Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Extracting common malicious temporal dependent behaviors from malware

Relatore: Prof. Stefano ZANERO
Correlatore: Prof. Federico MAGGI
Ing. Mario POLINO

Tesi di laurea di:
Alessio MASSETTI Matr. 795392

Anno Accademico 2014–2015

Ai miei genitori.

Acknowledgments

First of all I want to thank you the professor Stefano Zanero and Federico Maggi that helped me along with Ing. Mario Polino with thesis development during this year and a half. I want to give my special thanks also to all the students of NECSTLab for giving me an unique experience of teamwork during this thesis development.

I want to thanks also my teammates during this five years of studies at Polimi, in particular Andrea, Daniele, Francesco, Matteo and Mattia. Final thanks are dedicated to the rest of my family, in particular my parents that always believed in me leading myself to this important goal in my life.

Abstract

Malicious Software, from now on malware, *is any software that brings harm to a computer system.*

According to Pandalabs [19] 75 million new malware samples were observed during the last year, out of 350 million total malware specimens. This translates to 200,000 new malware samples every day.

Malware industry is rising as a real underground economy that generates huge illegal profits: stealing bank accounts, abusing credit cards number or penetrating email accounts. Malware samples are regularly sold on the market, and they can reach high prices. For this reason automatic malware analysis tools are strongly needed to optimize analysis time of new samples and understanding of their malicious behaviors.

Jackdaw [20], an automatic behavior extractor and semantic tagger, was built to address this need. Jackdaw is a tool that analyzes malware samples exploiting static and dynamic analysis procedures. Unfortunately, Jackdaw models are created and saved as *logical formulas,* and show some limits in preserving information about API calls numbers and taint dependencies between them. Jackdaw creates a basic model that does not allow, for example, to track the presence in the model of more files or more system resources in use in the analyzed behavior.

This thesis will focus on improving Jackdaw's model generation providing additional information with respect to the previous ones. Using static and dynamic analysis techniques we are going to generate taint dependencies between system calls and we are going to put them in a graph using taint dependencies analysis.

Our main goal will be to extract common behavioral models from clusters of mal-

ware, created by Jackdaw extracting common API call sequences that these shares. New behavioral models will be a graphs in which nodes represent API calls and edges dependencies between them.

The work presented in this thesis leads to the identification of 607 malicious behaviors models starting from a population of a large dataset of malware samples - those behaviors were divided into 37 groups of indistinguishable behaviors according to the old modeling system to prove the effective improvements in the quantity of the behaviors we can distinguish. Thanks to the model introduced in this thesis, the granularity of malware behavior distinguishable in our population increased of 85%.

Sommario

Un programma malevolo, d'ora in poi *malware*, è qualsiasi programma che può danneggiare un sistema informatico.

Secondo McAfee [13] nel 2013 ci sono 100.000 nuovi campioni di *malware* in circolazione ogni giorno. Questo numero è in costante crescita se pensiamo che l'ultimo report pubblicato dopo i primi tre mesi del 2014 da Science Magazine [11] lo ha ritoccato a 160.000 ed il report di Pandalabs [19] di fine 2014 parla addirittura di 75 milioni di nuovi *malware* usciti nell'ultimo anno su un totale di 350 milioni attualmente in circolazione: questo si traduce in 200,000 nuovi malware ogni giorno.

L'industria dei malware è quindi in costante crescita come una vera e propria economia nascosta che genera enormi profitti illegalmente: i programmi malevoli non crescono infatti solo in numero ma diventano anche più sofisticati grazie alle nuove tecnologie che permettono ad attaccanti malevoli di prendere il controllo della vittima grazie a nuove tecniche. Oltre ai nuovi *malware* i vecchi possono essere riscritti in nuovi, il più delle volte utilizzando strumenti automatici. [9, 23].

I malware generano quindi profitto tramite il furto di credenziali bancarie, quello dei numeri delle carte di credito o più semplicemente tramite l'intrusione in account di posta elettronica. I *malware* sono quindi venduti proficuamente anche sul mercato: per esempio un campione di LusyPOS è stato venduto per 2000 dollari americani [1]. Non è difficile quindi immaginare il ritorno economico che è possibile avere con l'utilizzo di questi. [3]

Per questa ragione sono fortemente necessari degli strumenti per l'analisi automatica di *malware* al fine di ottimizzare il tempo di analisi dei nuovi campioni e comprenderne il loro comportamento malevolo. I nuovi campioni che escono ogni

giorno non possono essere analizzati manualmente.

Per soddisfare questa necessità è stato costruito Jackdaw [20], un estrattore automatico di comportamenti malevoli. Jackdaw è uno strumento di lavoro che analizza gli esempi di *malware* tramite tecniche di analisi statica e dinamica. I malware analizzati sono quindi clusterizzati su base comportamentale: ogni singolo cluster è caratterizzato da un comportamento che è presente in ogni singolo *malware* del cluster.

Sfortunatamente, *Jackdaw* estrae un modello comportamentale come formula logica e presenta delle limitazioni nel mantenere informazioni sulle chiamate ad API e le dipendenze tra queste interconnesse: *Jackdaw* crea un modello di base che non permette, ad esempio, di tenere traccia della presenza di vari file o differenti risorse di sistema utilizzate contemporaneamente dal comportamento analizzato.

Questa tesi si focalizza sul miglioramento delle procedure di generazione dei modelli di *Jackdaw*, proponendo più informazioni rispetto alla versione precedente. Analizzeremo più specificatamente le limitazioni presenti nell'attuale sistema di modellizzazione e vedremo quali strumenti possiamo usare per superare le limitazioni presenti nei modelli di Jackaw. Questo nuovo sistema di modellizzazione ci permetterà, quindi, di introdurre nuove caratteristiche rispetto come il tenere traccia di chiamate multiple della stessa API e delle dipendenze tra queste.

