Tudor Mihai Avram

Machine learning graph filtering

Computer Science Tripos – Part II

Homerton College

March 23, 2018

Proforma

Name:	**Tudor Mihai Avram**
College:	**Homerton College**
Project Title:	**Machine learning graph filtering**
Examination:	**Computer Science Tripos – Part II, July 2018**
Word Count:	**1**[1]
Project Originator:	Dr Ripduman Sohan
Supervisor:	Dr Lucian Carata

Original aims of the project

Work completed

Special Difficulties

[1]This word count was computed by `detex dissertation.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation explores how machine learning techniques can be applied in order to visualise important parts of large graphs. This problem is particularly important for provenance graphs describing abstractions at the operating system level. When using such graphs for identifying potential attacks on systems, an analyst would need to sift through large and potentially irrelevant parts of the graph in search for suspicious activity. This project aims to provide a tool for helping the analyst in this task. To that end, I implemented a machine learning processing pipeline that classifies nodes as being of interest or not. On top of this classifier, I also implemented a server infrastructure that facilitates communication between the model and the client (see section 1.2).

1.1 Aims & Motivation

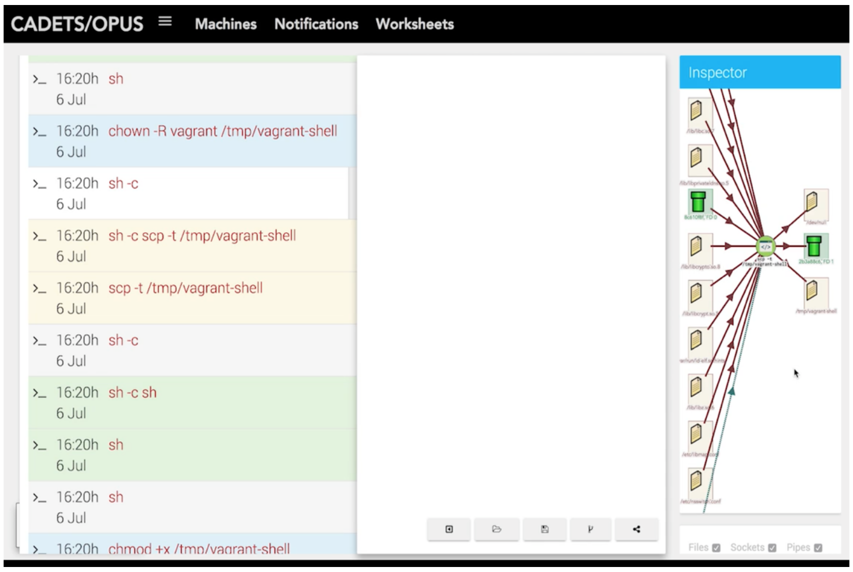
Analysing event logs of a distributed system to identify actionable security information and hence to act accordingly in order to improve a system's security has always been a desirable goal for both research and industry. The availability of cheap data storage has facilitated such large scale logs collection, on the order of several terabytes of data. This makes manual log review an unfeasible task.

Moreover, computer logs are becoming more structured. Using graphs to highlight the relationships between logs introduces a whole new dimension in terms of richness of the information that needs to be explored. The difficulty arises from the fact that graphs cannot be directly provided as machine learning input.

1.1.1 Overview of CADETS UI

The project at hand is an extension of the CADETS user interface, a cybersecurity provenance analysis tool developed as part of the CADETS and OPUS research projects. It displays OS-level abstractions as a network, emphasising the relationships between actors(processes, users) and objects(files, sockets, pipes). It is the analyst's task to explore this network in order to identify potentially suspicious nodes (i.e. nodes describing activities done by an attacker).

