IMPROVING JACKDAW: ALGORITHM LINEARIZATION, STREAMING AND VISUALIZATION

FEDERICO ALEOTTI

Relator: Prof. Stefano Zanero

Correlator: Ing. Mario Polino

Politecnico di Milano
April 2016

Cleverness is no guarantee of sensible behaviour.
— Last argument of kings

Assume nothing: you don't know until you know.
— City of stairs

ABSTRACT

In a growing malware panorama, automatic analysis systems are becoming essential. Jackdaw aims to be a global analysis tool based on both static a dynamic analysis techniques which extracts high-level behaviours from malware binaries, operating as an on-line service to which analysts can submit binaries to be analysed. Furthermore, Jackdaw is meant to allow the creation of a malware map that makes possible to visualise and understand the malware evolution in time, automatically based on the malware analysis results.

My goal consists of improving the existing framework by decreasing the temporal complexity of the analysis from NP to at least polynomial, restructuring the system to work as a stream service and creating a dynamical, real-time malware map.

ACKNOWLEDGMENTS

My first thanks are for Stefano, who inspired me with his lessons, who taught me a lot, pushed me when I needed to be pushed and who was always there when he was needed. He is a great professor and a great friend.

Not less important in the least, i have to thank my two IT friends: Daniele and Francesco. Both taught me a lot and helped me through the most difficult part of my career. They made my study place home.

And finally, i have to thank Mario, who supervised me during my thesis, had and endless flow of suggestions and steered me on the right track when I needed it.

Thank you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Malicious software, often referred to as malware, is one of the major threats to computer systems. Malware has become the main mean through which cybercriminals infect computer systems to gain access to sensitive information, turn computer into bots, and monetize by re-selling the computational and networking capabilities of infected computers for malicious purposes (e.g., spam sending, malware distribution, denial of service). The evolution of malware is a tangible effect of the emergence of a profit-driven under- ground economy. There are even real pay-per-install marketplaces where malware authors can pay to have their malicious software installed on victim machines, via social engineering, spam, and other infection vectors [4].

The industry of malicious software is evolving and new malware samples are frequently developed and deployed. In addition, the existing samples are constantly updated. Hence, a wide range of malware variants are spread around the world, and every day anti-malware companies receives huge amounts of new malware samples that must be analyzed in order to create signature for their detection. A signature is a distinctive feature of a malware sample (e.g.m sequence of bytes). However malware authors became skilled to evade signatures, changing their malicious code to avoid detection. For this reason, it is very important to analyze each malware sample to observe their behaviors. A behavior is a sequence of events (e.g., spam sending, adding a rule to a firewall, download and execute a file). An analyst can define and name behaviors manually (e.g., as a sequence of function names, or system events). Unfortunately, there are not enough analysts to dissect all these malicious programs, so there is a need to automate analysis procedures and increase malware detection rate.

The two main techniques to analyze (malicious) programs automatically are static and dynamic analysis. Static analysis works on the code (e.g., machine, assembly, source). It has the advantage of reaching high code coverage and scalability, but code obfuscation or packing may render it ineffective. Obfuscation techniques aim to confusing the executable code to make it difficult to analyze (e.g., to disassemble) while preserving its functionality. Packing techniques encrypt or compress the executable code. Dynamic analysis observes the execution flow of a running program. Usually, dynamic analysis techniques are used to search known malicious behaviors into a program. These techniques are effective even if the malware adopts obfuscation or packing. However, their code coverage is low, as pure dynamic analysis techniques only record the events observed during execution. Indeed, some evasive malware families change their behavior when executed in a monitored or debugging environment. Thus, one of the main problems in malware analysis is to automatically define interesting (malicious) behaviors.

Jackdaw is a tool to analyse malware that uses a novel approach to extract behavior specifications though a combination of static and dynamic analysis techniques. In a first phase, Jackdaw identifies behaviors, expressed as groups of APIs invoked by the malware. To this end, Jackdaw applies a clustering procedure on data obtained by static analysis, exploiting taint dependencies obtained through dynamic analysis. Then, Jackdaw builds a model of each cluster by means of representative API functions. The resulting models are the extracted behaviors. Last, Jackdaw attaches a semantic meaning to these behaviors. To this end, Jackdaw crawls StackOverflow (a questions-and-answers platform on a wide range of topics in computer programming) to link function names and semantic tags. Using this knowledge, Jackdaw selects the most relevant tags for each behavior and assigns a name to it.

The goal of this thesis is to bring Jackdaw on a whole new level: from malware analysis tool to be used by single analysts, to a shared tool to be used by a community. Jackdaw will be restructured in order to work as a streaming service so that analysts could just submit their malware samples and obtain the extracted behaviors. Also, Jackdaw will benefit from the amount of data it will receive since it will allow it to perform better clustering and better generalisation.

But in order to do that a lot of performance problems will have to be addressed, starting from the algorithm used in the generalisation step, which is currently an np-hard problem but needs to be reduced to polynomial time complexity in order to manage a greater input flow. This will be the most challenging part of this work: develop a new algorithm to generalize the API calls graphs in polynomial time, possibly a low-order-polynomial time. Generalisation is currently the most time-consuming step, so complexity needs to be reduced in order to allow Jackdaw to work as a streaming service. Details in the algorithmic problem can be found in section 2.1.3. Another big problem will be data representation, since a lot of additional data will need to be loaded thus increasing the spatial complexity[1].

In order to convert Jackdaw into a streaming service the whole structure must be reviewed. This will require a deep understanding of the program structure, capability to see the big picture and to reshape Jackdaw in order to keep the data necessary to add new analysis results to an already existing panorama of clusters and extracted behaviours.

1 Clusters get bigger the more samples are submitted, and need to be loaded every time the clustering step occurs

Finally, the huge amount of data available to this new Jackdaw will be exploited in order to build a malware map, i.e. a framework that will allow analysts to see the overall results of Jackdaw analysis: the relations between malware implementations and extracted behaviours via fps. Building a visualization infrastructure will be also a conceptual challenge: how can such a huge amount of data be represented in an undestrandable way? And it would be better if that map could evolve as new malware samples are submitted, instead of being periodically rebuilt. This will mean to create a live map, which is an additional challenge.

2 STATE OF THE ART

In this chapter a brief review of the literature will be provided, and the details of Jackdaw's functioning will be presented.

2.1 JACKDAW

Jackdaw is inspired by previous attempts to automatize the malware analysis process, and its primary goal is to be a fully automated tool for malware analysis.

For example, The work presented by Comparetti et al. [8], called Reanimator, can find different implementations of the same behavior to unveil the functionalities that are not exhibited during dynamic analysis by a malware, but it needs the behaviors to be manually specified by an analyst.

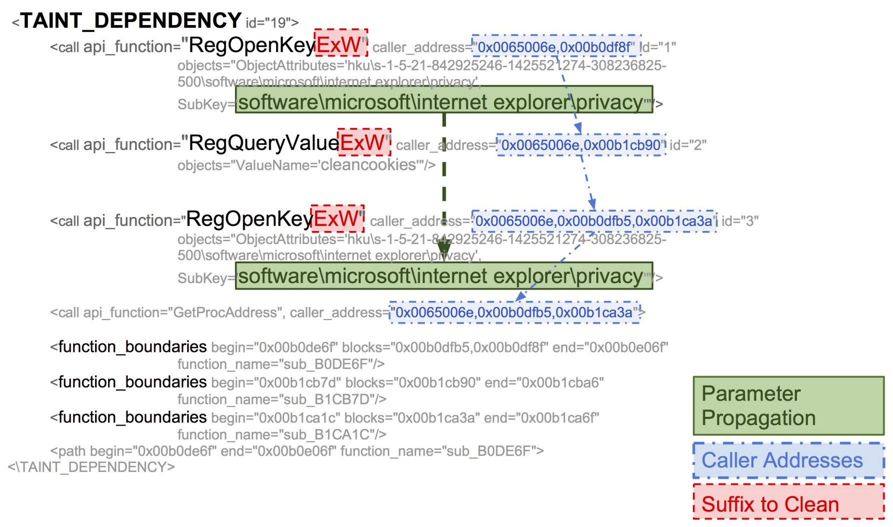
Along the line of [8], Lindorfer et al. [5] observe that malware authors regularly update their software in order to beat defenses, improve their capabilities or change their business model. For this purpose, the proposed system, called Beagle, regularly downloads new versions of the same malware, then compares new versions with the old ones through a series of static and dynamic analysis techniques in order to track its evolution. What is interesting is that Beagle needs manually-defined rules t identify behaviors.

Jackdaw, on the contrary, extracts behaviors in a fully automatic way, operating in four steps. In this section will be provided details on each step, highlighting which improvements are intended to be introduced by this work.

2.1.1 *Step 1: Data Collection*

To extract fingerprints and API functions call from raw data for each instance of a malware only the taints that contain fingerprints must be selected. At the taint level (Figure 2.1), the interest is in API functions, with their parameters, and fingerprints.

Extracting fingerprints is simple and for each fingerprint in a taint Jackdaw only takes the hash that represents it as described in [3]. Instead, a normalization process is needed in order to extract API functions.



```
<TAINT_DEPENDENCY id="19">
    <call api_function="RegOpenKeyExW" caller_address="0x0065006e,0x00b0df8f" id="1"
                objects="ObjectAttributes='hku\s-1-5-21-842925246-1425521274-308236825-
                500\software\microsoft\internet explorer\privacy',
                SubKey='software\microsoft\internet explorer\privacy'"/>
    <call api_function="RegQueryValueExW" caller_address="0x0065006e,0x00b1cb90" id="2"
                objects="ValueName='cleancookies'"/>
    <call api_function="RegOpenKeyExW" caller_address="0x0065006e,0x00b0dfb5,0x00b1ca3a" id="3"
                objects="ObjectAttributes='hku\s-1-5-21-842925246-1425521274-308236825-
                500\software\microsoft\internet explorer\privacy',
                SubKey='software\microsoft\internet explorer\privacy'"/>
    <call api_function="GetProcAddress", caller_address="0x0065006e,0x00b0dfb5,0x00b1ca3a">
    <function_boundaries begin="0x00b0de6f" blocks="0x00b0dfb5,0x00b0df8f" end="0x00b0e06f"
                function_name="sub_B0DE6F"/>
    <function_boundaries begin="0x00b1cb7d" blocks="0x00b1cb90" end="0x00b1cba6"
                function_name="sub_B1CB7D"/>
    <function_boundaries begin="0x00b1ca1c" blocks="0x00b1ca3a" end="0x00b1ca6f"
                function_name="sub_B1CA1C"/>
    <path begin="0x00b0de6f" end="0x00b0e06f" function_name="sub_B0DE6F">
<\TAINT_DEPENDENCY>
```

Figure 2.1: This is an example of taint that shows parameters propagation and API function names cleaning.

2.1.1.1 *API Function Name Normalization*

For API functions extraction, the namespace of the functions must be taken into account carefully. With namespace is meant that, according to the Windows API documentation [7], suffixes such as 'A','W','Ex' in the function name specify the mode in which execute the call (in particular they represent ANSI, Unicode and the win32 version of each function that could add specific parameters), without changing the semantic. These suffixes are not important for Jackdaws's analysis, because the API functions must be generalised as much as possible in order to create our behaviors, and then only in the semantic of the API functions matters.

> Example: 'CreateEvent' function (that creates or opens a named or unnamed event object) has the three form 'CreateEventA' (ANSI), 'CreateEventW' (Unicode) and 'CreateEventEx'. All of them have been renamed as 'CreateEvent'

Similarly, there are also prefixes such as 'WSA' and 'AFD ' that must be discarded in this analysis, because both of them refer to the same windows socket API.

> Example: 'socket' / 'WSASocket' are the same function that creates a socket that is bound to a specific transport service provider. Jackdaw represents it as 'socket'.

For each function, Jackdaw also extract the parameters, each divided in name and value. The problem here is that parameter names were not always meaningful, in the sense that in many cases the name of the parameter is generic like 'ObjectAttribute' and does not give a semantic of that parameter. In other cases they represent exactly the meaning of the parameters, e.g., 'ForeignPort' for the API connect represents the parameter with which specify the port of the connection.

2.1.1.2 *Cuckoo*

The fps extraction is provided by Cuckoo [2], a malware analysis system able to perform exactly the operation described above.

2.1.2 *Step 2: Clustering of Taint Information*

The goal of this step, is to group taints by behavior similarity. Jackdaw can exploit similarity based on colored CFG [3]. This representation of the code takes care of structure of basic blocks and system calls in it. So these fingerprints have inside all the information that Jackdaw needs to have a meaningful clustering. To this end Jackdaw uses a fast, one-pass algorithm for dynamic clustering of an input stream of data (Algorithm 2.1). It associates an item to the cluster with the highest similarity. The similarity among an item and a cluster is computed with single linkage policy. It is defined a threshold (s) of similarity that determines if an item is enough similar to be inserted in a cluster or if it has to be considered as a new cluster. In order to use Algorithm 2.1, a similarity measure is needed. From the result of our validation [9] has been concluded that Jaccard Similarity (2.1) is well suited for Jackdaw's purposes.

$$J(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|} \tag{2.1}$$

Equation 2.1 is computed on fingerprints. Each taint is viewed as a set of fingerprints.

2.1.3 *Step 3: Behavior Extraction*

In this phase Jackdaw extracts rules that characterize each cluster obtained with the previous step.

2.1.3.1 *Most Frequent Rule API Functions Extraction*

This is the first heuristic Jackdaw uses in order to extract the API functions that represent a cluster. It is based on the frequency of the API functions, i.e., an API is representative of a cluster if it appears more than n percent of taints belonging to that cluster. It has been chosen n = 0.70 empirically, as a result of an experiment [9]. The base shape of MFR is a conjunction of API fun

$$B_1 = API_1 \wedge API_2 \wedge \cdots \wedge API_n \tag{2.2}$$

Algorithm 2.1 Clustering algorithm based on ECM [10]: s is max of Jaccard similarity between t and all taints in the cluster

```
Input: taintset, clusterset={c[1..l]}
for all t in taintset do
    for i in {1..l} do
        s[i] = s(t,c[i])
    end for
    i∗ = argmax(s[i])
    if s[i∗] > threshold then
        c[i∗] .add(t)
    else
        new c[l+1]
        c[l+1].add(t)
        clusterset.add(c[l+1])
        l = l+1
    end if
end for
```

The Equation 2.2 means the behavior of the taint is B1 if each APIx is in the taint.

2.1.3.2 *Propositional Logic Rule API Functions Extraction*

There can be API functions that act in the same way but have different names.

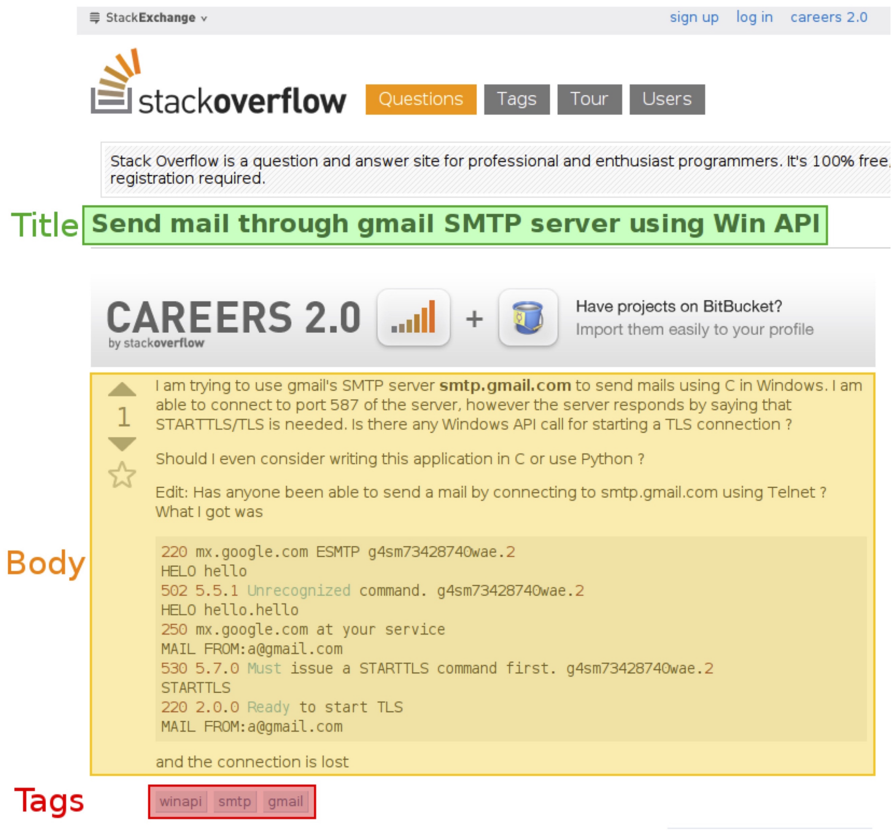
The basic shape of PLR is like MFR, but PLR also exploits the correlation between API functions presence in taints to identify system calls that have same effect, with the assumption that cluster contains only taints with the same behavior.