Il lavoro qui illustrato ha quindi portato all'identificazione di 607 modelli di comportamenti malevoli partendo da una popolazione sufficientemente larga di *malware* (3112 campioni). Questi comportamenti sono stati divisi in 37 gruppi di comportamenti non distinguibili secondo il vecchio modello, per provare l'effettivo miglioramento della quantità dei comportamenti che possiamo distinguere. Grazie al modello presentato in questa tesi la granularità dei comportamenti di *malware* distinguibili all'interno della popolazione di *malware* campione è aumentata dell'85% rispetto al modello precedente presentato in Jackdaw.

Contents

1 Introduction **1**

2 State of the Art **5**

2.1 Static and Dynamic Analysis 5

2.1.1 Static Analysis 5

2.1.2 Dynamic Analysis 6

2.2 Jackdaw 7

2.2.1 API call List extraction 7

2.2.2 Data Flow Dependency 7

2.2.3 Fingerprints and common behaviors 8

2.2.3.1 Fingerprints 8

2.2.3.2 Fingerprint matching 9

2.2.4 Clustering 9

2.2.5 Model creation 10

2.2.5.1 Parameter Models 10

3 Proposed Approach **13**

3.1 Problem Statement 13

3.2 Clustering results preprocessing 14

3.3 Model generation using paths 14

3.3.1 Paths approach assumptions 16

3.3.2 Matching improvements 18

3.4 Parameters model generation 21

4 Implementation details **23**

4.1 Path Generation 23

4.2 Common Behavior Search 25

4.2.1 Node initialization 27

4.2.2 Path searching 28

4.2.3 Path Matching 30

4.2.4 Node flagging 30

4.2.5 Parameters model creation 31

4.3 Graph Export 32

5 Experimental results **37**

5.1 Dataset 37

5.1.1 Cluster distributions 39

5.2 Model Analysis 43

5.2.1 Example 1 43

5.2.2 Example 2 44

5.2.3 Comparison with Jackdaw 47

5.3 Behavior matching comparison 47

5.3.1 To compare the old and the new system 48

5.3.2 Differences in behavioral models 51

5.3.2.1 Node similarity 51

5.3.2.2 Arc similarity 52

5.3.2.3 Similarity results in models 53

5.4 Performances 54

6 Conclusions **59**

Bibliography **63**

List of Figures

1.1 Malware data after Q1 2015 2

2.1 Summary of Jackdaw 8
2.2 Clustering process 10

3.1 Graph samples 15
3.2 Path models 17
3.3 Graph Ambiguity Cases 18
3.4 First Graph Matching 19

4.1 Data structure initialization 28

5.1 Cluster generation progression 40
5.2 Graphs distribution 42
5.3 Nodes distribution 43
5.4 Sample 1 Graph 45
5.5 Sample 2 Graph 46
5.6 Similarity Distributions 54
5.7 Clusterization Time 56

List of Tables

3.1 Paths of the Graph Samples in Figure 3.1 16
3.2 Paths Points in the Graphs 20

5.1 Behaviors Analysis Results 39
5.2 Behavior results having more than one API sample 40
5.3 Clusters data 41
5.4 Cluster distribution 42
5.5 API call presence in models 48
5.6 Behavioral Groups 49
5.7 Behavioral Differences 57
5.8 Similarity Results 58

List of Algorithms

4.1 Path Calculation 24
4.2 Minimum Behavior Search 26
4.3 Node initialization 27
4.4 Points Computation 29
4.5 Searching for path in other graphs 29
4.6 Path searching in a single graph 30
4.7 Path Matching in a Graph 31
4.8 Node flagging in a path 32
4.9 Parameters model creation procedure 34
4.10 Graph Export 35
5.1 Graph inequality algorithm 50

Chapter 1

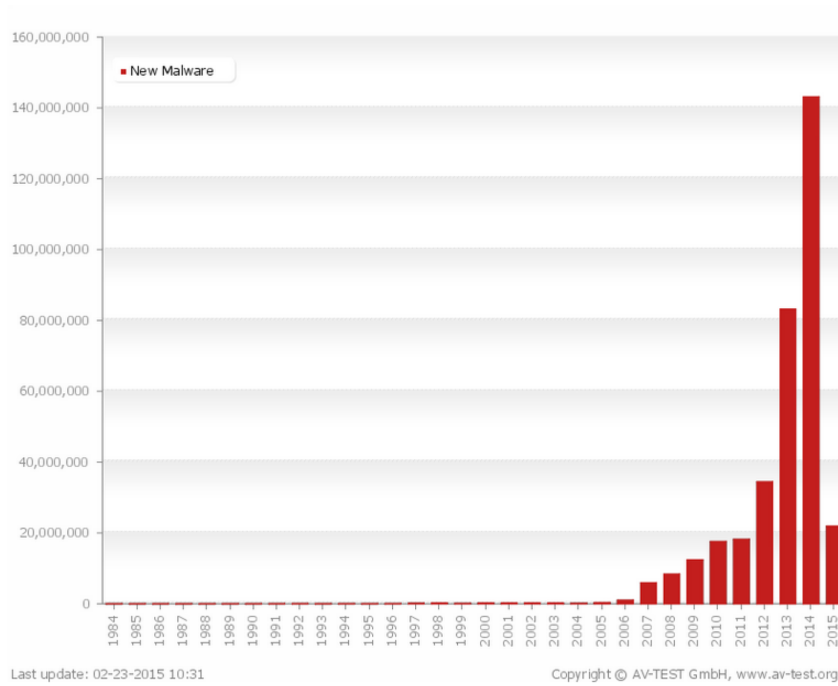
Introduction

Malicious Software, from now on malware, *is any software that brings harm to a computer system.*