In order to provide the analyst with comprehensive data from which he can infer useful information, the network will have a large number of nodes. This makes exploring the entire dataset a very difficult task for a human. For example, tracing two machines for 7 minutes can produce as many as 6,000 nodes in the network. As the number of machines and the time interval we do tracing on increase, this number will become considerably larger, presenting a significant scalability issue when it comes to manual reviewing. Therefore, it is imperative that we perform a filtering on these nodes in order to enhance the analyst's productivity.



```
16:20h sh
6 Jul
16:20h chown -R vagrant /tmp/vagrant-shell
6 Jul
16:20h sh -c
6 Jul
16:20h sh -c scp -t /tmp/vagrant-shell
6 Jul
16:20h scp -t /tmp/vagrant-shell
6 Jul
16:20h sh -c
6 Jul
16:20h sh -c sh
6 Jul
16:20h sh
6 Jul
16:20h sh
6 Jul
16:20h chmod +x /tmp/vagrant-shell
```

Figure 1.1: **Snapshot of CADETS UI**

1.1.2 Aims of the project

In general, the number of nodes that are of interest to the analyst is significantly lower than the number of nodes that are not. Therefore, the aim of this project is to asses the suitability of different machine learning algorithms that would significantly improve the analyst's experience by reducing the number of nodes he would have to manually inspect.

1.2 Overview of the architecture

The main component of the project is a supervised learning algorithm that classifies nodes from the graph as being of interest or not. The model is built using a labelled training set constructed based on a set of predefined ground-truths.

The communications between the CADETS user interface and the machine learning model is facilitated by a REST API(as shown in Figure 1.2). For performance purposes, it also uses a local cache of previous classifications.

When it receives a request from the client, the server would first check the local cache. If it can find the classification result for that node, then it just returns the cached value to the client.

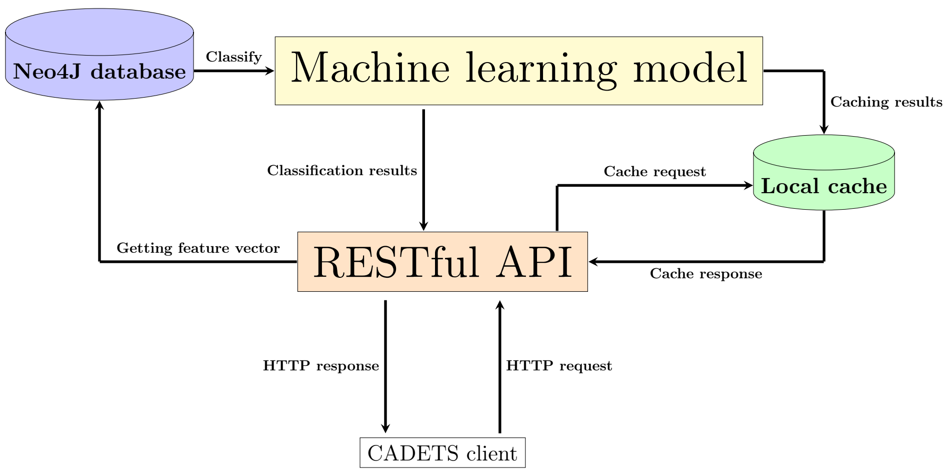


Figure 1.2: **Overview of the project's architecture**

If the result is not in the cache or if the cached value expired, it runs the machine learning model in order to classify that specific node. Once the classification is done, the result is stored in the local cache and returned to the client.

1.3 Related work

Cyber security has always been a major concern in computer science, both in industry and in research. With the recent interest shown in artificial intelligence and machine learning, there has been an increasing number of tools that use related methodologies to identify malicious behaviour. This section will address some of these tools.

1.3.1 Clearcut

Clearcut[1] is an open source tool that uses Machine Learning techniques for incident detection. It takes HTTP proxy logs as input and filters them for manual review, in order to aid the analyst in reviewing them.

The algorithm used in this case is Random Forrest Classification, which essentially means having multiple decision trees in training. The returned class is corresponding to the mode of the classes returned by the decision trees running separately.

Compared to Clearcut, the project described in this dissertation takes the inputs as a graph, thus taking into consideration the logs as well as the relationships between them.

[1]`https://github.com/DavidJBianco/Clearcut`

1.3.2 Polonium

Polonium is a scalable and effective technology that uses graph mining for malware detection. The tool uses a bipartite graph whose nodes describe machines and files as its training data. It uses the Belief Propagation algorithm, which, at a high level, infers the label of a node from some prior knowledge about the node and from the node's neighbours. This is done through iterative message passing between all pairs of nodes (u, v) in the graph.