PLR heuristic uses a logic expression with APIs as propositions to define a behaviour, allowing thus more expressive power in defining it for example including conditional APIs, i.e. APIs with different names that act in the same way.

2.1.3.3 *Improving generalization*

What MFR and PLR both fail to keep into account are the relations between API calls into the same cluster: a cluster is not a set of API calls, but a graph of API calls connected to each other by directed graphs that represent the data flow between them, and this information should be kept into account in order to obtain a better generalisation and thus a more precise behaviour extracted. Furthermore, it doesn't rely anymore on the assumption that a cluster contains only taints with the same behavior, which is not easily proved.

A few steps in this directions were already taken, but the problem is that generalising graphs essentially means to extract the maximum



Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, registration required.

Send mail through gmail SMTP server using Win API

I am trying to use gmail's SMTP server **smtp.gmail.com** to send mails using C in Windows. I am able to connect to port 587 of the server, however the server responds by saying that STARTTLS/TLS is needed. Is there any Windows API call for starting a TLS connection ?

Should I even consider writing this application in C or use Python ?

Edit: Has anyone been able to send a mail by connecting to smtp.gmail.com using Telnet ? What I got was

```
220 mx.google.com ESMTP g4sm73428740wae.2
HELO hello
502 5.5.1 Unrecognized command. g4sm73428740wae.2
HELO hello.hello
250 mx.google.com at your service
MAIL FROM:a@gmail.com
530 5.7.0 Must issue a STARTTLS command first. g4sm73428740wae.2
STARTTLS
220 2.0.0 Ready to start TLS
MAIL FROM:a@gmail.com
```

and the connection is lost

winapi smtp gmail

Figure 2.2: This is an example of posts as result of "connect port 25" researches. Here is highlighted the elements that Jackdaw uses for its analysis.

common subgraph, which is notoriously an np-hard problem. One of the goals of this thesis will thus be to develop an algorithm to generalize the API calls graphs of each cluster in polynomial time.

2.1.4 *Step 4: Semantic Tagging*

Jackdaw uses StackOverflow, a famous question and answer site for programmers, to allows crawler to extract semantic data related to the generalized API calls: for each element of the power set of API functions of each behavior the crawler searches StackOverflow questions that contain that element.

> Example: this is an example of behavior: DnsQuery, Socket, LoadLibrary, Connect(port:25,IP: public) The crawler searches each element of that behavior and all its subsets, like (DnsQuery,Socket), (Socket,Connect(port:25,IP: public)), etc.

From each post (referring to Figure 2.2) title, body and tags are extracted. Then Jackdaw analyses post tags. A tag, according to Stack overflow definition,"is keyword or label that categorizes a question with other, similar questions. Using the right tags makes it easier for others to find and answer your question".

Jackdaw extracts from all the posts related to a behavior tags (Figure 2.2) and give to them a vote. Finally, it applies majority votes technique to determine the tags that represent hints.

A score is computed for each post in order to understand if it's a post that is really related to that specific API or not. In order to do that, three list of words have been created. These lists represent:

- Interesting words: a positive score is given to the post for each word of this list contained in it; this list contains words like windows,winapi that suggest that post is related to Windows API.
- Trifling words: a negative score is given to the post for each word of this list contained in it; this list contains words like php,python.. that means that the post is related to a specific programming language and not to the API.
- Blacklist: in this list have been put the words that is not interesting having into the hints list, because they do not describe the API.

Given those lists, the score of the post is computed as the sum of points collected by itself tags. A post is marked as important if it has a positive (greater than or equal to zero) score. Thus Jackdaw weights each tag with the post score as shown:

$$tagScore_{i,j} = (postScore_i * \frac{1}{n}) * isTagInPost_{i,j}$$

where $\mathrm{isTagInPost}_{i,j} \in 0,1$ represents if the post ith contains the tag jth and n is the number of posts related to that behavior; the vote for each tag is computed as the sum of tag score related to that behavior as below:

$$tagVote = \sum_{i=1}^{n} tagScore_{i,j}$$

where n is again the number of posts related to the behavior.

3 PROBLEM ANALYSIS AND SOLUTION DEVELOPING

In this chapter the specific problems of this work will be detailed and analyzed, and the solution development will be reported step by step.

3.1 GENERALIZATION ALGORITHM

The first problem to be solved is how to perform a better generalization of the API calls graphs: after clustering on the FPs, each cluster is a collection of API calls graphs, theoretically similar to each other. It is necessary to generalize this collection in order to isolate the API calls that better characterize the cluster. The problem is, thus, a common subgraph problem between all the graphs of the cluster. The common subgraph problem is a well known problem in information technology, and not an easy one: it is a renowned NP-hard problem.

3.1.1 *Problem analysis*

In this section the problem of generalization will be detailed and analyzed.

The goal is to find the maximum common subgraph between a bunch of graphs, keeping into account a subgraph tolerance. The first thing that must be defined is what is the maximum common subgraph. With maximum common subgraph is intended the subgraph with the highest number of nodes and edges among common subgraphs. But what if this subgraph is not connected, i.e. it's composed by two subgraphs? This case needs to be kept into account, thus the common subgraph is a multigraph

APPROXIMATE SOLUTION The goal consists not only in solving the problem of generalization, but especially in doing it in polynomial time because a higher temporal complexity time is too much for the program to process the expected workload. The solution will thus be an approximate one and, if possible, it should be a low-order polynomial one, since we expect a lot of graphs in each cluster and each graph has a lot of nodes.

The temporal complexity should not be measured w.r.t the number of graphs, because graphs are complex objects often made of a large number of nodes, which could be processed several times each. Thus, a temporal complexity based on the number of graphs would be deceiving.

The temporal complexity would be better defined considering the number of edges, as it is usually done with graphs, but such a complexity would not consider node operations. The complexity will thus be defined on the total number of nodes, i.e. adding up the number of nodes of each graph, plus the total number of edges, i.e. adding up the number of edges of each graph.

SUBGRAPH TOLERANCE Since the clustering is made on the FPs, and the generalization is performed with regard to the API calls, it is possible to find in some clusters an API graph which is really different from the others in the cluster. The probability of this is expected to be low, since the whole work is based on the assumption that FPs are strongly related to the correspondent API calls, but the possibility needs to be considered, since a single different graph in a cluster can produce an empty common subgraph. In order to address this problem a subgraph tolerance must be introduced: the common subgraph needs to be computed excluding a small percentage of graphs from the cluster graphs. However, excluding graphs may still be not enough: some graph features can be common features between the most of the graphs in the cluster, but not all, and this features can be spread non homogeneously between the graphs. Let's suppose we have five graphs: 1, 2, 3, 4 and 5. Feature A is common between 1, 2, 3 and 4. Feature B is common between graphs 2, 3, 4, and 5. Feature C is common between 1 and 2. Which features should be included in the common subgraph, considering a tolerance of 80%? Feature A and feature B, because they're shared between the 80% of the graphs. But this cannot be achieved by simply removing the 20% of the graph, i.e. one graph. Thus, the tolerance must be considered explicitly while building the common subgraph: each node must be included in the common subgraph if and only if it's shared between at least the 80% of the graphs, and the same for the edges.

EXISTING SOLUTIONS The first step toward finding a solution to the generalization problem has been analyzing existent ones. The real problem in doing this was the subgraph tolerance, since it is not commonly addressed in solving the common subgraph problem. The most promising paper is Mining Significant Graph Patterns by Leap Search [11]. The algorithm proposed has very hight time complexity, too hight to be used, but the paper provided an interesting insight: the main idea was to reduce the number of the graphs to be checked in order to reduce time complexity.

We used this idea in our own solution.

PROBLEM ANALYSIS ON GENERIC GRAPHS In order to work on graphs, a representation system must be adopted. Graphs are usually defined as a set of nodes and a set of edges, where each edge connects two nodes. Viewing graphs as sets opens up an interesting perspective: operations on sets are fast, polynomial operations, so why it's so complex to operate on graphs? The problem is in the edges: each edge point to two nodes, thus each edge can be seen as a set of nodes, but a single node can be contained in multiple edges. Also, nodes and edges are indistinguishable from each other, making a matching between two graphs a really hard operation. The real problem emerging from this analysis is the topological information: if you consider the graph just a set of nodes and edges not linked to each other, the problem becomes simpler by some orders of magnitude.

PROBLEM ANALYSIS ON SPECIFIC GRAPHS Since the graphs the algorithm need to generalize are not generic graphs, but specific API call graphs deriving from the Cuckoo analysis, it is worth to analyze a sample of them to verify if some simplifying assumption can be made. From this analysis two main features emerged:

- In API calls graphs each node is thus characterized by a label, containing the name of the corresponding API call, and some attributes with the API call parameters.
- The API calls graphs have a relatively low number of edges w.r.t the number of nodes.

3.1.2 *Solution development*

In this section a solution will be developed basing on the key elements emerged from the previous section's analysis.

DEFINING A TOLERANCE THRESHOLD Defining a threshold for the subgraph tolerance is not an easy task: how can a fitness function be defined for the common subgraph depending on the tolerance? Fortunately, if our assumption is correct, i.e. there's a strong correlation between FPs and API graphs, the number of "strangers" in the graph population will be small. And if this number is small, even if a bunch of correct graphs are excluded, generalization shouldn't be lost in the common subgraph, since the features are almost universally shared.

> Thus, the precise value of the threshold is not so relevant provided that a threshold exists (Assumption 0).

BASIC SOLUTION IDEA: PRUNING The operation that must be performed on the graphs can be viewed as a search for the maximum common subgraph into the space of the possible subgraphs. In order to find a possible subgraph, a search into the space of possible graphs obtained using nodes and edges from the graph population must be performed.

As seen into the Mining Significant Graph Patterns by Leap Search paper, reducing the search space is the key to reduce complexity. From artificial intelligence, the easiest way to perform a search in some space with low complexity is to perform some sort of pruning.

Pruning is an operation usually performed on search trees, which decides, basing on some heuristic, that some branches need no further exploration, for example because you can prove that any result found exploring that branches is suboptimal. In order to prune effectively an heuristic is needed to accept or discard candidates, and it must be a low complexity one. Furthermore, this heuristic must keep into account the subgraph tolerance.

THE CANDIDATES GENERATION PROBLEM The solution idea is to apply a pruning heuristic to the common subgraph search space, i.e recursively reducing the number of possible candidates to be the maximum common subgraph, starting from all subgraphs of all graphs as candidates.

To further reduce the time complexity, instead of explicitly generating the candidates the pruning concept is applied to the original graphs, removing nodes and edges that can't be in the maximum common subgraph, and further reducing time complexity. Then the pruning can also be applied to the whole graphs which subgraphs cannot be the maximum common subgraph, removing them from the search space. So the goal is now to find an heuristic to determine if a node can be included in the solution or not, and the same for the edges. And then an heuristic to determine if a graph can be declared certainly not part of the solution, so that we can reduce the number of graph to be pruned.

HEURISTIC IDEA 1 If graphs are represented as a set of nodes and a set of edges, it is straightforward to see that the common subgraph must have as nodes the intersection of the sets of nodes. We do not, of course, perform a topological match since the complexity is NP. Instead, we perform a match on API labels (if more than one node in a graph can have the same label, we consider them indistinguishable). With this heuristic all the nodes which label is not in the intersection of node label sets can be pruned from each graph, along with all its edges. To keep into account the subgraph tolerance we adjust the definition as follows:

> "A node label is in the intersection if it appears in at least the threshold percent of the graphs in the cluster."

In this way only the nodes which labels are in the intersection are kept.

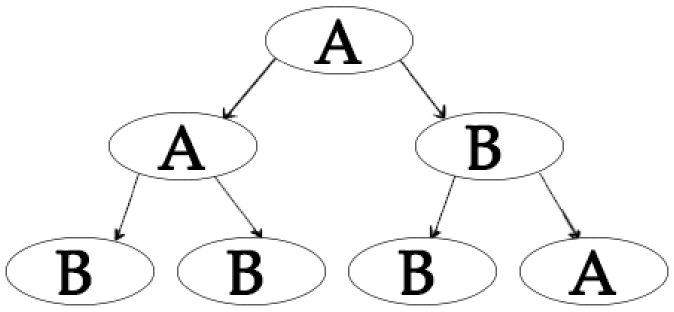
HEURISTIC IDEA 2 Similarly, we can quickly prune edges based on the labels of the nodes they connect. Since edges are directional, we use the concatenation of the node labels to represent the edge label, and then apply the same reasoning of heuristic 1 to edges: an intersection of edge labels can be defined, keeping into account the subgraph tolerance, as:

> "An edge label is in the intersection if it appears in at least the threshold percent of the graphs in the cluster."

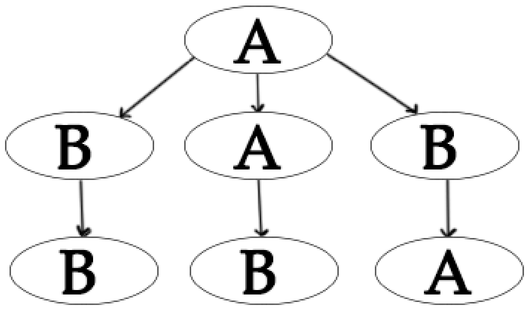
heuristic idea 3 Heuristics one and two prune graphs by removing all edges and nodes that are not part of the intersection. Still, an arbitrary number of nodes and edges remains, with no way to decide which ones to keep. The problem is evidently the lack of topological information, so an heuristic that incorporates some topological information must be found, even if it means to sacrifice linear complexity.

We tried using specific features of the cluster's graphs, instead of considering the graphs as generic. In the cluster's graphs the edges are directed, and this fact can be used to describe topology by simply representing a father-son relationship between nodes. Exactly as before, the problem is how to match nodes between graphs in order to understand which node has to have which child. The API label can be used to partially solve this problem: a count of the child nodes can be kept for each node label. For example, considering graph (a) in Figure 3.1, A-nodes have a total of 1 child of type A and 3 child of type B, while B-nodes have a total of 1 child of type A and 1 child of type B. This information can be used to compute the number of child of each node label that the common subgraph must have, and then this number can be used to exclude graphs that have less child than the solution, since the solution can't be included in them. The problem is that this information cannot be used to prune graphs, and also the topological information contribution is really low, because, for example, graph (b) in Figure 3.1 has nothing to do with the graph above, still the child count is the same exact as before.

A further improvement of this heuristic can be obtained by representing child not with their total number but with a set containing the number of child for each node label. Taking graph 1 as an example, A-nodes would have {1,0,0} child of type A, and {0,2,0} child of type B, while B-nodes would have {1,0,0,0} child of type A and {0,0,1,0} child of type B. Confronting it with graph 2, which A-nodes have {1,0,0} child of type A and {2,0,0} child of type B, and which B-nodes have {1,0,0,0} child of type A and {0,1,0,0} child of type B, it's evident that the only difference is in the order, i.e. the topological distribution, of the child. Even though an univocal ordering of the child can be introduced, like the one obtained by exploring deep-first the graph, the problem of finding the root node remains: in the examples simple trees have been used for simplicity, but there's no guarantee of the cluster graphs being trees. So the child computation should



(a)



(b)

Figure 3.1: Even if the two graphs are notably different, the label count is the same for each one of them.

be repeated for each possible root node, increasing significantly the complexity. (And this excluding the case, possible but not present in the sampled graphs, of not having any root node, i.e. not having a node without incoming edges).