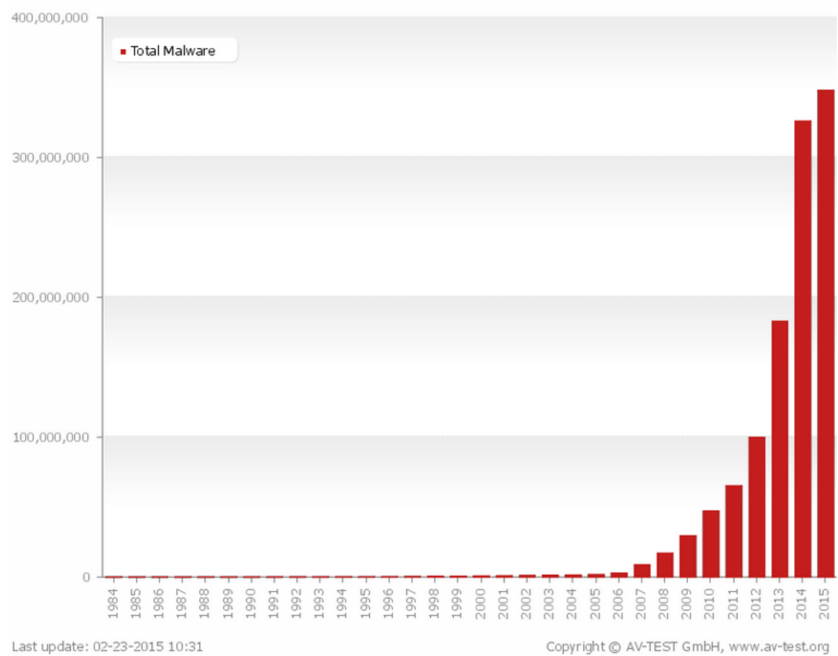
According to McAfee [13] there were 100,000 new malware samples every day in 2013. This number seems to be constantly increasing: the latest report, published Q1 2014 by Science Magazine states that 160,000 malware specimens are created every day [11] while Pandalabs [19], at the end of 2014 speaks of 75 million new samples in the last year, with a total of 350 million, which translates to 200,000 new ones every day. Figure 1.1 significantly evidentiates this fact.

Malware industry is rising as a real underground economy that generates huge illegal profits: malware samples don't grow in number only, but also become more sophisticated, thanks to new technologies that allow malicious attackers to get control of the victim using new techniques. In addition to new malwares samples old ones can be mutated into new ones, most of times using automated tools [9, 23].

Malware generates profit trough stealing bank accounts, abusing credit cards number or penetrating email accounts. Malware samples are also profitably sold on the market: for example, a LusyPOS sample was sold on underground markets for 2000 US dollars [1]: it's not difficult to imagine the economic return of using such samples. [3]



(a) New malware samples every year after Q1 2015



(b) Total Malware in circulation after Q1 2015

Figure 1.1: Malware data after Q1 2015

For this reason automatic malware analysis tools are strongly needed to optimize analysis time of new samples and understanding of their malicious behaviors. New samples that cames out every day cannot be studied manually.

Jackdaw [20], an automatic behavior extractor and semantic tagger, was built to address this need. Jackdaw is a tool that analyzes malware samples exploiting static and dynamic analysis procedures. Analyzed malware are clustered together on a behavioral basis: every single cluster is characterized by behavior that is present in each malware sample in the cluster.

Unfortunately, Jackdaw extracts a model of such behavior as a *logical formula* and presents some limitations in preserving information about the number of API calls, and taint dependencies between them. Jackdaw creates a basic model that does not allow, for example, to track the presence in the model of various files or different system resources in use.

This thesis will focus on improving Jackdaw model generation, providing additional information with respect to the previous one. We are going to analyze the limitations of the current modeling system and see what tools we can use to overcome existing limitations in Jackdaw models. The new behavioral model will be a graph in which nodes represent API calls, and edges dependencies between them. This new modeling system allows us to introduce new characteristics such as tracking multiple calls of same API and dependencies between API calls.

The work presented in this thesis leads to the identification of 607 malicious behavior models, starting from a population of a large dataset of malware samples (3112 samples). Those behaviors were divided into 37 groups of indistinguishable behaviors according to the old model, which shows the effective improvements in the quantity of the behaviors we can distinguish. Thanks to the model introduced in this thesis, the granularity of malware behavior distinguishable in our population increased of 85% than using Jackdaw old modeling system.

The rest of this document is summarized as follows: after the introduction in Chapter 2 we will describe the challenges of malware analysis, summarize Jackdaw and present how dynamic behaviors are extracted, introduce the concept of finger-

print and describe how malware is clustered together. In Chapter 3, we will see how the proposed approach can synthesize malicious behaviors to produce behavioral models. Chapter 4 will describe the implementations details, Chapter 5 experimental results of our approach while in Chapter 6 we will see limitations and future works.

Chapter 2

State of the Art

In this chapter we present Jackdaw and show how using this tool behaviors are extracted and clustered together.

2.1 Static and Dynamic Analysis

As stated in the Introduction tools of automatic malicious software analysis are needed. Given a malicious software sample, two basic type of analysis can be performed on it: static and dynamic. Let's see briefly how they are performed and what are the advantages and disadvantages of each technique, to understand how they can be useful for our purposes.

2.1.1 Static Analysis

Static analysis consists in using analysis techniques that does not need to execute the file.

Usually, we assume to be able to obtain a readable assembly code using a disassembler, and look at the entire sample composed by its various basic blocks[1]. Basic blocks are connected in control flow graph. We can also produce pseudo-code that tries to represent the original code (compilation is not usually reversible). We can look at the *whole* program, even parts that would not be promptly executed and contain hidden functionalities: an analyst can find out every aspect of the malware.