Although the problem statement is similar to that my project tries to address, the algorithm used here is not applicable in my case, because it requires a very large dataset (Polonium uses a graph with $\approx$ 1 billion nodes). It also requires a pre-computed *machine reputation* score that in this case is calculated using a proprietary formula of Symantec[2], the company that produced the tool.

Polonium works at a much higher level in terms of events compared to what my project attempts. Namely, it uses a significantly coarser grained graphs, with a lower diversity of objects: it only takes into consideration files and machines, while the dataset I used also includes processes, sockets and pipes.

1.4 Licence

The code is publicly available as an open-source project on GitHub[3], under an APACHE 2.0 licence[4].

[2]`https://www.symantec.com/`
[3]`https://github.com/a96tudor/Part2Project`
[4]`https://www.apache.org/licenses/LICENSE-2.0`

Chapter 2

Preparation

In this chapter, I will explain all the work completed before any code was written. This includes a discussion on the structure of the data and the decisions made for pre-processing it (section ??), the theory behind the models that were implemented (section ??) and an outline of the server architecture (section ??) as well as the requirements analysis and software engineering principles applied for a successful implementation(section ??).

2.1 Data analysis

This section describes the raw data used by the CADETS user interface and how it is preprocessed in order to be used by the machine learning models described in section ??.

The data used by the CADETS UI is stored in a Neo4J[1] graph database. This gives a simple and straightforward representation of the OS-level abstractions we want to store as well as of their relationships.

2.1.1 Data structure analysis

The data is stored as a graph, consisting of nodes and edges. Here, nodes represent the actors(processes, users) and objects (files, sockets, pipes, machines), each identified by an unique $(id, timestamp)$ pair. Multiple nodes can represent the same entity, as it evolves over time. Each node is associated a set of features describing the specific actor/ object.

The chart in figure 2.1.1 shows the log-scale distribution of nodes' frequencies. The 'Meta' nodes are associated with processes, representing the initial state a specific process was started in. From the chart, we can easily observe that the 'File' nodes are the most frequent (representing more than 87.4% of the nodes in the graph). Therefore, we can deduce that the data we are working with is highly unbalanced. We have to keep this in mind when designing our model, in order to avoid having a biast classifier (i.e. a model that classifies 'File' nodes correctly and misclassifies the other node types).

[1] `https://neo4j.com/`

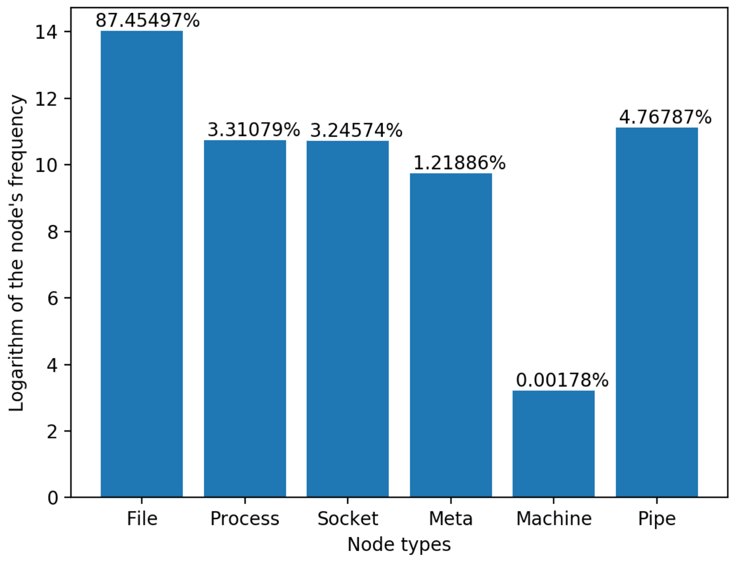


Figure 2.1: Bar chart above showing the log-scale node frequency in a database of $1,402,053$ nodes and $2,090,741$ edges.

Relationships between nodes are illustrated by different types of edges. Some edges also illustrate how an object (File, Socket, etc.) evolves over time. This is done using the *GLOB_OBJ_PREV* edge and helps us to easily visualize the different versions of an object.