It's evident that the more topological information is included, the highest the complexity becomes. So the problem becomes a tradeoff between complexity and topological information. Even worse, the topological information gain upon raising the complexity is very low. In order to define a tradeoff we would need to switch to an experimental approach, but doing that will mean define a quality measure, which is not an easy task.

Before moving to the experimental phase, we decided to try to exploit the topological information already available, implicitly contained in the solution candidates. The main problem in exploiting this fact is that an heuristic to prune nodes or edges from our candidates is not available, except for the first two. But heuristics to prune graphs from the search space can be created, as proved with heuristic 3. So we tried to build an heuristic able to isolate a single candidate by pruning graphs from the search space.

HEURISTIC 4 We have to find an heuristic to quickly exclude a big amount of graphs. Even better if that heuristic could just pinpoint the right candidate among the others.

The first problem is that the common subgraph is not included in every single cluster's graph, because of the subgraph tolerance, so we must find a way to exclude all the graphs that cannot contain the solution.

In fact, the common subgraph could be fully contained in no graph, but let's assume for a moment that there is at least one graph which fully contains the common subgraph. This assumption will be referred as Assumption 1.

The easiest way to do that is to compute the number of nodes of the common subgraph by counting the nodes of all graphs and selecting the lowest number after considering the subgraph threshold. Then, all the smaller graphs can be deleted. And this process can be repeated for edges too. This idea can be improved by considering the information obtained by applying heuristic 1 and 2: instead of computing the total number of nodes of the common subgraph, we compute the number of nodes of the common subgraph for each node label. (Heuristic 1 already needs to scan the nodes in order to compute which node labels have to be pruned). The subgraph threshold is considered by computing for each graph the total number number of nodes with that label, and then choosing among these numbers the highest number which relative frequency is at least equal to the subgraph tolerance threshold. Unfortunately this heuristic can be used to prune graphs but not nodes, because even if a graph is known to have too many nodes labeled for example A, there is no way to decide which one to prune and which one to keep.

The really interesting point is that this heuristic can be applied in linear time too (w.r.t the number of nodes): with a single scan of all the nodes it is possible to fill in a list, for each graph, with all node labels, and for each label progressively increment the node counter. An example of such a table can be found in Table 3.1.

	Graph1	Graph2	Graph3
Label A	2	0	3
Label B	5	6	4

Table 3.1: This table can be computed with just a single scan of the nodes of all graphs, i.e. in linear time.

The final operation of extracting the number of nodes for each label requires only to sort the number of nodes of each graph for that label, and since it's an integer sorting, it can be performed as a counting sort, keeping the complexity linear.

Again, as with nodes, the number of edges with each label that must be part of the intersection can be computed. The process is identical to the one presented for nodes, so the details will not be provided. As before, the whole operation can be made in linear time, this time w.r.t the number of edges.

APPLYING HEURISTIC 4 Heuristic 4 is able to determine the number of nodes and edges that the common subgraph must have, for each label, keeping the subgraph tolerance into account. This heuristic need to be used to isolate the most-promising candidate among all pruned candidates. In order to do that, all graphs which have a number of nodes or edges smaller than the computed one, even if for only one label, are deleted. That done, the remaining candidates can either have the exact number of nodes and edges for each label, or more than that. Now it must be remembered that, with the previous assumption (Assumption 1) the common subgraph is surely fully included in at least one graph. So if there is one and only one candidate which has the right number of nodes and edges for each label, and if enough graphs from the total population have been excluded to represent the whole subgraph tolerance (Assumption 2), that graph must be the common subgraph. If, instead, all the remaining graphs have more nodes or edges than the solution in at least one label, or there is more than one distinct candidate, with the exact number of nodes and edges, a way to isolate the common subgraph is needed.

Now we need an heuristic to extract the common subgraph from the common subgraph between candidates. A possible solution would be to perform the perfect intersection (since assumption 2 is assumed to hold and thus the subgraph tolerance has already been fully taken into account) between the remaining graphs, but this is an NP operation. To improve that, since the complexity of the intersection cannot be reduced, the number of elements must be kept as low as possible. Thus, the intersection could be performed only between the two smallest graphs among the survivors: it would still be NP-complex, but it would be performed on fewer elements. Then the intersection could be tested to see if it was the only one graph with the right number of nodes and edges for each label, and if not the intersection could be repeated with the next smallest graph.

However, what is the probability of the smallest graph being so different from the solution? Since it is the smallest and the solution is surely included in it, it is reasonable to suppose that the error committed in choosing it is small (Assumption 3). Of course this will need to be experimentally verified, but if it works then the algorithm is completely linear in time.

DISCUSSING ASSUMPTIONS Now that a scratch of solution has been found, the assumptions made need to be discussed. Assumption 1 is the most demanding:

> "there is at least a graph which fully contains the common subgraph".

This would be undeniably true if the subgraph tolerance threshold was 1, i.e. the perfect common subgraph. Since it is not, but it would probably have a value of around 80%, this assumption would still be reasonable only if the subgraph tolerance applied only to graphs (i.e. excluded only some graphs instead of specific features): once that 20% of graphs has been excluded, the common subgraph with subgraph tolerance reduces to the perfect common subgraph.

Now, the point is that if the percentage of graphs excluded by the graph pruning of heuristic 4 is large enough to represent the subgraph tolerance, the common subgraph can be found by performing the perfect intersection among the remaining graphs, which is exactly the case discussed above implying that assumption 1 holds.

And since the assumption that states:

> "the percentage of graphs excluded by the graph pruning of heuristic 4 is large enough to represent the subgraph tolerance"

is assumption 2, the bottom line is that assumption2 => assumption 1, i.e if assumption 2 holds, then assumption 1 holds too.

Now let's analyze assumption 2, which states that the percentage of excluded graphs due to the pruning (the graphs that become too small after the pruning to be possible candidates for the solution) is such that the remaining graphs, in percentage, are equal or fewer to the subgraph tolerance threshold. (e.g. if we have a population of 100 graphs in the cluster, and the threshold is 80%, assumption 2 states that at least 20 of that graphs are excluded due to the pruning). Of course this cannot be proved, because if we had 100 identical graphs, none would been pruned and thus excluded. But the interesting property that emerges from the definition of assumption 2 is that it depends on the subgraph tolerance threshold. And it was previously shown that is reasonable to assume that the threshold value is not really important (assumption 0), as long as the threshold exists (and it's > 50%, for obvious reasons). Then, if the threshold value is not important to the quality of the solution, it's straightforward that if the pruning excludes at least some graphs, a threshold can be defined such that the percentage of surviving graphs is equal or less than this threshold. And assuming that the pruning excludes some graphs means that some node or edge labels are not commonly shared among all graphs, which is reasonable and also can be easily tested by looking at some sampled graphs. So, provided that there is a certain variety in the graphs, assumption 0 => assumption 2 => assumption 1.

THE FINAL SOLUTION Provided that assumption 0 holds, and that there is a certain variety in graphs, which will be proved experimentally since it's not possible to prove them theoretically, the solution algorithm can now be detailed.

Once the cluster graphs have been loaded, each node must be scanned in order to build a table like the one detailed in heuristic 4. This table will be used to compute the number of nodes the common subgraph must have for each label. Then, for each node label that has a count of 0, i.e. must not be included in the solution, each node with that label in each graph will be deleted.

Algorithm 3.1 Node counting

```
#count the number of nodes each graphs has, for each node
    type (associated api)
nodeCounter = dict()
nodeCounter['total'] = collections.Counter()
for g in graphs: for n in g.nodes():
    if getApi(n) not in nodeCounter:
        nodeCounter[getApi(n)] = collections.Counter()
        nodeCounter[getApi(n)][str(id(g))]+=1

#count how many nodes of each type the solution must have
solutionNodes = collections.Counter()
for s in nodeCounter.keys():
    solutionNodes[s] = Numberize(graphs, nodeCounter[s],
        threshold)
```

This process will be repeated step by step for edges too, as detailed in heuristic 4.

Algorithm 3.2 Edge counting

```
#count the number of edges each graphs has, for each edge
   type (the type of an edge depends on the type of the
   nodes it connect and its direction)
edgeCounter = dict()
edgeCounter['total'] = collections.Counter()
for g in graphs:
    for e in g.edges():
        if (getApi(e[0])+getApi(e[1])) not in edgeCounter:
            edgeCounter[(getApi(e[0])+getApi(e[1]))] =
               collections.Counter()
        edgeCounter[(getApi(e[0])+getApi(e[1]))][str(id(g))
           ]+=1

#count how many edges of each type the solution must have
solutionEdges = collections.Counter()
for s in edgeCounter.keys():
    solutionEdges[s] = Numberize(graphs, edgeCounter[s],
       threshold)
```

Algorithm 3.3 This function extracts from a list of integers the highest number in the threshold percent of the list's items

```
def Numberize(graphs, counter, threshold):
    #transforms the counter in a list
    myList = list()
    for g in graphs:
        myList.append(counter[str(id(g))])
    counting_sort_reversed(myList)
    #find the % and return results
    return myList.pop(int(round(len(myList)*threshold) -1))
```

Each graph's number of nodes and edges will be checked, for each label, against the number of nodes and edges of the common subgraph. If at least one of this numbers is lower in the graph than in the solution, that whole graph will be deleted.

After performing the smaller graphs removal (Algorithm 3.5), since assumption 0 => assumption 2 => assumption 1, the percentage of surviving graphs is smaller or equal than the subgraph tolerance threshold. This means that the perfect common subgraph among the survivor is the solution. And since it is the perfect common subgraph,

Algorithm 3.4 Graph pruning

```
for g in graphs:
    for n in g.nodes():
        if solutionNodes[getApi(n)] == 0:
            g.remove_node(n)
    for e in g.edges():
        if solutionEdges[(getApi(e[0])+getApi(e[1]))] == 0:
            g.remove_edge(*e)
```

Algorithm 3.5 Smaller graphs removal

```
for g in graphs:
    for n in nodeCounter.keys():
        if (nodeCounter[n][str(id(g))] < solutionNodes[n]):
            graphs.remove(g)
            break
    if g in graphs:
        for e in edgeCounter.keys():
            if (edgeCounter[e][str(id(g))] < solutionEdges[e
                ]):
                graphs.remove(g)
                break
```

it must be fully included into all other graphs. So we only need to extract the smallest survivor graph, which is the common subgraph.

Algorithm 3.6 Solution extraction

```
#solution extrction
counting_sort_nodes(graphs)
if len(graphs) > 0:
    exportgraph = graphs[0]
else:
    exportgraph = nx.DiGraph()
```

As shown in algorithm 3.6,the solution extraction is obtained just by sorting the graphs on the number of nodes and exporting the smallest. Of course if all the graphs have been removed, the graph to be exported is an empty graph.

3.1.3 *Complexity proof*

In this section a formal proof of the algorithm's complexity will be provided.

ANALYSIS OF MATRIX CONSTRUCTION In the code which builds the node counter matrix, provided in Algorithm 3.7,

Algorithm 3.7 Node counter matrix

```
for g in graphs:
    for n in g.nodes():
        if getApi(n) not in nodeCounter:
            nodeCounter[getApi(n)] = collections.Counter()
        nodeCounter[getApi(n)][str(id(g))]+=1
    nodeCounter['total'][str(id(g))] = g.number_of_nodes()
```

the loop cycles on each node in every graph, i.e. has a complexity of O(n), with n = total number of nodes in all graphs. For the complexity to be linear all the instruction executed inside the graph must have a complexity of O(1), and in fact:

```
if getApi(n) not in nodeCounter:
```

checking if a key is present in a counter, since the counter is an hash structure, is O(1)

```
nodeCounter[getApi(n)] = collections.Counter()
```

initializing a new key in the counter is an O(1) instruction

```
nodeCounter[getApi(n)][str(id(g))]+=1
```

assigning a value (or incrementing the previous one) to a counter is O(1) due to the hash nature of the counter data structure.

```
nodeCounter['total'][str(id(g))] = g.number_of_nodes()
```

counting the number of nodes is a O(n) instruction, but it's executed out of the cycle, so it doesn't raise the complexity.

ANALYSIS OF SOLUTION SPECS DEFINITION Starting from the code:

```
for s in nodeCounter.keys():
    solutionNodes[s] = Numberize(graphs, nodeCounter[s],
        threshold)
```

This loop cycles on all the keys of nodeCounters, i.e. all the different API attributes of the nodes in all graphs, without duplication, so this loop is O(k). The instruction inside the loop must, thus, be O(something) so that O(something*k) $\leq$ O(n). To compute the complexity of that instruction the function Numberize needs to be analyzed.

ANALYSIS OF NUMBERIZE In this paragraph will be provided the analysis of the function Numberize, which code can be found in Algorithm 3.8.

Algorithm 3.8 Numberize

```
def Numberize(graphs, counter, threshold):
    #transforms the counter in a list
    myList = list()
    for g in graphs:
        myList.append(counter[str(id(g))])
    counting_sort_reversed(myList)
    #find the % and return results
    return myList.pop(int(round(len(myList)*threshold) -1))
```

First of all, the counter which now contains the number of nodes with a specific API indicized on the parent graph, is transformed into a list. To do that a loop on the graphs, i.e. with complexity O(g) is performed. This means that the total complexity is O(k*g).

Fortunately, the graphs are quite similar so every graph shares almost all its nodes API with all the other graphs. In the worst case scenario, each graph's nodes have a different API, so k = #nodes of graph g, and thus k*g = total number of nodes in all graphs = n, i.e. O(k*g) = O(n).

```
counting_sort_reversed(myList)
```

The counting sort is notoriously O(elements), and the elements here are equal to the number of graphs g, so again the total complexity is O(k*g) = O(n).

```
return myList.pop(int(round(len(myList)*threshold) -1))
```

The final instruction is a return, which performs a pop operation on a list (O(1)), calculating the index of the element to pop with simple math (O(1)).

Algorithm 3.9 Graph pruning

```
#graph pruning
for g in graphs:
    for n in g.nodes():
        if solutionNodes[getApi(n)] == 0:
            g.remove_node(n)
```

ANALYSIS OF PRUNING From the code of the pruning algorithm provided in Algorithm 3.9, the double loop cycles as usual on the total number of nodes, thus being O(n). The instruction inside need to be O(1):

```
if solutionNodes[getApi(n)] == 0:
```

accesses solutionNodes, which is a dictionary and thus can be accessed through hash, in O(1), and

```
g.remove_node(n)
```

which is O(1) since graphs nodes are directly accessible from the pointer n.

Algorithm 3.10 Smaller graphs removal

```
for g in graphs:
    for n in nodeCounter.keys():
        if (nodeCounter[n][str(id(g))] < solutionNodes[n]):
            graphs.remove(g)
            break
    if g in graphs:
        for e in edgeCounter.keys():
            if (edgeCounter[e][str(id(g))] < solutionEdges[e
                ]):
                graphs.remove(g)
                break
```

ANALYSIS OF GRAPH REMOVAL From the code of the graphs removal algorithm provided in Algorithm 3.10, graph removal performs a loop on graphs, which is O(g). As previously proved during the analysis of Numberize, looping on the graphs inside a loop on the different API labels, or vice-versa, has a total complexity of O(n).

The function graphs.remove is the deletion of the element of a list, thus direct accessed and performed in O(1).

Since the only operation performed inside this double loop is a conditional graph removal based on simple math, the overall complexity of the graph removal is still O(n).

