

Sharif M., Yegneswaran, V., Saidi, H., Porras, P.A. and Lee, W., *Eureka: A Framework for Enabling Static Malware Analysis*, in Proceedings of the 13th European Symposium on Research in Computer Security, Malaga, Spain, October 2008.

Hex-rays. The IDA Pro Disassembler and Debugger. http://www.hexrays.com/idapro/, 2008.

Yan, X., and Han, J. *gSpan: Graph-Based Substructure Pattern Mining*, in Proceedings of the 2002 IEEE International Conference on Data Mining.