[1] A basic block is a group of instruction that is always executed consecutively

Unfortunately, this is not an ideal world and there are some strong limitations to this type of approach: strong code obfuscation techniques can prevent the disassembling itself [10]: for example, uncommon instructions such as opaque constants calculations, or using a lot of jump instructions. There are also techniques of code compression and code packing that encrypt the code and allows to see executed in clear only a little part of it avoiding the unpacking of the whole malware for the execution: code is decrypted and/or decompressed only at run-time, and not always in the same way. [15] In fact, code can be dynamically generated too, obtaining many different samples of the same malware, simply changing the algorithm or using a different encryption key: this phenomenon is called *polymorphism*. Static Analysis can partially be performed by automatic tools - in our work we use *disasm [8]* by C. Krugel to analyze statically the control flow graph of malware[2].

2.1.2 Dynamic Analysis

Dynamic analysis consists in letting malware execute and tracking what happens. Obviously, executing malware in a device compromises the device itself, so we use a sandbox. A sandbox is a security mechanism to separate running programs, often using virtual machines [6]. Often anti-viruses themselves had a sandbox that use to dynamically analyze malware that are not in database or software that they do not trust (Sandbox Analysis).

Dynamic analysis allows to track all the modifications that are done on the system trough, for example, Windows API calls, as wells as network connections instantiated by the malware. Sandboxes such as Cuckoo, track the calls before they happened with hooking techniques: Other techniques simply look at the difference of system images before and after malware execution.

The big limitation of this approach, differently from static analysis, is that not all the code is tracked, but only the part that is effectively executed in the sandbox. On the other hand, dynamic analysis is immune to obfuscation attempts, and has no problems with self-modifying programs [2].

[2]see paragraph 2.2.3.1

2.2 Jackdaw

Jackdaw [20] is an automatic behavior extractor and semantic tagger. It is used to extract malicious behaviors from malware samples and it is the tool we want to improve. We can define a behavior as *a group of different API calls sequence that realize a specific goal of the malware.*

In the following paragraphs, the main concepts in the behavior extraction and clusterization that we used for our purposes are presented. Malware is sent for analysis to both Cuckoo (Dynamic Analysis) and Disasm (Static Analysis): results are combined to generate taint analysis of malware and at this point behavior is ready for clustering phase. This procedure is summarized in Figure 2.1.

2.2.1 API call List extraction

The first part of Jackdaw concerns about dynamic analysis of the malware sample, extracting the Windows API calls [14] executed trough hooking techniques. This part of work in Jackdaw is done by Cuckoo Sandbox and as output is provided the complete API call list executed by malware in the Sandbox API calls, parameters included. At this stage those list of API is provided us without taint dependencies between API calls..

2.2.2 Data Flow Dependency

The process of behaviors identification is performed exploiting dataflow dependency techniques similar to the ones already presents in literature. As general rule it is possible to define that a *dependency exist between instructions A and B when instruction B execution requires data produced by instruction A [5, 16, 18].* Parameter propagation between the various System Calls is traced to reconstruct the call hierarchy detecting those requirements.

Exploiting data flow dependency, Jackdaw bound the API calls list into a graph that represents the malware behavior.

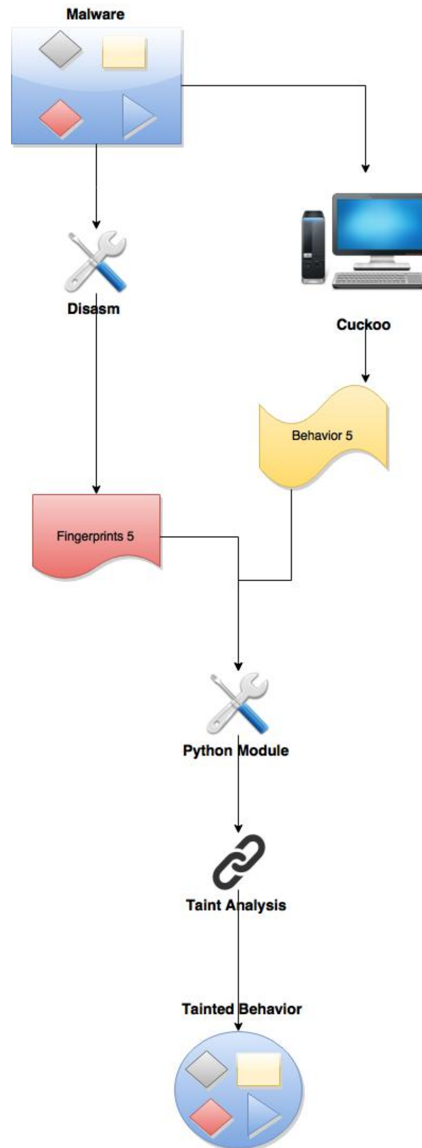


Figure 2.1: Summary of Jackdaw

2.2.3 Fingerprints and common behaviors

2.2.3.1 Fingerprints

Fingerprint generation is based on the Control Flow Graph of the process: the main CFG returned by data flow dependency is divided into sub-CFG that have a dimension of k[3] nodes: one CFG node is a basic block of instruction.

Our target is to map the code portions recognizing similar ones in different parts of malware or different malwares: every sub-CFG generates one or more strings

[3] In Jackdaw, coefficient k, in our procedures set to 10 like in Krugel original paper [8]

associated uniquely to its structure: those strings are called *fingerprints*.

2.2.3.2 Fingerprint matching

Disasm results are produced by static analysis on the sample provided, behavior are provided by Cuckoo Sandbox on dynamical analysis. The lasts are sent to a Python module of Cuckoo, that is used at this point to match fingerprints to API call list. Code analyzed in this way is mapped by using virtual memory addresses: basic blocks are marked with an initial and a finish address, all API calls whose caller address is between those block limits is mapped to the basic block and consequently presents the same fingerprints.