Algorithm 3.11 Solution extraction

```
counting_sort_nodes(graphs)
if len(graphs) > 0:
    exportgraph = graphs[0]
else:
    exportgraph = nx.DiGraph()
```

ANALYSIS OF SOLUTION EXTRACTION From the code of the solution extraction algorithm provided in Algorithm 3.11, the counting sort is notoriously linear in the number of elements to be sorted, i.e the number of graphs, but this is not a normal counting sort since it orders more complex data than integers. As will be proved below, this sort complexity is O(n)

The export operation is beyond the scope of this analysis, since the solution has already been found and can be accessed in O(1).

Anyway, it's an operation that involves only the solution graph, so for n large enough O(export) $\leq$ O(n).

ANALYSIS OF COUNTING SORT Since the counting sort is not native in python, two implementations have been written and will now be analyzed. The first implementation, which is provided in Algorithm 3.12, sorts an array (list) of integer numbers, in reversed order, and one that sorts a list of graphs basing of the number of nodes.

Algorithm 3.12 Simple counting sort

```
def counting_sort_reversed(array):
    count = collections.Counter()
    # init with zeros
    max = 0
    for a in array:
        if a > max:
            max = a count[a] += 1
    # count occurences i = 0
    for a in reversed(range(max+1)):
        # emit
        for c in range(count[a]):
            # - emit 'count[a]' copies of 'a'
            array[i] = a i += 1
    return (array)
```

The basic counting sort begins creating a support structure. I decided to use a counter structure to have a structural relation between the element and its ordering.

```
for a in array:
    if a > max:
        max = a count[a] += 1
```

The first loop cycles on the elements of the array, thus being O(elements). All the operation it performs are O(1) (simple math, value assignment and counter value increasing).

When the support structure is fully initialized, it contains the new ordering of the elements, so the only operation needed is to replicate this new ordering in the array and return it.

```
for a in reversed(range(max+1)):
    for c in range(count[a]):
        array[i] = a i += 1
```

This loop complexity is more difficult to compute. The external loop cycles from the maximum element down to 0, but sometimes does nothing at all: the inner cycle only activates if count[a] $\neq$ 0, i.e. if there is at least one element in the original array which has a value equal to the current value of a. This means that for each possible value of a, the assignment operation is performed exactly the number of times that an element with value a appears in the original array. Thus, this two loops perform an assignment for each element of the original vector, so that the complexity is still O(elements).

Algorithm 3.13 Advanced counting sort

```
def counting_sort_nodes(list_of_graphs):
    count = dict()
    max = 0
    for g in list_of_graphs:
        if g.number_of_nodes() > max:
            max = g.number_of_nodes()
        if g.number_of_nodes() not in count.keys():
            count[g.number_of_nodes()]= dict()
            count[g.number_of_nodes()]['count'] = 0
            count[g.number_of_nodes()]['graphs'] = list()
        #count occurences
        count[g.number_of_nodes()]['count'] += 1
        #append graph
        count[g.number_of_nodes()]['graphs'].append(g)
    i = 0
    for n in range(max+1):
        if n in count.keys():
            for c in range(count[n]['count']):
                # - emit 'count[n]' copies of 'n'
                list_of_graphs[i] = count[n]['graphs'].pop()
                i += 1
    return (list_of_graphs)
```

This advanced version of the counting sort, provided in Algorithm 3.13, is based on the exact same principle of the previous, but sorts a more complex data structure: a graph. The tricky part is that now the sorting has to be performed on data which is not an integer, but has only an integer property.

```
for g in list_of_graphs:
    if g.number_of_nodes() > max:
        max = g.number_of_nodes()
    if g.number_of_nodes() not in count.keys():
        count[g.number_of_nodes()]= dict()
        count[g.number_of_nodes()]['count'] = 0
        count[g.number_of_nodes()]['graphs'] = list()
    #count occurences
    count[g.number_of_nodes()]['count'] += 1
    #append graph
    count[g.number_of_nodes()]['graphs'].append(g)
```

In this first loop the initialization of max (the maximum number of nodes among all graphs) is performed. Since the loop cycles on g, and the number_of_nodes function is O(nodes), the total complexity of the maximum extraction is O(n). Then the loop performs a series of assignments and initializations, which are O(1). Finally, it assigns to the corresponding counter the graph that originates that node count, so that, e.g. for a counter value of 4, all graphs with 4 nodes can be accessed. this is essential for the algorithm to work, but is just a O(1) complexity operation.

```
for n in range(max+1):
    if n in count.keys():
        for c in range(count[n]['count']):
            list_of_graphs[i] = count[n]['graphs'].pop()
            i += 1
```

This second cycles are pretty much the same than the basic version: n ranges from 0 to the maximum numbers of nodes. If there is any graph with a node count equal to n, for each of such graphs that graph is appended to the list of sorted graphs. Again the two loops combined cycle exactly on the number of elements of the initial graph list, and they perform only assignments, which are O(1) operations, for a total complexity equal to O(elements) = O(g).

The overall complexity of this sorting is thus O(n).

FINAL CONSIDERATIONS I proved complexity only for the operations on nodes, but since the code is pretty much identical, the same goes for operations on edges.

Since no part of the algorithm has a more than linear complexity, the overall complexity is O(n) for the node operations, where n is the total number of nodes considering nodes from all graphs, and O(e) for the edges operations, where e is the total number of edges considering edges from all graphs.

And since the nodes code does not interact with the edges code, to total complexity is O(n+e), which is linear.

3.2 BATCH TO STREAM

The next problem that needs to be addressed is transforming the existing infrastructure to make it support a constant stream of data, instead of analyzing only a given batch of samples. The idea is that anyone should be able to provide a malware binary sample anytime, and the sample has to be analyzed by cuckoo, clustered and generalized in order to extract his behaviors, and the behaviors delivered back to the user.

3.2.1 *Problem analysis*

To transform the infrastructure in a streaming infrastructure, there are three big problem that must be faced.

- First, the infrastructure have to be structured to support streaming.
- Second, since streaming implies a big volume of data, the spatial complexity can become a bottleneck. The memory usage needs thus to be checked and possibly optimized.
- Third, since a big amount of sample is expected, the time schedule of each phase needs to be well planned to avoid bottlenecks and to deliver the results in a reasonable time.

STRUCTURE PROBLEM The structure problem in transforming the infrastructure into a streaming one is essentially a problem of data storing: cluster information need to be stored in order to be able to perform the clustering on future data. Also, to be able to perform generalization, each graph in each cluster must be stored, since the generalization performed is not a perfect one, and thus the common subgraph is not sufficient to perform the intersection. Finally, a system to link each analyzed sample with the extracted behaviors need to be implemented.

While this are the main implementational differences, there is also another aspect to be considered: actually the infrastructure is composed by two different parts, the cuckoo analysis and the clustering + generalization. These two operations need to be chained somehow to be performed automatically in sequence, but this problem will be left to be analyzed in the time problem section.

MEMORY PROBLEM The memory problem arises as a consequence of the structure problem: since all the clusters need to be loaded in memory to perform the clustering of new samples, the space the clusters occupy needs to be reduced to the minimum. Currently the clusters object contains all the fps that characterize that cluster, in a list, and all the graphs that are part of that cluster, in another list.

Also, all the graphs of each cluster should be loaded in order to perform the graph generalization, and since graphs are heavy objects, this can quickly become a problem.

TIME PROBLEM The time problem is a double problem: first of all, the time complexity bottleneck needs to be identified, if present, and tried to be solved. Second, but not less important, the activities need to be scheduled. The analysis is divided in three phases: cuckoo analysis (fps & api calls extraction), clustering, generalization. The first phase is totally independent from previous data, so can be scheduled independently. The problem rises with the other two, since there is a lot of data loading, which is slow, and does not depend on the volume of data to be analyzed (e.g. to generalize even a single sample, all the graphs in that sample's clusters need to be loaded), a compromise between the overall time needed to provide results and the necessity to analyze a minimum amount of data needs to be found.

3.2.2 *Solution development*

STRUCTURE SOLUTION IDEA First of all a system to keep track of which behavior is extracted from which sample needs to be found. The most straightforward solution is to include in the cluster structure a list field containing the reference of all the binaries which fps have been clustered into that cluster.

On second thought, anyway, it has been decided to also export the cluster data in a database in order to be able to query it. It has been decided to use a simple, non-relational database like Mongodb to export the cluster data structure. The db structure consists of a single collection containing cluster information in form of id, list of the binaries, path of the generalized graph representing the extracted behavior, and list of the fps characterizing that cluster. To this end, the clusters structure has been modified to include an id field, which is just an increasing integer number.

The second part of the structure problem is the very structure of this work. Leaving aside the cuckoo analysis part, which is easily separable from the second part, before starting any analysis the existing clusters need to be loaded, and a global variable named lastcluster, which keeps track of the highest cluster number assigned as an id, needs to be initialized.

Then, every time the clustering phase is run, all the clusters which receive a new element or the newly created clusters are flagged as modified. To this purpose a boolean field has been inserted in the clusters data structure. In this way, during the generalization phase, only the clusters flagged as modified are generalized, saving computational time, but even more important avoiding to load the graphs of the other clusters.

The generalization phase is mostly untouched by the transformation into streaming since it needs to be totally repeated, but fortunately it's a linear operation.

MEMORY SOLUTION IDEA The basic idea in order to save memory is, obviously, to load only what is strictly necessary, only when it's strictly necessary, and then unload it. The problem is that loading data is a very time-consuming operation. A trade off thus needs to be found.

The cluster's problem is the first problem to be addressed: clusters are used in the clustering phase, and the only data needed there are the fps, the ids and the modified flags. Since there is no use for graphs in this phase, the graphs can be detached from the clusters data structure. Ids and flags are atomic data units, so there is nothing that can be done to reduce their space, but fps are a list which contains all the fps associated to all the graphs in each clusters. And each element of that list is a list composed by the single fps of that specific graph. This implies a lot of data duplication, which could be avoided using a set structure instead of a list of lists. Unfortunately, this deeply changes the cluster's data structure, and thus the clustering algorithm needs also to be restructured. A further discussion of this idea can be found in the following paragraph: Time solution idea, since changing the algorithm can also change it's time complexity.

The second data structure on which space can be saved is graphs. Graphs are really complex data structures which requires a lot of space to be loaded, so, again, it would be better to load them only at the last moment. Graphs are extracted from the cuckoo analysis results during the clustering phase. To reduce as much as possible the memory usage, only one result file is loaded at a time, and when the clustering algorithm has finished processing it, all the extracted graphs are stored. This does not increases the number of load/store operation since all the graphs need to be loaded one time and stored one time in the clustering phase. The only alternative would be to keep all the graphs in memory between the clustering phase and generalization phase, but this is clearly not a feasible option. In order to keep a link between each graph and the cluster it belongs to, without increasing the complexity of the data structure or the memory usage, each graph is stored in a directory which name is the id of the cluster that graph belongs to.

Since graphs then are only used in the generalization phase, that is the point they will be loaded in memory. The generalization phase processes one cluster for a time, so only one of the cluster's graphs need to be loaded, and after the generalization is finished the memory is freed. It could be theoretically possible to load only one graph for a time, compile the graph matrix, switch to the following graph, and so on, but this increases drastically the number of load operations[1], thus drastically decreasing the performance of the generalization phase.

TIME SOLUTION IDEA First, the scheduling problem will be addressed: thanks to the memory optimizations, clustering and gener-

1 Because the graph needs also to be loaded during the pruning operation, and then again during in the small graphs removal sub-phase

alization may be performed without recurring to compromised solution like loading only a sample of the graphs. The only consideration to be kept in mind is that the more sample are included in the analysis, the better the efficiency is. So the ideal setup would be to perform the clustering and generalization only after a certain number of binaries has been submitted. This number cannot be calculated at the moment, because it will mostly depend on the data volume submitted by the user.

In order to start the clustering a script has been written that checks the result folder from cuckoo analysis and moves all the result files into a specific directory. A second script placed at the beginning of the clustering algorithm checks this directory periodically and if the number of file is greater than the threshold, starts the analysis. The cuckoo result files are deleted as they are processed since they're not important per se.

The second part of the time problem is more challenging: in order to find the possible bottleneck the execution has been analyzed with c-profile, and it resulted that, on average, the clustering phase takes 10 times the time of the generalization phase. In order to reduce the complexity of the clustering phase, and possibly also the memory usage, a solution could be to use sets instead of lists to collect the cluster's element's FPs, and then instead of checking the sample FPs against each other sample's FPs set in the cluster, a single check against the whole set of FPs would be performed (Figure 3.2).

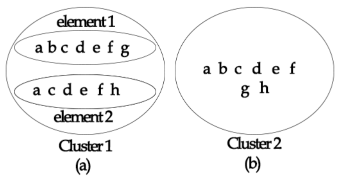


Figure 3.2: The Jaccard clustering performs a similarity check against each element of the cluster (a). The overlap clustering, on the the hand, performs only one check, and avoids FPs duplication in the sme cluster(b).

This is a deep change in the cluster structure, however, and requires the clustering algorithm to be rebuilt, since the FPs set thus obtained will grow (in number of elements) as more graphs are added to the cluster, but the FPs of a single graph will remain a few. This will cause the jaccard similarity, defined as

$$J(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

to decrease with the increasing of the cluster's elements, which is obviously undesired. A new similarity measure is needed, which can be the overlapping similarity:

$$Overlap(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

Being the denominator only dependent on the smallest set size, the effect mentioned above does not occur with this similarity. Of course the performance improvement needs to be tested, and also the quality of the extracted behaviors need to be compared. A comparison between the two clustering algorithms are shown in Algorithm3.15 and Algorithm 3.14.

3.3 VISUALIZATION

The goal of this part is to build an infrastructure that will allow to see the structure of the analyzed data and results: the relation between malware samples and extracted behaviors via fps.

3.3.1 *Problem analysis*

The basic problem is to build an infrastructure that will be able to render the relationship between analyzed sample and results. The goal of this visualization is a global one: being able to see if, adding samples, the emerging behaviors are shared among them, which behaviors are the most common ones, which data is anomalous and, ultimately, to verify the assumption of this whole work: that the system presented is able to extract meaningful behaviors from malware using its combined analysis method.

The biggest problem in this visualization task is to allow the user to see a huge amount of data and understand something, possible not by using numbers as aggregators, but to actually allow the user to see the results.

So it's necessary to represent in some way all the malware samples provided, and all the extracted behaviors. Also, the link between each sample and behavior needs to be represented. This link is a many to many link, since a single sample can present more than one behavior and each behavior hopefully is shared among a lot of samples. Finally, ideally a third level need to be introduced, explicitly representing the fingerprints as the linking element between sample and behavior.

This means that a data structure containing that information need to be saved. The cluster structure already contains the link between each behavior and the generating fps, so the easiest thing would be to add to the cluster data structure information on the generating sample.