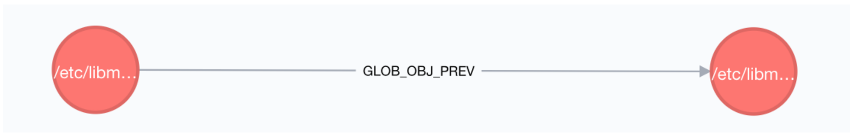


Figure 2.2: Two versions of the same file connected via a GLOB_OBJ_PREV edge.

A number of the edge types also use a *state* field, in order to provide further information regarding the relationship between two nodes. For example, in the case of a *PROC_OBJ* edge connecting a File to a Process, the *state* field is used to show whether the Process reads/ writes to the File or if the File is the binary the Process is executed from. *PROC_OBJ* edges also connect Processes to Sockets. Here, the *state* field can take values such as: *Server* (if the Process uses the Socket to accept new connections), *Client* (if the Process uses to Socket to connect to a different Process that acts as a server - which may or may not be on a different machine) and *RaW* (if the Process also reads and writes through the Socket).

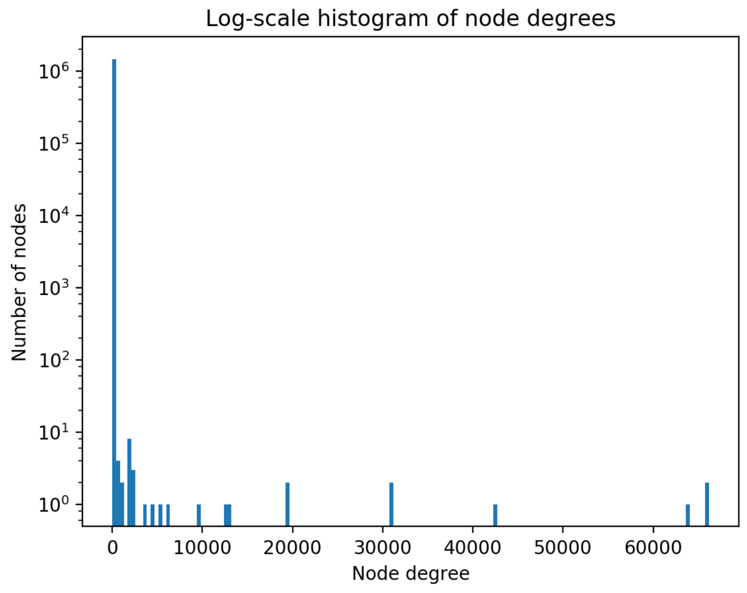


Figure 2.3: Log-scale histogram showing the distribution of node degrees, using the same graph as the one from Figure 2.1.1 as source(i.e. $1,402,053$ nodes and $2,090,741$ edges).

The graphs resulting from tracing are very sparse, with a number of edges almost equal to the number of nodes. This observation is also sustained by the histogram in Figure 2.1.1, where we can observe that most of the nodes have degrees between 0 and 10, while there also are a few nodes with a very high degree (up to $60,000$).

2.1.2 Ground truths

In order to apply appropriate supervised learning algorithms, I had to build a labelled dataset of nodes from the raw data stored in graph format. This required setting a number of *ground truths* of what a node of interest is.

These ground truths are represented by a set of 6 rules, where I took into consideration a subset of node types: Files, Processes and Sockets. The rules are:

1. **Sockets that connect to an external IP:** Any socket that connects to an IP address other than 127.0.0.1 (localhost) can be considered a possible security breach because it could be used to leak information.

2. **Files downloaded from the web and then executed:** Any file downloaded from the web can be suspicious, because we can not trust the source. Especially if it is executed, it can pose a real security threat to the system. The graph representation in the Neo4J database can be seen in Figure 2.