Fingerprints are so inserted in the final malware behavior: they are memorized as a list in the node we will process. The process requires malware to be unpacked.

2.2.4 Clustering

At this point of the analysis we have the behavior results for each sample we analyze. Our purpose, now, is to cluster together similar behaviors. In this phase all the behaviors obtained at point 2.2.2 are merged and processed subsequently independently from the malware sample they belong to.

Sub-graphs obtained are clustered according to similarity of their set of fingerprints using ECM Algorithm [22]. Using Jaccard Similarity [4], sets are clustered into a dictionary, in which fingerprints are the key to access the set and the list of graphs that matches those fingerprints are the values. Clustering phase is summarized in Figure 2.2. Form of the polygons in tainted behaviors represents various behaviors of an hypothetical malware analysis: similar forms represents similar behavior, different color are used to represent the fact that those behaviors are similar but not identical.

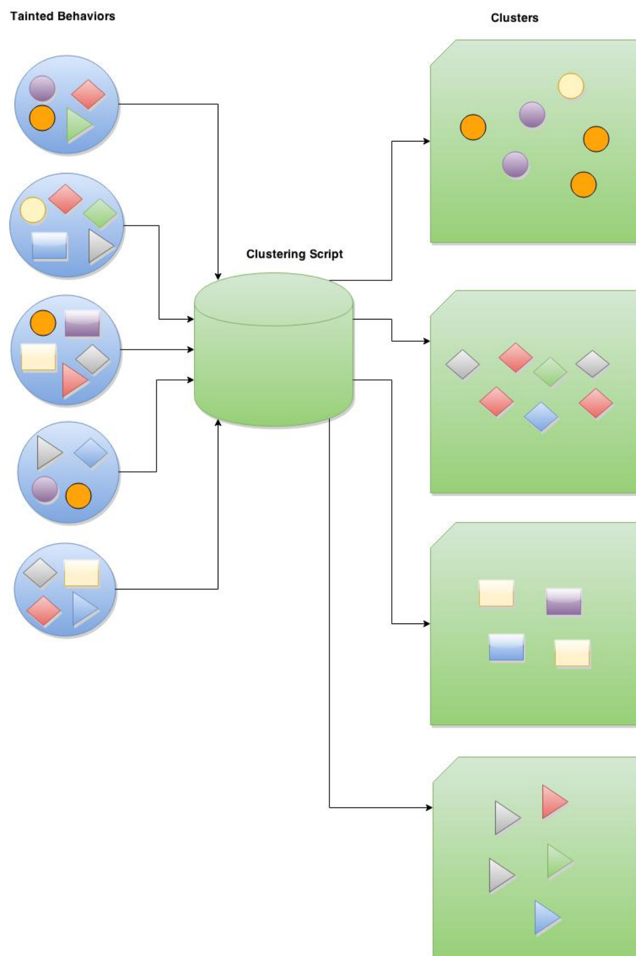


Figure 2.2: Clustering process

2.2.5 Model creation

In Jackdaw, after the clustering phase behavior were extracted using the *Most Frequent Rule API extraction*.

The cluster behavior is represented as a set of API functions. The API functions chosen are the ones that appears in 70% of cluster behaviors. The choice of 70% is an empirical choice, based on experiments.

2.2.5.1 Parameter Models

After this step a model for parameters of each API in the final behavioral model is calculated. These are the three model types we import from Jackdaw: we are going to briefly describe them, then we look how a new parameter can or cannot

match with these models.

- **Token Model:** The token model creates a set of all the possible values of the parameter. A new parameter match with this model if parameter value is present in the set provided by the model itself.
- **String Model:** Strings are modeled according to length: a mean and variance are generated. Matches with new samples are given according to Chebyshev inequality [12, 17]
- **Ip Model:** Matches are done with the IP regular expression. Essentially the purpose of this model is to classify IP addresses in the three IP classes: local, private and public

The parameters models are associated to each API call provided after their generation.

Chapter 3

Proposed Approach

In this chapter we will see how clusters are processed to generate the final behavior models. In section 3.1 we will analize Jackdaw limitations, and set our target to overcome them. In section 3.2 we show what pre-processing steps we introduce in the clustering phase. In section 3.3 we will show our modelization choices and process, in the subsection 3.3.2 we will see how to optimize them during the modelization process. At the end, in section 3.4 we will see how Jackdaw API call parameters are reintroduced in our model.

3.1 Problem Statement

We have seen in section 2.2.5 how Jackdaw generates model, unfortunately model generated with this approach have strong limitations.

The main limitation we want to overcome in this thesis is that API call order and taint dependencies are not taken into account in the behavioral model: Jackdaw, in fact, tracks only the presence or the absence of API calls.

This implies, for example, that we cannot distinguish multiple instances of same actions defined by API calls, like opening or writing in a specific file multiple times, but only the presence or the absence of those actions. Our target is to represent the malware behavior using a new concept of behavioral model, that permits us to take care of API call repetitions and behavioral taint dependencies using the minimum number of API calls possible.

It must be noted that behavior instances in most cases are not composed by the precisely but rather share some common alities that we want to model in a better way than a simple API call set. In other words, we have to understand those differences and eventually «merge» them. This was achieved by Jackdaw with the 70% treshold.

3.2 Clustering results preprocessing

We choose to discards some behaviors into a preprocessing step.

- **Behaviors with one node.** Our decision is not to consider these behaviors because they are simple and already «modeled»
- **Behaviors with 100 or more nodes.** As we are going to see in the next paragraphs, our procedures admits as hypothesis that graphs are relatively small as we are going to transform it into trees and then generates all the paths. This is true for most of the malware behaviors but not for all. For performance reasons, big graphs are discarded
- **Behaviors with no fingerprints.** All sub-graphs where all API calls has no fingerprints are automatically discarded. We cannot clusterize any behavior if there are no fingerprints in it according to paragraph 2.2.3.2.