Algorithm 3.14 Jaccard clustering

```
def clusterize_jaccard(fps, graph):
    global lastcluster
    for c in clusters:
        sim = jaccardSim(c['fps'], fps)
        if sim > 0.60:
        c['graphs'] += 1
            f = 'clusters/'+str(c['id'])+'/graph'+str(c['
                graphs'])
            nx.write_gpickle(graph, f + '.gpickle')
            nx.write_dot(graph, f + '.dot')
            c['fps'] = c['fps'].union(fps)
            c['modified'] = True
            return c['id']
    c = dict()
    c['fps'] = set()
    c['fps'] = c['fps'].union(fps)
    c['graphs'] = 0
    c['modified'] = True
    c['id'] = lastcluster+1
    lastcluster+=1
    os.mkdir('clusters/' + str(c['id']) + '/')
    clusterlist.append(str(c['id']))
    f = 'clusters/' + str(c['id']) + '/graph' + str(c['
        graphs'])
    nx.write_gpickle(graph, f + '.gpickle')
    nx.write_dot(graph, f + '.dot')
    clusters.append(c)
    return c['id']

#Compute Jaccard Similarity of 2 sets
def jaccardSim(setA, setB):
    if len(setA) == 0 and len(setB) == 0:
        return 1
    commonAB = setA.intersection(setB)
    unionAB = len(setA) + len(setB) - len(commonAB)
    return float(len(commonAB))/float(unionAB)
```

Algorithm 3.15 Overlap clustering

```
def clusterize_overlap(fps, graph):
    global lastcluster
    for c in clusters:
        sim = overlapSim(c['fps'], fps)
        if sim > 0.60:
        c['graphs'] += 1
            f = 'clusters/'+str(c['id'])+'/graph'+str(c['
                graphs'])
            nx.write_gpickle(graph, f + '.gpickle')
            nx.write_dot(graph, f + '.dot')
            c['fps'] = c['fps'].union(fps)
            c['modified'] = True
            return c['id']
    c = dict()
    c['fps'] = set()
    c['fps'] = c['fps'].union(fps)
    c['graphs'] = 0
    c['modified'] = True
    c['id'] = lastcluster+1
    lastcluster+=1
    os.mkdir('clusters/' + str(c['id']) + '/')
    clusterlist.append(str(c['id']))
    f = 'clusters/' + str(c['id']) + '/graph' + str(c['
        graphs'])
    nx.write_gpickle(graph, f + '.gpickle')
    nx.write_dot(graph, f + '.dot')
    clusters.append(c)
    return c['id']

#Compute Overlap Similarity of 2 sets
def overlapSim(setA, setB):
    commonAB = setA.intersection(setB)
    lenght = min([len(setA), len(setB)])
    return float(len(commonAB))/float(lenght)
```

3.3.2 *Solution development*

From the problem analysis emerged clearly that what is needed to be represented is a three layer structure composed of interconnected nodes that must be enriched with at least basic tracking information. This infrastructure must be able to provide statistical information at glance, and to represent huge amounts of data while being easily comprehensible.

Reasoning on the structure of the data, it's reasonable to think of a sort of graph. This graph must have three different types of node: sample, fps and behavior, and each node must be linked to the generating nodes.

The problem is that huge graphs are hardly readable, and the main problem is that there are too many nodes in a small space. So how can the space be augmented, without using impossibly huge screens? The first idea is to zoom out. To maintain readability while zooming out, each node type must be immediately recognizable. In order to achieve that, different colors on a black background may be used. But still, with a lot of nodes the graph readability could not be easy.

To find a solution to the problem, comparison with a similar one can come in hand. It has been looked for the usual readable representation of something with a lot of elements, like stars. When representing stars, it is usually used a star map, which is a 2D maps of a little region of the space. But what if one wanted to represent and see all the known stars? The answer is pretty obvious: a 3D map can be used, and navigated to explore the different regions.

This is the main idea for such a complex representation: an explorable 3D map, which of course needs to convey the largest number of information from the maximum possible distance of the camera.

And a graph can easily be represented as a 3D object.

At this point, the best way of convey the largest amount of information needed to be found. Colors has been used to distinguish different types of nodes:

- red for behaviors, which are the most important nodes
- yellow for samples
- green for fps

The chosen colors can easily be seen on a black background, improving glance visualization.

Then, applied the principle that the most important information must be more visible has been applied:

- Empty behaviors, while still worthy of being represented, are not of much interest: they have been rendered as grey nodes.

- Samples which extracted behaviors are generated only from one or two samples, are not of much interest, because the aim of this work is to highlight that behaviors are shared among a large number of samples, so they have been rendered rendered blue.

Following the same visibility principle, nodes have been sized accordingly:

- FPs nodes are small, since they're only needed to support the edges, but don't convey information by themselves.
- The more samples a behavior is extracted from, the bigger it's rendered, following a logarithmic scale to avoid giant behaviors to hide the smaller ones.

Finally, the node shape has been used to convey some information of the behavior itself:

- the more complex a behavior is, i.e. the more nodes are in the generalized graph, the more complex it's shape is. To allow users to understand shapes with a glance, only four shapes have been used: cubes, octahedrons, dodecahedrons, and icosahedrons.

To guarantee traceability, each behavior is labeled with the number of the cluster is generalized from, so that its structure can be inspected manually.

3.3.3 *Implementation*

The first issue in implementing the solution presented above is how to keep track of the linking between behavior, fps and samples. Since in the cluster data structure, which has a record for each behavior, information on the FPs is already included, the simplest solution is to attach to each FPs a reference to the generating sample. Of course each sample may generate more than one subgraph with relevant FPs, so the sample identifiers here may be duplicated.

That decided, the whole visualization infrastructure must be implemented. Fortunately, a similar product already existed: Ubigraph [1]. Ubigraph is system to visualize dynamic graphs that make use of a server to host the whole rendering infrastructure. The server can be easily commanded by python scripts. The advantage of this solution is that the server can run indefinitely and new nodes can be added as new data is processed, allowing users to effectively see the whole process in action.

A script has been implemented to convert the analysis results in a graph and send it to the Ubigraph server to be rendered on the screen. Ubigraph also provides basic controls for navigation, allowing the user to change the angle of view, zoom into the interesting structures to view better interesting features and zoom out to get a general picture of what is happening.

Algorithm 3.16 Rendered data structure

```
import xmlrpclib
import sys
import os
import networkx as nx
import pickle
import math
import time
# Create an object to represent our server.
server_url = 'http://127.0.0.1:20738/RPC2'
server = xmlrpclib.Server(server_url)
server.ubigraph.clear()
# define something useful
lastcluster = -1
clusters = []
next_id = 0
samples = dict()
clusternodes = dict()
fps_struct = dict()
U = server.ubigraph
edges = set()
```

As detailed in Algorithm 3.16, it has been defined a data structure to keep track of which behaviors, FPs and samples have already been rendered. This is vital since this GUI need to be executed continuously while new data is being processed, and needs thus to be able to update the graph without rebuilding it. In order to do that a way to compute the differences is needed, and this data structure provides an easy way to do that. The reference of the already created nodes are kept as an integer number: the id that Ubigraph requires to identify each node. At the same way, the edges are identified by the concatenation of the ids of the nodes they connect.

The most up to date cluster information is loaded through through the same exact function that loads them in the generating algorithm (see Algorithm 3.17and Algorithm 3.18)

Then, first of all the sample nodes are generated by loading information from the reference, since in the cluster structure they are duplicated, while here each sample has a unique record, as shown in Algorithm 3.19.

Algorithm 3.17 Look for clusters

```
#look for clusters
try:
    os.stat('clusters/')
except:
    os.mkdir('clusters/')
clusterlist = os.listdir('clusters/')
if '.DS_Store' in clusterlist:
    clusterlist.remove('.DS_Store')
importClusters()
```

Algorithm 3.18 Import clusters

```
def importClusters():
    global lastcluster
        if len(clusterlist) == 0:
            print "warning: no prior existing clusters"
            return False
        else:
            for c in clusterlist:
                f = open('clusters/' + c + '/' + c, "r")
                clusters.append(pickle.load(f)) f.close()
                print "loaded cluster:", c
                if int(clusters[len(clusters)-1]['id']) >
                    lastcluster:
                    lastcluster = int(clusters[len(clusters)
                        -1]['id'])
            return True

CreateSampleNodes()
```

Algorithm 3.19 Create sample nodes

```
def CreateSampleNodes():
    global next_id
    reference = load_obj('reference')
    for impl in reference:
        if impl not in samples:
            if len(reference[impl]) > 0:
                samples[impl] = next_id
                U.new_vertex_w_id(next_id)
                U.set_vertex_attribute(next_id, 'color', '
                    #0000ff')
                next_id += 1
    return

def save_obj(obj, name ):
    with open(name + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

def load_obj(name ):
    with open(name + '.pkl', 'rb') as f:
        return pickle.load(f)
```

Once the sample nodes are ready, the clusters are processed, and for each behavior not already present a new node is created. Then the FPs of the considered cluster are scanned, and for each one, if a node representing that is not already present, a new one is created. Then, the FPs node is linked with the originating sample node and the extracted behavior node. The FPs are checked even if the behavior node was already present because the cluster can have been modified, i.e. new samples can have been added to that cluster and thus new fps nodes or new edges must be created. See Algorithm 3.20for the details.

Finally, it is checked if the behavior is extracted from more than two samples, the color of those sample nodes is set to yellow, and all the properties of the behavior node (size, color, label and shape) are set according to the specifics detailed in the previous section, as shown in Algorithm 3.21.

Algorithm 3.20 Cluster node creation

```
#for each sample node, create its fps nodes
for c in clusters:
    if c['id'] not in clusternodes:
        clusternodes[c['id']] = next_id
        U.new_vertex_w_id(next_id)
        next_id += 1
    n_source = set(c['source'])
    for i in range(0, len(c['fps'])):
        #linking fps to implementations
        k = frozenset(c['fps'][i])
        if k not in fps_struct:
            CreateNewFps(next_id)
            CreateNewEdge(samples[c['source'][i]],next_id)
            CreateNewEdge(next_id, clusternodes[c['id']])
            next_id += 1
        else:
            CreateNewEdge(samples[c['source'][i]],fps_struct
                [k])
            CreateNewEdge(fps_struct[k], clusternodes[c['id'
                ]])
        if len(n_source)>2:
            U.set_vertex_attribute(samples[c['source'][i]],
                'color', '#ffff00')
    SetBehaviourNodeProperties( clusternodes[c['id']],len(c[
        'source']))

def CreateNewFps(FpsId):
    fps_struct[k] = FpsId
    U.new_vertex_w_id(FpsId)
    U.set_vertex_attribute(FpsId, 'color', '#00ff00')
    U.set_vertex_attribute(FpsId, 'size', '0.2')
    return

def CreateNewEdge(node1, node2):
    if str(node1) + str(node2) not in edges:
        U.new_edge(int(node1), int(node2))
        edges.add(str(node1) + str(node2))ù
        return True
    else:
        return False
```

Algorithm 3.21 Cluster node properties

```
def BehaviourNodeShape(NumberOfNodes):
    if NumberOfNodes > 3 and NumberOfNodes <= 7:
        return 'octahedron'
    else:
        if NumberOfNodes > 7 and NumberOfNodes <= 15:
            return 'dodecahedron'
        else:
            if NumberOfNodes > 15:
                return 'icosahedron'
    return 'cube'

def SetBehaviourNodeProperties(NodeId, SourceLength):
    U.set_vertex_attribute(NodeId, 'color', '#ff0000')
    U.set_vertex_attribute(NodeId, 'label', str(c['id']))
    U.set_vertex_attribute(NodeId, 'size', str(round(math.
        log10(SourceLength)+1,1)))
    graph = nx.read_gpickle('clusters/' + str(c['id']) + '/
        export.gpickle')
    if graph.number_of_nodes() == 0:
        U.set_vertex_attribute(clusternodes[c['id']], 'color
            ', '#474747')
    else:
        U.set_vertex_attribute(clusternodes[c['id']], 'shape
            ', BehaviourNodeShape(graph.number_of_nodes))
    return
```

4 EXPERIMENTS

In chapter 4 all the experiments needed to verify assumptions and results form chapter 3 will be presented, and results will be discussed.

4.1 GENERALIZATION ALGORITHM TESTING

Since the whole algorithm is based on the assumptions,these are the first thing we will test:

- graph variety, i.e. the percentage of identical graph in a cluster is small.
- Assumption 3: the error committed choosing the smallest surviving graph is small.

All other assumption have been proved to depend on either of these on or assumption 0, which is the fundamental assumption for all the project.

It's also straightforward that assumption 3 is valid if and only if the quality of result is good, and vice versa.

4.1.1 *Graph variety*

In order to test the graph variety in each cluster, a little script has been created. The code is provided in Algorithm4.1

The script simply confront each graph in the cluster with each other graph in the same cluster, checking isomorphism and API attributes matching.

The expected result is a low percentage of matches, which, note, is not the percentage of matching graphs because each graph is checked multiple times against the others. A reasonable result would thus be $\leq 30\%$.

The script was run on a sample of 218 clusters generated with the algorithm detailed in section 1 from a sample of 1000 cuckoo analysis results, and resulted in a match percentage of 19%, well below the limit.

Please note that a low percentage of graph matching does not invalidate the clustering phase: in the clustering phase the goal is to match similar graphs, and similar graph are considered non matching from this script even if there is only the slightest difference. On the contrary, similar but different graphs implies that this generalization phase is necessary and useful.

Algorithm 4.1 Graph variety script

```
import os
import networkx as nx
import sys
import pickle
def equalNode(node1,node2):
    if str(node1['api']) == str(node2['api']):
        return True
    else:
        return False

def isomorphic(graph1,graph2):
    if nx.is_isomorphic(graph1, graph2, node_match =
       equalNode):
        return True
    else:
        return False

if __name__ == '__main__':
    clustermatch = list()
    mypath = sys.argv[1]
    clusters = os.listdir(mypath)
    for c in clusters:
        if os.path.isdir(miopath + c):
            print " processing cluster: " + c
            total = 0
            match = 0
            graphls = os.listdir(mypath + c + '/')
            graphls.remove('export.gpickle')
            graphlist= list()
            for g in graphls:
                if g.endswith('.gpickle'):
                    graphlist.append(g)
            for g in graphlist:
                for g2 in graphlist:
                    if g != g2:
                        total+=1
                        if isomorphic(nx.read_gpickle(mypath
                             + c + '/' + g),nx.read_gpickle(
                            mypath + c + '/' + g2)):
                            match+=1
            if total > 0:
                clustermatch.append(match*100/total)
                print match*100/total
            else:
                #rare case in which there's only one graph
                clustermatch.append(0) print 0
    match_percentage = sum(clustermatch) / len(clustermatch)
    print " matching percentage: " + str(match_percentage)
```

4.1.2 *Solution quality*

Defining a test to verify the quality of the solution is not a straightforward task, for the simple reason that the effective test would be to implement the np algorithm and confront the results. This is impractical for several reasons: first, implementing an NP algorithm is anything but simple; second, in order to use it to generalize a sample of graphs would take a lot of time, since complexity is so high. So we decided to exploit another algorithm, heuristic based, that was implemented to perform the generalization but was discarded for two reasons: exponential complexity and threshold fixed to 1. This algorithm has been implemented in [6], and details about it can be found there. We will assume, basing on experimental verification of [6], that that algorithm works, and we will refer to it from now on as the previous algorithm.

Thus, the testing of the algorithm will be divided in several phases:

1. Testing against known solution.
2. Testing against the previous algorithm using threshold 1.
3. Introduction of empty graphs to verify threshold effectiveness.
4. Testing against the previous algorithm manually examining a sample of non matching cases.

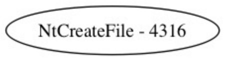
4.1.2.1 *Testing against known solution*

The first test will be to generate a bunch of graphs, in a way that the perfect common subgraph is known a priori, and then run the algorithm on them. The easiest way to do that is to start from the desired common subgraph itself, and then add nodes randomly, minding never to include the same node in two different graphs. Then the threshold will be set to one and the algorithm run. The expected result is the perfect common subgraph.

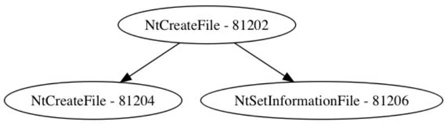
TEST 1 As can be seen from Figure 4.1 the generalization algorithm is correctly able to extract the common subgraph.

TEST2 Again the algorithm is able to extract the common subgraph, as shown in Figure 4.2

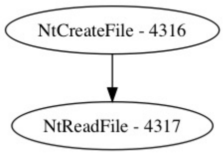
TEST 3 The third test is another simple test, but this time the graphs used will be generated in such a way that no node is shared among them. Then the threshold will be set to one and the algorithm run. The expected solution is an empty graph. As shown in Figure 4.3, the algorithm correctly extracts an empty behavior.