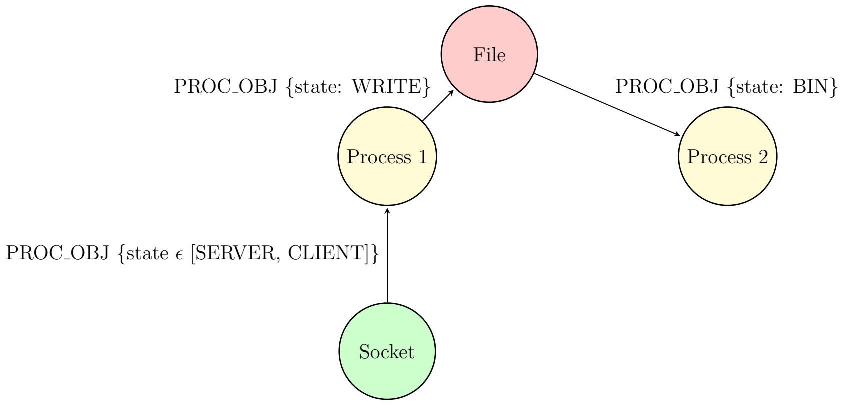


Figure 2.4: Graph representation of a file downloaded and then executed

In the figure above, Process1 writes the File while it is connected to the Socket and then Process2 starts, using File as binary.

3. **File read from/ written to by a Processes that also opens a Socket to a different machine:** Here are two cases we need to treat separately. If the Process reads from the File, we face a potential leak of the File's contents. If the Process writes to the File, on the other hand, it might be the case that it corrupts its contents. This is a potential threat especially if the File is a sensitive file of the operating system (e.g. any file in /lib/ or /bin/).

4. **Processes that open a Socket to a different machine:** As mentioned at point 1, any Socket connecting to an external machine is a potential threat. In the same time, any process that opens a Socket to a different machine can be a source of suspicious behaviour, as well.

5. **Processes that runs suspicious commands:** Here, I define the term of *suspicious command* as being one of the following bash commands:

   - *sudo* - gives the user running the Process root privilege. An attacker might use this to access OS-sensitive locations (such as /bin or /lib).
   - *usermod/ groupmod* - an attacker might make use of these commands to change the running user's privileges and access files that it wouldn't otherwise have access to.
   - *chmod* - an attacker might use this command to change the access control to a specific File in order to make it accessible from external sources.
   - *rm -rf* - an ill-intended user might use this command to delete files crucial to the system.

6. **Processes that writes to files in suspicious locations:** Here, I define the term of *suspicious location* as being a location that is essential to the running of the system. These locations include: */bin, /etc, /lib, /usr/bin, /usr/lib, /boot, /root, /dev, /etc/pwd.*

2.1.3 Data preprocessing

The implemented machine learning models only look at 3 of the 6 types of nodes: **File**, **Process** and **Socket** nodes.

In order to achieve this, I had to define a set of features that would construct the *feature vectors* representing each node. Furthermore, for graph-specific models (such as Graph Attention Networks, described in section 2.2.4) I also built the adjacency matrix of the given graph.

I defined 13 features that would describe each node, in the same time taking into account its 'neighbour'. For a Process, the neighbour is the closest File or Socket connected to it. Here, *'the closest node'* is the node that with the closest timestamp to it. Similarly, for a File or Socket, the neighbour is the closest Process node connected to it.

Feature	**Explanation**			Type
	Process	Socket	File	
Node type	The type of node used in this case (i.e. Process, Socket or File)			**Categorical**
Neighbour type	The closest node connected to it. Has to be one of File and Process	Will always be Process		**Categorical**
Edge type	The type of edge connecting the node to the neighbour. It will always be a *PROC_OBJ* edge, but the *state* field can take multiple values, such as: *READ, WRITE, RaW, BIN, SERVER, CLIENT, NONE.*			**Categorical**
Connected	Whether the Process in question connects to a different machine via a Socket.	Whether the Sockets connects to an IP address other than 127.0.0.1	Whether the File was downloaded from the web or not	**Binary**
Neighbour connected				

2.2 Machine learning models

2.2.1 Baseline

2.2.2 Multi-perceptron model

2.2.3 Convolutional Neural Network

2.2.4 Graph Attention Network

2.2.5 Non-parametric techniques

2.3 API architecture

2.4 Requirements analysis

Chapter 3

Implementation

Chapter 4

Evaluation

Chapter 5

Conclusion