3.3 Model generation using paths

We have seen in the previous section that after the clusterization phase every cluster is composed by a set of directed graphs, where nodes represents the API calls and edges represent taint dependencies.

To create the behavioral model we follow this rule: *if one or more API calls are in the final behavioral model, they must be, at least in the same quantity, in all the graphs from which the model is generated.* What we are doing is a sort of «graph intersection», starting from the assumption that nodes with the same API calls can be matched. Obviously, this means that the final result will be smaller than the smallest graph in the cluster in terms of number of API nodes.

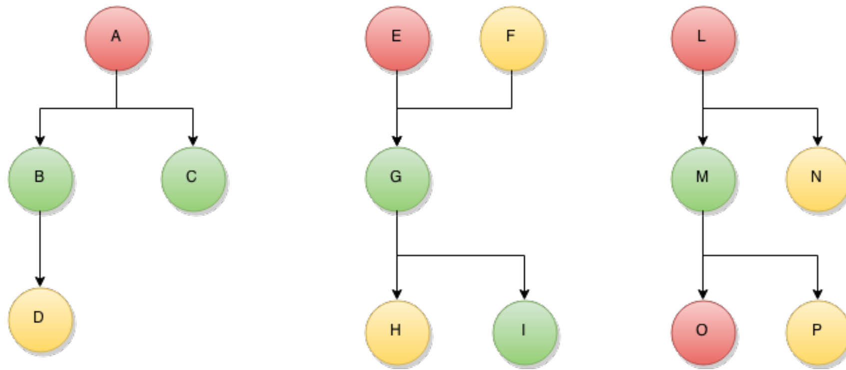


Figure 3.1: Graph Samples

Let's consider the example in Figure 3.1. Each circle represents an API call, with same colors represents the same call. Each arrow represent a taint dependency. We first identify root nodes and leaves in the directed graph, then generate a list of all the possible paths that the graph provides: every graph has to be reconsidered in terms of root nodes, leaf nodes and paths. We can define those concept in this way:

- **Root node:** a node that has only outcoming edges
- **Leaf node:** a node that has only incoming edges
- **Path:** an ordered node list between a root node and a leaf node. Node i and $i+1$ in the list should have a directed connection from node i and node $i+1$ in the graph.
- **Sub-path:** every subset of the path list

We reason over path because we assume that each path represents the minimal unit of behavior we can consider without losing information on dependencies.

Nodes are processed according to the API call they contain: nodes that have the same API call are considered as matching. This aspect creates a of possible ambiguity in the creation of our behavioral model that we will see in the Chapter 5.

Model creation can be done using two options created by design choice. First one computes only the paths, the second one add to the set of the computed paths, also the sub-paths as shown in Table 3.1. In this last case, we can add in the final model

also part of paths that matched between themselves from different models. At the moment this is only a design choice: we cannot say between the two options what is the best one, only that the second one will provide obviously large models than the first.

In both cases, the result we want to obtained from the example provided in Figure 3.1 is to match paths A-B-D from Graph 1, E-G-H from Graph 2 and L-M-P from Graph 3.

We define «common paths» every path that contains the same API calls in the same order in all the graphs. As we are searching for common paths we start from one graph and one of its generated paths - let's say that in this case we want to match path A-B-D, that means «red-yellow-green» API calls. We check in the list of generated path for the other graphs if there is a path with that condition in Graph 2 and Graph 3 and, if there is, path is added to behavioral model, in other word *the final graph is constituted by the set of the common paths of the cluster graphs.* Same API call can be contained into different paths, this information is maintained during the path generation phase in a way that permits us to reconstruct all the dependencies during the final behavioral export, building a new graphs from the set of the common paths we have.

3.3.1 Paths approach assumptions

In this sub-paragraph we want to see if our modeliazion choice are valid, what are the consequences of our simplifications and if they can be accepted.

We can start doing it looking at first graph of Figure 3.1. With procedures exposed in the previous section two paths are generated from this graph: A-B-D and A-C: those paths are independent one by another. In fact, even if nodes B and C

Table 3.1: Paths of the Graph Samples in Figure 3.1

	Graph 1	**Graph 2**	**Graph 3**
Paths	A-B-D; A-C	E-G-H; E-G-I; F-G-H; F-G-I	L-M-O; L-M-P; L-N
Sub-Paths	A-B; A-C; B-D;	E-G; F-G; G-H; G-I;	L-M; L-N; M-O; M-P

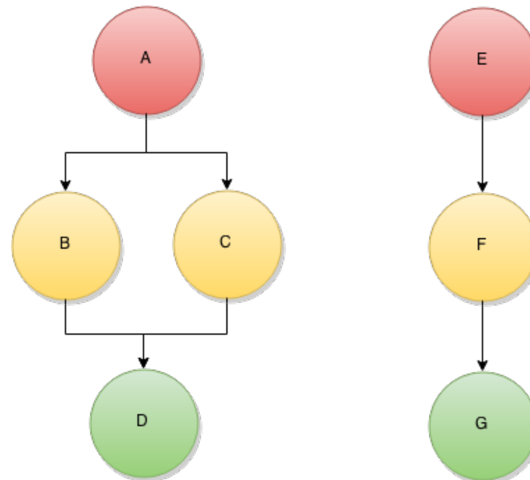


Figure 3.2: Path models

are both connected to node A, the taint dependency between them simply means they need the information A provided, but for different purposes. The information will be, in fact, processed in two different ways, the first with node B and D and the second with node C, but what is done with the API call in nodes B and D does not affect node C and vice-versa. Reasoning by paths has no influence in the case information produced from one API call are needed for two of more subsequently API calls. There are no side effects during the processing phase of information that those path shares.