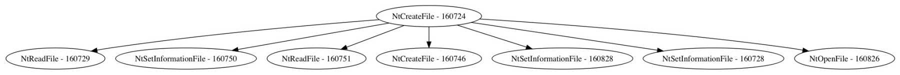
(a) Generating common subgraph



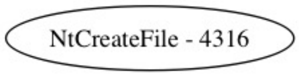
(b) Added graph



(c) Added graph

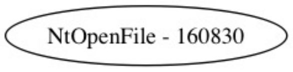


(d) Added graph

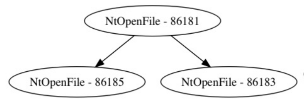


(e) Extracted common subgraph

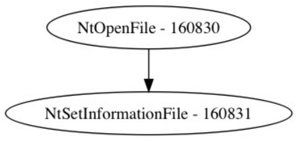
Figure 4.1: Starting from a generating common subgraph (a), additional graphs are generated by inserting different nodes and are added to the cluster (b, c, d). Then the generalization algorithm is run to verify if it is able to extract the common subgraph (e), which must be identical to the generating one(a).



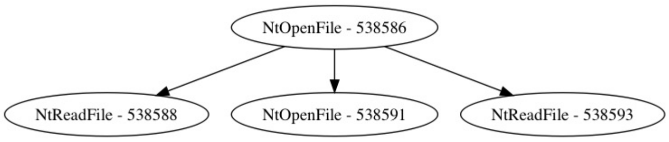
(a) Generating common subgraph



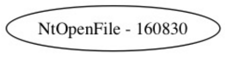
(b) Added graph



(c) Added graph

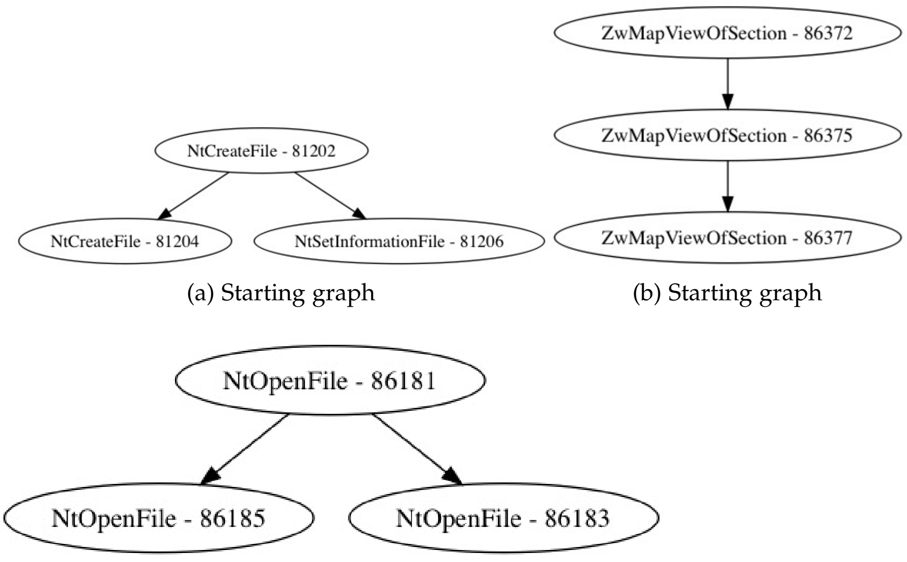


(d) Added graph



(e) Extracted common subgraph

Figure 4.2: Starting from a generating common subgraph (a), additional graphs are generated by inserting different nodes and are added to the cluster (b, c, d). Then the generalization algorithm is run to verify if it is able to extract the common subgraph (e), which must be identical to the generating one(a).


NtCreateFile - 81202
NtCreateFile - 81204
NtSetInformationFile - 81206
ZwMapViewOfSection - 86372
ZwMapViewOfSection - 86375
ZwMapViewOfSection - 86377
NtOpenFile - 86181
NtOpenFile - 86185
NtOpenFile - 86183

(a) Starting graph (b) Starting graph

(c) Starting graph (d) Extracted common subgraph

Figure 4.3: Starting from three graphs without even a common node (a,b,c) the algorithm is run to verify if it is able to extract an empty common subgraph (d)

4.1.2.2 *Testing against the previous algorithm using threshold 1*

The second phase consists of more refined tests: the previous algorithm will be run on a sample of clusters, large enough to have a good statistic (>200 clusters), then the algorithm developed in this work will be run on that same clusters setting the threshold to 1 and the resulting common subgraphs will be confronted using the matching algorithm detailed in Algorithm 4.1 opportunely modified. The expected result is an high percentage (>90%) of matching between the two algorithms. A 100% is not expected since the algorithm developed in this work is still an approximated algorithm: it's impossible to provide perfect results since the solution of the common subgraph problem is NP, and the algorithm developed in this work is linear.

TEST 4 The test results is a matching percentage of 99% over 218 analyses clusters, exactly as expected.

4.1.2.3 *Introduction of empty graphs to verify threshold effectiveness*

The third phase consists of inserting empty graphs into the previous clusters. Since the only common subgraphs with an empty graph is an empty graph, this should make all the solutions converge to an empty graph.

TEST 5 An empty graph has been inserted in every cluster, then all 218 clusters have been generalized and the resulting common subgraphs matched against an empty graph. The result is a matching percentage of 100%

TEST 6 Now, the threshold is going to be lowered to allow the subgraph tolerance to compensate for the empty graphs inserted. Since one empty graph will be inserted into each cluster, and the average cluster has 4-5 graphs, a 75% value should be sufficient to compensate for all (or almost all) the clusters. That done, the new common subgraphs will be matched with the one obtained with threshold = 1 without empty graphs. The expected result is a perfect (or almost perfect) matching.

The result after all 218 clusters were matched is a total matching percentage of 97%

4.1.2.4 *Testing against the previous algorithm manually examining a sample of non matching cases*

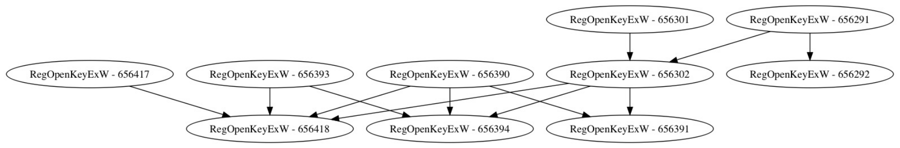
The final test to be performed is to run both the previous algorithm and the algorithm developed in this work on some of clusters, using a the threshold obtained from the tests in 4.1.4, and confront the results. Obviously the expected result is not a perfect match, since with the algorithm developed in this work a subgraph tolerance is being used, but the purpose of the confrontation is only to isolate the cases in which the tolerance makes the difference in the final results. This cases will be sampled and manually examined to verify that the result difference is due to the threshold variation, and not to an error of the algorithm.

TEST 7 All 218 clusters were matched.The 97% of the solutions were identical, the other 3% was different and has been manually inspected, and the results can be seen in Figure 4.4, Figure 4.5, Figure 4.6 and Figure 4.7.

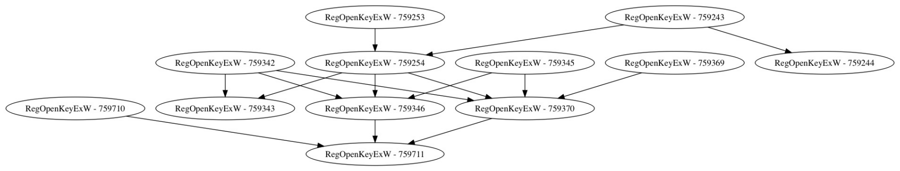
4.1.3 *Performance testing*

Since the major purpose of this thesis is the creation of an efficient algorithm, the performance deserves to be measured as an additional proof of the quality of the algorithm. Since one of the reasons for which the previous algorithm could not be used was his time complexity, i.e. it's inefficiency, that algorithm will be used as a baseline to calculate the speed improvement.

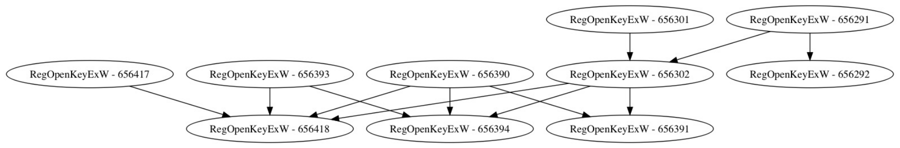
In order to measure the execution time of the algorithm a python-integrated profiling framework will be used: c-profile. C-profile computes the total time of execution of a program, with a reasonable over-



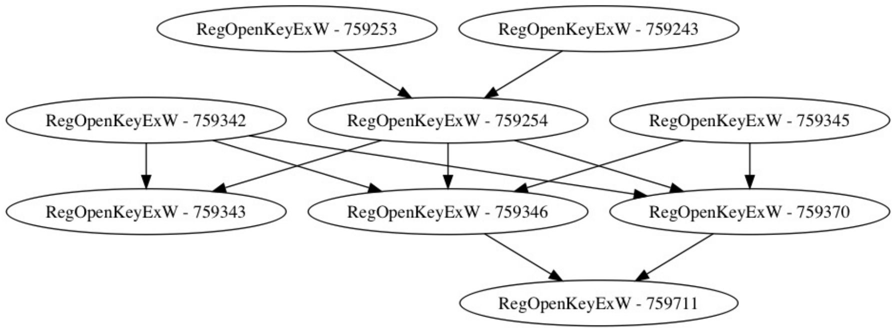
(a) Cluster 152 - Graph 1



(b) Cluster 152 - Graph 2

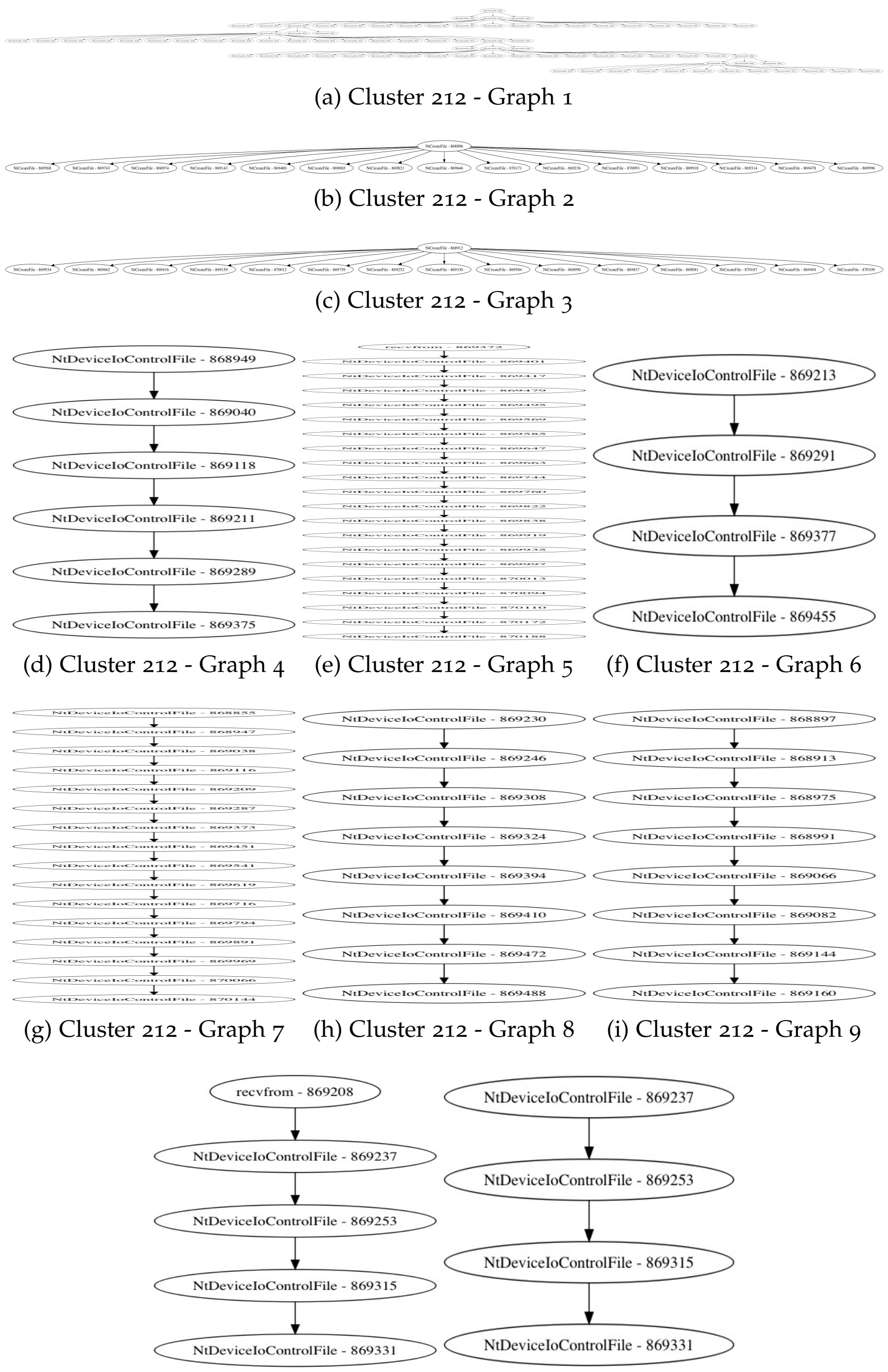


(c) Extracted common subgraph



(d) Previous algorithm's extracted common subgraph

Figure 4.4: Cluster 152 presents anomalous graphs: all the nodes had the same API (a, b). It's reasonable that the algorithm developed in this work does not have the tools to extract the common subgraph, and in fact the solution it extracted is not exact (c). But to be fair, neither the previous algorithm was able to extract the perfect common subgraph (d).


NtDeviceIoControlFile - 868949
NtDeviceIoControlFile - 869040
NtDeviceIoControlFile - 869118
NtDeviceIoControlFile - 869211
NtDeviceIoControlFile - 869289
NtDeviceIoControlFile - 869375
NtDeviceIoControlFile - 869213
NtDeviceIoControlFile - 869291
NtDeviceIoControlFile - 869377
NtDeviceIoControlFile - 869455
NtDeviceIoControlFile - 869230
NtDeviceIoControlFile - 869246
NtDeviceIoControlFile - 869308
NtDeviceIoControlFile - 869324
NtDeviceIoControlFile - 869394
NtDeviceIoControlFile - 869410
NtDeviceIoControlFile - 869472
NtDeviceIoControlFile - 869488
NtDeviceIoControlFile - 868897
NtDeviceIoControlFile - 868913
NtDeviceIoControlFile - 868975
NtDeviceIoControlFile - 868991
NtDeviceIoControlFile - 869066
NtDeviceIoControlFile - 869082
NtDeviceIoControlFile - 869144
NtDeviceIoControlFile - 869160
recvfrom - 869208
NtDeviceIoControlFile - 869237
NtDeviceIoControlFile - 869253
NtDeviceIoControlFile - 869315
NtDeviceIoControlFile - 869331
NtDeviceIoControlFile - 869237
NtDeviceIoControlFile - 869253
NtDeviceIoControlFile - 869315
NtDeviceIoControlFile - 869331

(a) Cluster 212 - Graph 1

(b) Cluster 212 - Graph 2

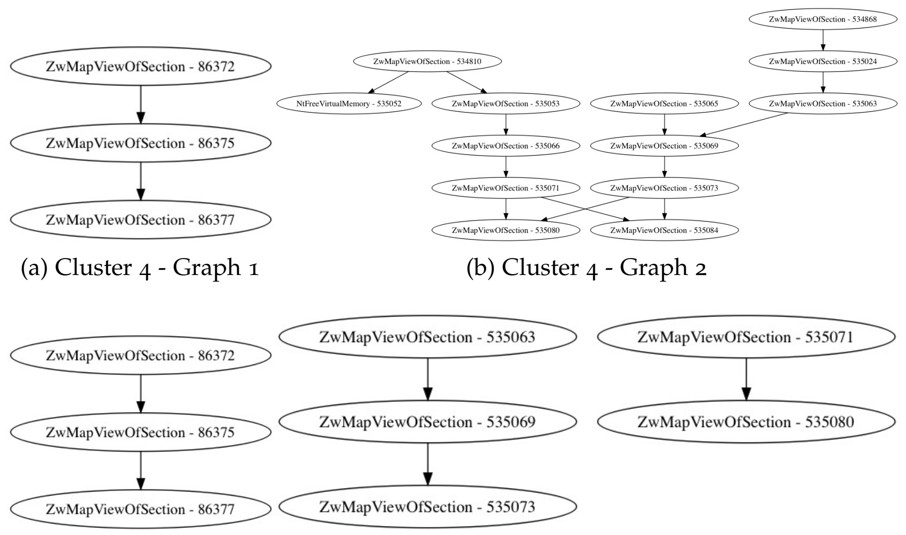
(c) Cluster 212 - Graph 3

(d) Cluster 212 - Graph 4 (e) Cluster 212 - Graph 5 (f) Cluster 212 - Graph 6

(g) Cluster 212 - Graph 7 (h) Cluster 212 - Graph 8 (i) Cluster 212 - Graph 9

(j) Cluster 212 - Graph 10 (k) Extracted common subgraph

Figure 4.5: Cluster 212 has three anomalous graphs, which are totally different from the other seven graphs in the cluster: a, b and c are all composed of NtCreateFile nodes, whereas d, e, f, g, h, i and j are made of NtDeviceControlFile nodes. The previous algorithm extracted an empty common subgraph, but thank to the subgraph tolerance the algorithm developed in this work was able to extract a better solution (k)


ZwMapViewOfSection - 86372
ZwMapViewOfSection - 86375
ZwMapViewOfSection - 86377
ZwMapViewOfSection - 534868
ZwMapViewOfSection - 534810
ZwMapViewOfSection - 535024
NtFreeVirtualMemory - 535052
ZwMapViewOfSection - 535053
ZwMapViewOfSection - 535065
ZwMapViewOfSection - 535063
ZwMapViewOfSection - 535066
ZwMapViewOfSection - 535069
ZwMapViewOfSection - 535071
ZwMapViewOfSection - 535073
ZwMapViewOfSection - 535080
ZwMapViewOfSection - 535084
ZwMapViewOfSection - 86372
ZwMapViewOfSection - 86375
ZwMapViewOfSection - 86377
ZwMapViewOfSection - 535063
ZwMapViewOfSection - 535069
ZwMapViewOfSection - 535073
ZwMapViewOfSection - 535071
ZwMapViewOfSection - 535080

(a) Cluster 4 - Graph 1

(b) Cluster 4 - Graph 2

(c) Exported common subgraph

(d) Previous algorithm's exported common subgraph(s)

Figure 4.6: Cluster 4 is a normal cluster, and the algorithm developed in this work extracts the common subgraph without any problems (c). The reason why the matching is different is that the previous algorithm erroneously extracts two common subgraphs, one of which is contained in the other (d).

head. Since the same profiler will be used to measure both algorithms execution times, the overhead will be negligible.

The expected result is an execution time for this at least one order of magnitude smaller than the previous algorithm's one.

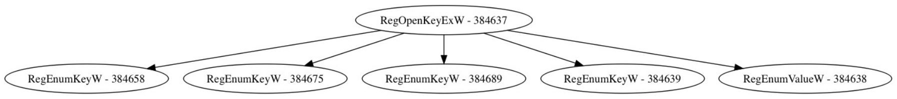
Test results are reported in Table 4.1.

	Test 8	Test 9	Test 10
Number of sample clusters	218	216	213
Number of graphs	1000	1100	1050
Generalization time	11.993 seconds	21.972 seconds	15.443 seconds
Previous time	429.207 seconds	578.493 seconds	468.245 seconds
Speedup	x35.79 (+3479%)	x26.32 (+2532%)	x30.32 (+2932%)

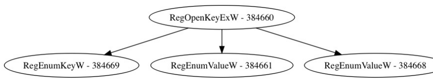
Table 4.1: Test 8-10: Speedup w.r.t previous algorithm

4.1.4 *Threshold testing*

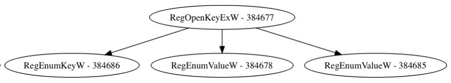
Assumption 0 states that the threshold is robust to small variation, i.e the solution does not change if the threshold varies by a small amount. This assumption can be verified by running the algorithm several time, each time varying the threshold of a unit, and computing the matching between the result of each execution with the following



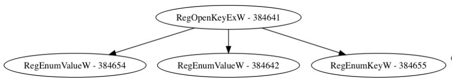
(a) Cluster 53 - Graph 1



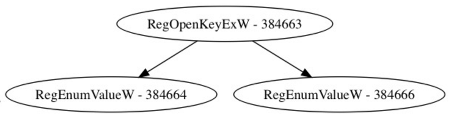
(b) Cluster 53 - Graph 2



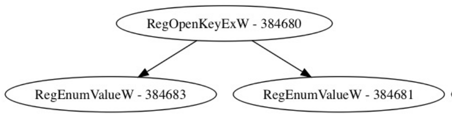
(c) Cluster 53 - Graph 3



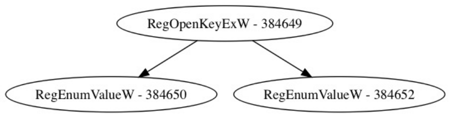
(d) Cluster 53 - Graph 4



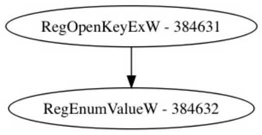
(e) Cluster 53 - Graph 5



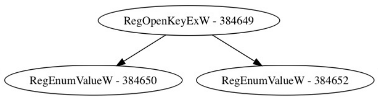
(f) Cluster 53 - Graph 6



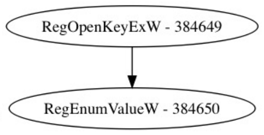
(g) Cluster 53 - Graph 7



(h) Cluster 53 - Graph 8



(i) Extracted common subgraph



(j) Previous algorithm's extract common subgraph

Figure 4.7: Cluster 53 is a normal cluster. The difference in results is due only to the subgraph tolerance: the solution extracted by the algorithm developed in this work (i) has an additional node w.r.t the previous one (j), which is shared between all the graphs in the cluster (a, b, c, d, e, f, g) but one (h):

one. This matching can then be plotted against the threshold. The expected result is a flat line, indicating that the solution is robust to small threshold variation. The result is reported in Figure 4.8.

The next test is to progressively increase the threshold variation entity, and for each threshold value calculate the solution variation due to that threshold variation. The expected result is the presence of a maximum in the matching, which represent the threshold value more resistant to variation, and this value is expected to be found in the 75-85% interval.

In order to perform this test the algorithm will be looped and with each iteration the threshold will be increased of 1, from an initial value of 60 to a final value of 90, to obtain the common subgraph for each threshold value, for each cluster. Then, a modified version of the matching script will compute the match percentage for each solution with the following one (i.e. the one obtained by the algorithm with the threshold increased by 1), initially, and then augmenting the gap increasingly.

The results have been plotted against the threshold to better be able to see the point of maximum (see Figure 4.9).

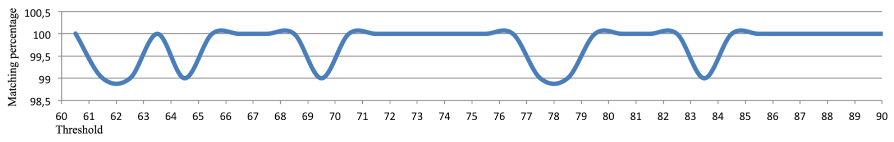
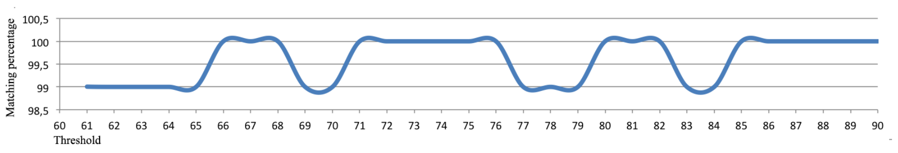


Figure 4.8: This graph has been obtained confronting each cluster's solution with the one obtained with a threshold increasing of 1. It's evidently almost a flat line, since it only varies between 100% and 99% matching. This proves that assumption 0 holds.

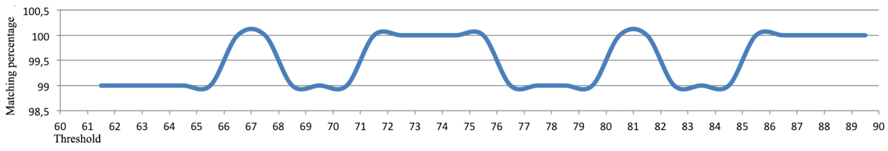
Looking at Figure 4.9it's evident, even if the variation is really small (1%), that there is a point of maximum between 73 and 74%, and also another possible one past 87%.

I decided to keep 74% as the best threshold value, since it's in the middle of the analyzed interval and it's obviously more robust to the presence of anomalous graphs. Also, in order to guarantee that assumption 2 holds, a lesser threshold is better.

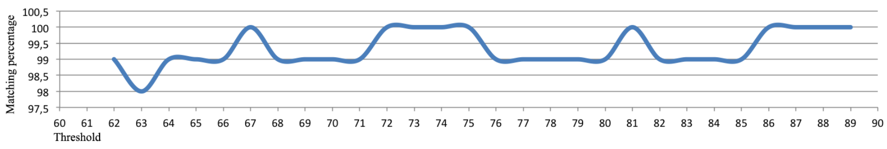
Note: it can be wondered why the graphs are so flat (they still vary between 98% and 100%) and if such data is meaningful. The reason for this flatness is to be found in the method used to perform the comparison between results: for reason of complexity, a binary algorithm has been used, i.e. an algorithm that returns 1 if there is a perfect match between the compared graphs, and 0 if not. Using a fuzzy algorithm, i.e. an algorithm able to return a similarity measure between matched graphs, would most probably lead to a better curve.



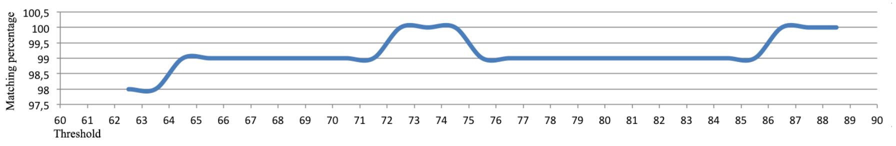
(a)



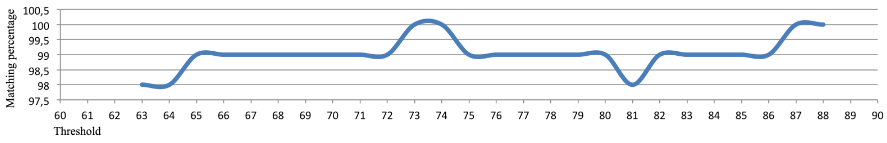
(b)



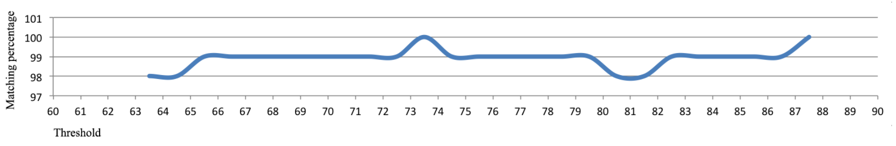
(c)



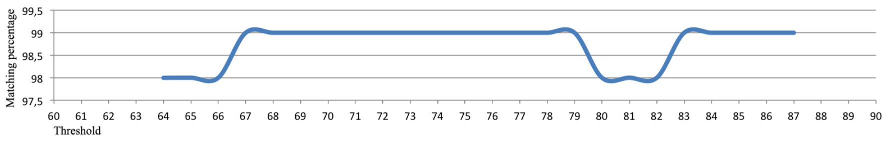
(d)



(e)



(f)



(g)

Figure 4.9: These graphs are obtained by increasing progressively the threshold distance in the matching phase. The first (a) is thus obtained with a distance of 2, the second (b) with a distance of 3 and so on.

4.2 BATCH TO STREAM TESTING

4.2.1 *Memory solution testing*

The solution detailed in the previous section has been implemented and tested with the following results:

TEST 11 The test has been executed by manually checking the amount memory occupied by the algorithm during the two phases of clustering and generalization. The test has been repeated a few time using different graphs.

As can be seen in Table 4.2, the memory consumption has decreased at least by 4 times, proving the adopted strategy effective.

Number of tests	10
Number of clusters (avg.)	217
Number of graphs	1000
Clustering memory usage (avg.)	< 300 MB
Previous clustering memory usage (avg.)	2 GB
Generalization memory usage (avg.)	< 500 MB
Previous generalisation memory usage (avg.)	2 GB

Table 4.2: Test 11: Memory performance

4.2.2 *Time solution testing*

The first testing performed is a performance testing: the two clustering algorithms have been run on the same data several times, and the total execution times have been averaged and confronted.

Number of tests	10
Number of graphs	1000
Number of generated clusters (avg..)	214
Average Jaccard execution time	277.255 seconds
Average Overlap execution time	265.749 seconds
Speedup	4.5%

Table 4.3: Test 12: Clustering performance

TEST 12 As can be seen from Table 4.3, the speedup is not really relevant since the goal was a magnitude order speedup. Furthermore in some of the test the Jaccard algorithm performed even better than the Overlap algorithm. The explanation is straightforward upon data observation: the first algorithm to be run always outperformed slightly the other, and this is surely due to the cache spatial localization which allowed the second algorithm to load the data faster. This means that the bottleneck is data loading, and the algorithm efficiency is not enough to represent a distinctive gain.

Since for the visualization part would be better to have a data structure with all the distinct FPs instead of a single, huge set, we decided to stick with the jaccard similarity, which results in terms of quality had already been tested.

4.3 VISUALIZATION TESTING

In order to test the visualization part Jackdaw has been run first on a small amount of binaries, allowing thus a small and highly readable map to be obtained. Later, the test has been repeated on a higher amount of samples in order to test the scalability.

4.3.1 *Visualization test*

In order to test the visualization part Jackdaw has been run on 1000 sample binaries in order to create a topological map of the behaviors. The results can be seen in Figure 4.10, in which behaviors appear as red nodes and binaries appear as blue or yellow nodes.

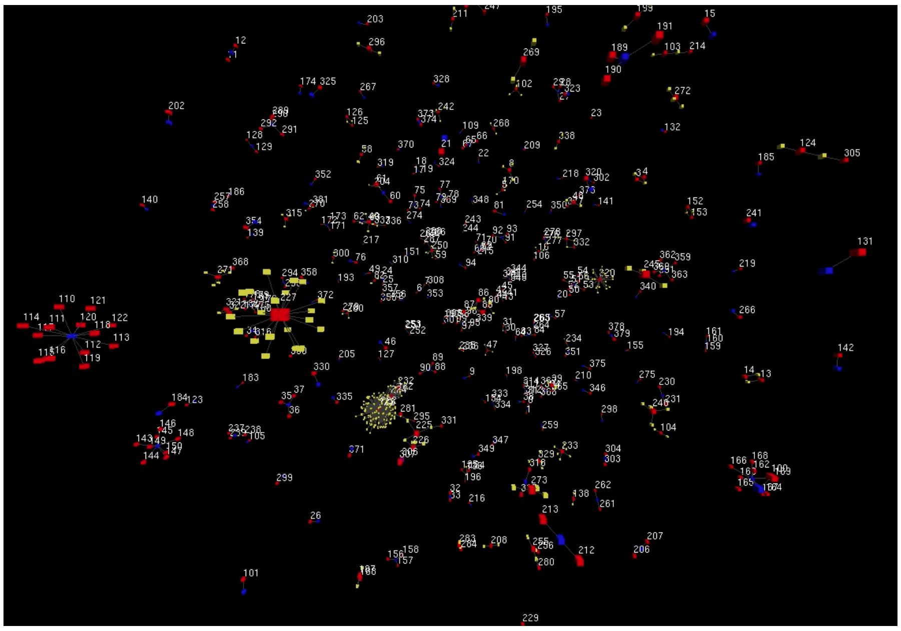


Figure 4.10: Test 13: Visualization

There are evidently behaviors that are not shared among many binaries, but this is acceptable considering the small amount of samples that have been used. On the other side, there are some examples of highly shared behaviors, like the one reported in Figure 4.11 which has been extracted from the map in Figure 4.10. Furthermore, there is a strong connection between behavior 222 and behaviors 225 and 232, since a significative number of binaries shares at least two of them.

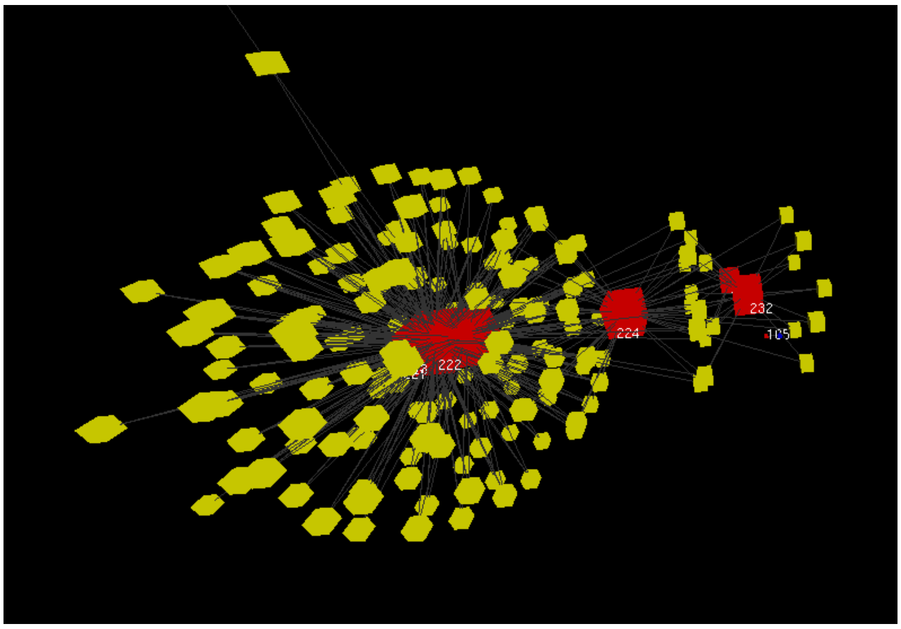


Figure 4.11: Highly shared behaviour

4.3.2 *Visualization scalability test*

Verified the the map is working as intended, the test has been repeated using more than 8000 graphs. The results can be seen in Figure4.12. It's immediate that the visualization algorithms does not scale in the least, since the map is unreadable. This evidently requires the algorithm to be re engineered in order to better scale.

4.3.3 *Scalable visualization algorithm*

The evident problem with the map it's the number or edges between the nodes: the number of edges is too high to allow a readable distribution of the weekly connected components of the map's graph. In order to reduce the number of edges, the easiest idea is to eliminate the fps nodes: their informative contribution is small, since there is always a correspondence one to one between FPs and behaviors.

However, this solution reduces the number of edges at most of a number equal to the number of behavior, which is far smaller than the number of binaries.

In order to build a really scalable visualization algorithm a solution should be found to reduce the number of edges down of some orders on magnitude. Possibly, also the number of nodes should be reduce, without sacrificing the expressive power of the map.

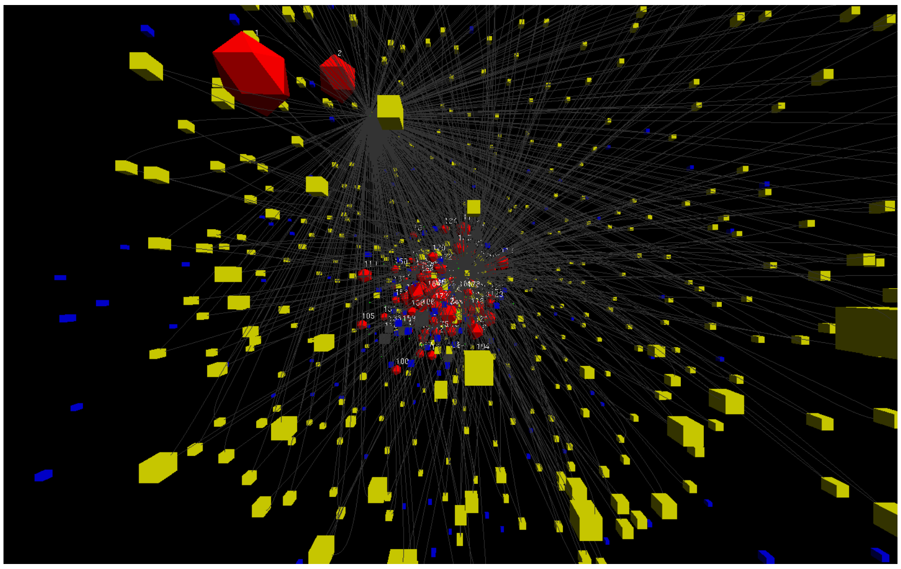


Figure 4.12: Test 14: Visualization scalability

The proposed solution is to build an algorithm that is able to recognize congruent nodes. A node A is defined congruent to another node B if and only if the group of nodes G connected to node A is equal to the group of nodes G′ connected to node B, i.e. if A and B have the exact same edges.

In this case, by rendering a single node AB the number of nodes and edges is halved once for each congruent node in the map. A label is added to the node with the number of congruent nodes represent as a single one, and the size f the node is proportional to this number, allowing the user to understand the magnitude of generating samples at glance.

The assumption that the number of congruent nodes rises with the number of samples analyzed is compatible with the assumption of Jackdaw that the number of behavior is limited and behaviors are highly shared among binaries.

The proposed idea has been implemented in the Algorithm 4.2.

4.3.4 *Scalable visualization testing*

The new visualization algorithm has been run on the same 8000 samples, and the results are reported in Figure 4.13. As shown, the map is now clear and readable, and the frame rate has improved from 1fps to 30 fps, allowing a fluider navigation.

Figure 4.14 reports a detail of the map, highlighting a structure in which 6 samples share behavior 6, which is also shared by 7 samples which also share behavior 4.

Algorithm 4.2 Scalable visualization

```
if 1==1: #loop here
    try: os.stat('clusters/')     #look for clusters
    except: os.mkdir('clusters/')
    clusterlist = os.listdir('clusters/')
    if '.DS_Store' in clusterlist: clusterlist.remove('.
        DS_Store')
    importClusters()
              for c in clusters:
        for s in c['source']:
            if s not in samples:
                samples[s] = dict()
                samples[s]['set'] = set()
            samples[s]['set'].add(c['id'])
    for c in clusters:
        if c['id'] not in clusternodes:
            clusternodes[c['id']] = next_id
          U.new_vertex_w_id(next_id)
          clusternodes[c['id']] = next_id
          next_id = next_id +1
          SetBehaviourNodeProperties(clusternodes[c['id']],
              len(c['source']))
    removals = list()
         for s in samples:
        conta = 1
        for s2 in samples:
            if samples[s] == samples[s2] and (s != s2):
                conta+=1
                removals.append(s2)
        samples[s]['conta'] = conta
    for r in removals:
        if r in samples:
            del samples[r] #comment this line to have the
                extended visualization
    for s in samples:
        U.new_vertex_w_id(next_id)
        samples[s]['id'] = next_id
        U.set_vertex_attribute( next_id, 'label', str(
            samples[s]['conta']))
        U.set_vertex_attribute(next_id, 'size', str(round(
            samples[s]['conta']/10,1)))
        if samples[s]['conta'] > 2:
            U.set_vertex_attribute( next_id, 'color', '#
                ffff00')
        else:
            U.set_vertex_attribute( next_id, 'color', '#0000
                ff')
        for c in samples[s]['set']:
            CreateNewEdge(next_id, clusternodes[c])
        next_id = next_id + 1
```

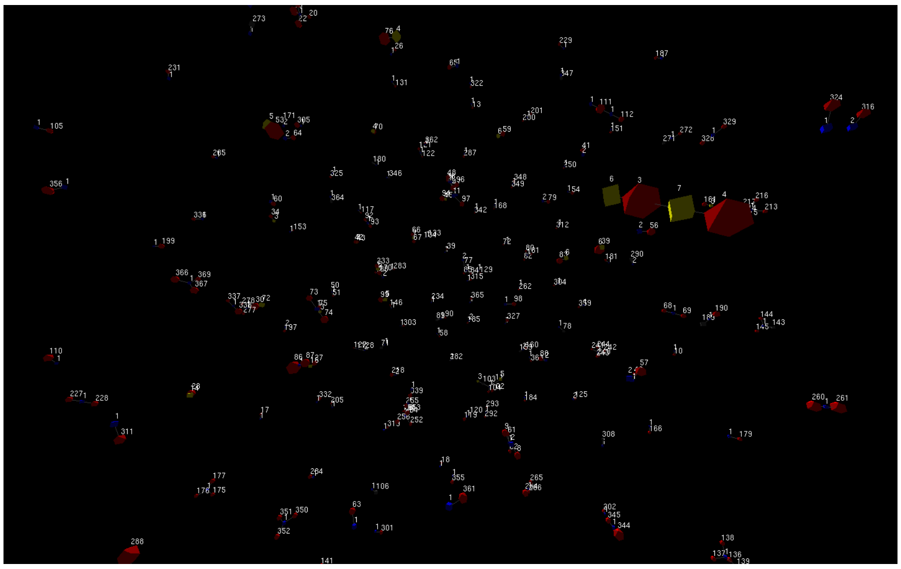


Figure 4.13: Test 15: scalable visualization

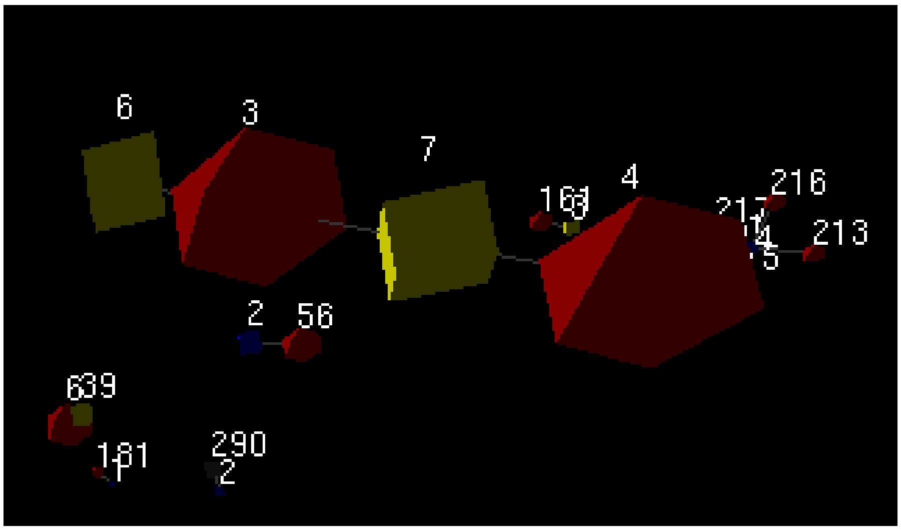


Figure 4.14: Test 15: Detail

5 CONCLUSIONS, LIMITS AND FUTURE WORKS

In this chapter a summary of the improvements provided to Jackdaw by this work will be presented, the existing limits will be highlighted and additional improvements that could be added in the future will be detailed

5.1 CONCLUSIONS

The most significative contribution of this work to the Jackdaw project is without doubt the development and implementation of a generalization algorithm which can perform in linear time (as proved in section 3.1.3). The previous generalisation algorithm performed in NP time complexity, which made Jackdow impossibile to be used in streaming due to the volume of sample provided.

The new generalisation algorithm shifts the bottleneck to the clustering phase, which can be parallelised to improve performance (see section 5.2), allowing an additional overall performance improvement.

Thanks to this new algorithm, it has been possibile to rewrite Jackdaw to work as a streaming service, and it is now able to process samples completely automatically until the final result. The streaming conversion allows to overcome one of the greatest limits of Jackdaw, i.e. the requirement of having multiple instances of the same malware to be able to extract the behaviour: working as a streaming service will allow this new Jackdaw to collect a lot of samples of every malware.

The other major contribution of this work to the Jackdaw project was the visualization system, which allows Jackdaw to be used in a completely new way: not only to analyse single malware samples, or batch of malware samples, but to monitor the whole malware ecosystem. Once a sufficient amount of samples will be submitted, the visualization tool will effectively map the malware behaviour ecosistem, allowing amalysts to monitor the emergence of new behaviours.

The visulisation system has proven easy to navigate and to understand, that was the main concern with such a feature, and has already show that there are commonly shared behaviours between different malware implementations, which was the fundamental assumption of the Jackdaw project.

5.2 LIMITS AND FUTURE WORKS

The main limit of this work is that it needs an interface, even a simple one, to allow users to easily submit samples and receive results. Building such an interface, anyway, is not a complex task since all the chaintool is ready and working.

The malware map could also be enriched by adding additional information and references. A tracebak algorithm has been sketched in order to allows to match a mouse click on a node to the object represented by that node, and could be exploited to convey additional information on that specific behaviour directly on the map. Such a feature has not been implemented in this work because additional information on the map reduced readability and understandability of the whole ecosystem, which was the main point in realizing a map. An aoutomatic system could also be developed to track changes in the malware map, effectively extracting information on the malware ecosystem evolution without requiring an analyst to examine the map.

To the performance side, the clustering algorithm can not be further improved, but it can be parallelized, be it locally or even better distributed on a network of machines. At that point, the generalization could also be split allowing each machine to generalise only a specific set of clusters.

BIBLIOGRAPHY

[1] Ubigraph. `http://ubietylab.net/ubigraph/`. (Cited on page 37.)

[2] Cuckoo sandbox. 2014. `http://cuckoosandbox.org`. (Cited on page 7.)

[3] Darren Mutz William Robertson Christopher Kruegel, Engin Kirda and Giovanni Vigna. *Polymorphic worm detection using structural information of executables*. Springer-Verlag, Berlin, Heidelberg, 2006. `http://dx.doi.org/10.1007/11663812_11`. (Cited on pages 5 and 7.)

[4] Christian Kreibich Juan Caballero, Chris Grier and Vern Paxson. Measuring pay-per-install: the commoditization of malware distribution. *Proceedings of the 20th USENIX conference on Security*, 2011. `http://dl.acm.org/citation.cfm?id=2028067.2028080`. (Cited on page 1.)

[5] Federico Maggi Paolo Mi-lani Comparetti Martina Lindorfer, Alessandro Di Federico and Stefano Zanero. Proceedings of the annual computer security applications conference (acsac). 2012. (Cited on page 5.)

[6] Alessio Massetti. Extracting common maliciuos temporal dependent behaviours from malware, 2015. (Cited on page 45.)

[7] Microsoft. Msdn. 2013. `http://msdn.microsoft.com`. (Cited on page 6.)

[8] Engin Kirda Clemens Kolbitsch-Christopher Kruegel Paolo Milani Comparetti, Guido Salvaneschi and Stefano Zanero. Proceedings of the 2010 ieee symposium on security and privacy. 2010. (Cited on page 5.)

[9] Mario Polino and Andrea Scorti. *Jackdaw: Automatic Behavior Extractor and Semantic Tagger*. 2013. (Cited on page 7.)

[10] Qun Song and Nikola Kasabov. *Ecm - a novel on-line, evolving cluster- ing method and its applications*. The MIT Press, 2001. (Cited on page 8.)

[11] Jiawei Han Philip S. Yu Xifeng Yan, Hong Cheng. Mining significant graph patterns by leap search. *In Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008. (Cited on page 12.)

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LYX:

`http://code.google.com/p/classicthesis/`

Final Version as of April 8, 2016 (`classicthesis`).

DECLARATION

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it contains no material previously published or produced by anothe party in fulfillment, partial or otherwise of any other degree of diploma at another University or institute of higher learning, except when due acknolegment is made in the text.

Como, April 2016

Federico Aleotti