In Figure 3.2 is shown, on the other hand, the opposite case in which an API call (D) depends from information generated from different API calls (B-C). We assume that these two graphs were clustered together, so they represent two behaviors that are similar between them and we want to merge. Reasoning by paths we see that D depends from both B and C in the left graph, and its equivalent API call G depends only from F in the right graph. Obviously, A and D are the same API calls of E and G and probably do the same things - the difference is the API calls B and C in the first graph and the only API call F in the second graph. There are three possibilities to explain this in a real case:

- API call F do the same things as B and C combined

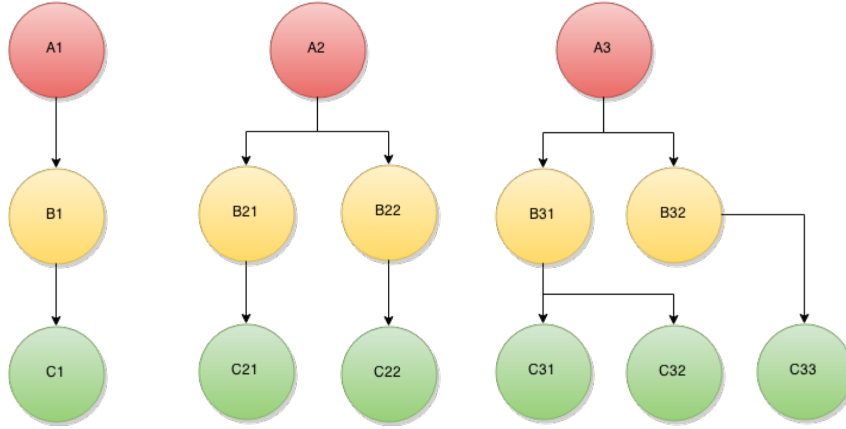


Figure 3.3: Graph Ambiguity Cases

- F does the same things of the API as B, and C is useless or junk code
- F does the same things of the API as C, and B is useless or junk code

We cannot decide in which of these three cases we are, thus A-B-D or A-C-D can independently be chosen to match with E-F-G.

A mistake can be produced in the parameters model of the API chosen, but this cannot be avoided. No errors are introduced in the final API calls model.

3.3.2 Matching improvements

We can improve the matching process avoiding ambiguities and repetitions. Let us first consider Figure 3.3

If we start processing from graph 1, we have one «A-B-C» or «red-yellow-green» path, and we will match that path with the first path we found in the graph 2 and 3, i.e. A2-B21-C21 and A3-B31-C31. On the other hand, if we start processing from graph 2 we have two paths «A-B-C» to be matched which is redundant. In other words, *path matching operation should track memory of paths already matched.* To solve this, we introduce a «read» flag: when a path is matched, all nodes on the matched paths are marked as «read».

Let's suppose we start processing from graph 2 and we matched A2-B21-C21 path with A1-B1-C1 and A3-B31-C31 path. The situation after the first matching,

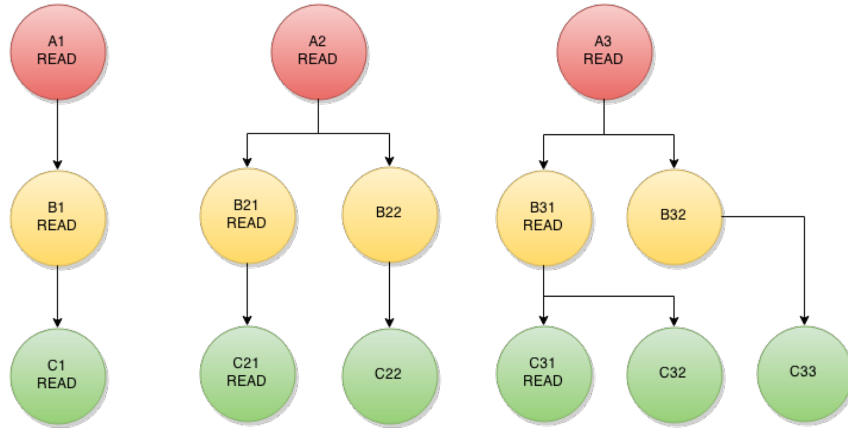


Figure 3.4: First Graph Matching

now, is the one in Figure 3.4

At this point, the algorithm checks for a path that can match with A2-B22-C22, meaning a path that has a Red API call with a read flag, Yellow API call without read flag, Green API call without read flag. This path cannot match with A1-B1-C1 for the second time, because all of three APIs are marked as read from the previous match. The path A2-B22-C22 so matches only with A3-B32-C33 and having no match in graph 1, is not added to result paths. In other words, introducing a read flag avoid redundant matches of the paths composed by same API calls.

We can assume, trivially, that A3-B32-C33 is a «better» match for Graph 3 than the previous one, because B32 node has one child as B1, differently from B31 that has two: in other words, *we try to match paths not only from their API but, where possible, also from the number of taint dependencies presented in the path.*

After taking into account node readability we can overcome the second limitation and make every path matching with a path that not only has the same API but has the similar number of taint dependencies possible to the path we want to be matched. To do this task, another abstraction level in our data is added, this time at path level: every path has given a number that is generated according to his heuristic formula in this way

$$\sum_{d=1}^{k}(n*d*10)+d$$

Where

- d represent the depth of the node in the path. As nodes in the path are stored from root to leaves, the first node has $d = 1$ and the last node has d equal to the path length
- n represent the number of the neighbors of the current node in the process

Applying to the path generated from Figure 3.3 the results are the one that follows in Table 3.2

In this way paths A3-B31-C31 and A3-B32-C33 are different because their API calls are different. The path matching will be turned into two separate steps:

- **Step 1:** match all the path with same APIs and same node read conditions in other graphs
- **Step 2:** from the paths found at previous step, if multiple matches are available, choose the one that minimize $\Delta points$ (difference between path to match and candidate path for matching)

Path A1-B1-C1 will be matched with A3-B32-C33 in graph C, because its point difference is less than the A3-B31-C31 path, even if this difference exists and there will be some minor differences in API calls just because A1 has one dependency and A3 has two. And about graph 2, we do not care what path will be matched between

Table 3.2: Paths Points in the Graphs

Path	Step 1 (A node)	Step 2 (B node)	Final Step (C node)	Total Points
A1-B1-C1	$1*1*10+1=11$	$11+1*2*10+2=33$	$33+3=36$	36
A2-B21-C21	$2*1*10+1=21$	$21+1*2*10+2=43$	$43+3=46$	46
A2-B22-C22	$2*1*10+1=21$	$21+1*2*10+2=43$	$43+3=46$	46
A3-B31-C31	$2*1*10+1=21$	$21+2*2*10+2=63$	$63+3=66$	66
A3-B31-C32	$2*1*10+1=21$	$21+2*2*10+2=63$	$63+3=66$	66
A3-B32-C33	$2*1*10+1=21$	$21+1*2*10+2=43$	$43+3=46$	46

A2-B21-C21 and A2-B22-C22: for our purpose, having the same composition, they are the same path and we will accept one of two randomly, discarding the other.

3.4 Parameters model generation

After the generation of the behavioral model with this procedure, we have a graph in which every node represents an API call. Those API calls lacks of parameter modeling: in this step we have to recreate parameters model using paradigms provided by Jackdaw, explained in the previous chapter in paragraph 2.2.5.

After matching procedures our graph is composed by a list of all the paths that the graphs presents in the cluster have in common. For each one of those paths, we stored apart a list of the paths that have matched between themselves during algorithm execution: let us think, for example, at Figure 3.1: A-B-C matched with F-G-H and L-M-P. We actually have in the final graph information about the fact that there is a path in common composed by three API calls of «red-green-yellow» type in that specific order: we don't have any information about parameters that those API calls used during their execution.

Those paths, from now called *stored paths*, are, by construction, of equal length and composed by the same type of API calls in the same order. In our parameters model generation, models of parameters is thus done by confronting together API calls that stands at the same order in every stored path: this procedure is repeated for every group of stored paths. This means that as we stated in paragraph 3.3.1 we choose to model each API call in each common path with one and only one API call in each graph.

After this generation we isolate every element of all the lists of all the stored paths in order. In the previous example, for «red» type of API call, we process together node A, node F and node L from the three graphs to infers their parameters and generate a model for each one of them. Having the API type and all the occurrences in the stored paths, next step is to obtain names of the parameters to model for that API calls. Sharing these API calls the same type, all the parameters names and

types of the values that those parameters assumes will be the same for all the API calls of the same type: i.e. we infer API calls names and type from API call A, then use them to obtain also the values in F and L from known names and types.

There are some parameters that we does not want to model: we know *a priori* their names and we have inserted them in a blacklist: all the parameters with a name in this list are automatically excluded from this process.

Acting as described before, at the end, for every parameter, we have a list of all values that every parameter assumes in every occurrences for every API call in the path. The last step is send the list of value to Jackdaw, calling the right Jackdaw API call to obtain the desired type of model: in our algorithm we modeled parameter values according to the following list.

- If the type of parameters values is not a String or an Integer the parameter list is not modeled.
- If the type of parameters values is an integer the parameter list is modeled as a *Token*
- If the type of parameters values is not an integer is a String. We will search with regular expression if there are any String patterns to distinguish if is an Hex Address, an IP or a String
  - If the type of parameters values an Hex address pattern is matched, the parameter list is not modeled
  - If the type of parameters values an IP address pattern is matched, the parameter list is modeled as *IP*
  - If the type of parameters values nor an Hex or an IP address pattern is matched, the parameter list is modeled as a *String*

We are not interest in modeling Hex address because information provided in this part are not relevant

Chapter 4

Implementation details

In this chapter we will show the implementation details of the solution we propose. In paragraph 4.2 we are going to see details of relevant part of our processing, then in part 4.3 we are going to see the export process of final results using graphs. All those algorithms are coded using Python 2.7[1].

4.1 Path Generation

We have seen how clusters are created. Now, in Algorithm 4.1 we will see how paths are calculated from each cluster. Algorithms to find root and leaf nodes are very trivial, they simply search for all nodes and checks edge condition as defined marking them as roots or leaves. The path generation between two nodes is managed by *Networkx* Python libraries. In addition, this computation all the one who follows are done in parallel for each cluster: algorithm 4.1 requires a single cluster as input and during the execution clusters are processed in parallel.

At the end of Algorithm 4.1 we have in the variable *clusterpaths* the list that contains all the paths of the cluster. To find the minimum behavior three parameters are passed:

- **Clusterpaths**: all the paths generated by the process exposed above
- **Clusternumber**: the number of the cluster we are currently processing

[1] https://www.python.org/downloads/release/python-279/

Algorithm 4.1 Path Calculation

```
 1: procedure ProcessCluster(cluster, clusternumber)
 2:     clusterpaths ← list
 3:     graphs ← cluster[graphs]
 4:     for all g ∈ graphs do
 5:         singlegraphpaths ← list
 6:         if allpaths then
 7:             for all node1 ∈ nodes(g) do
 8:                 for all node2 ∈ nodes(g) do
 9:                     graphpaths ← getAllPaths(node1, node2)
10:                     for all path ∈ graphpaths do
11:                         singlegraphpaths ← singlegraphpaths + path
12:                     end for
13:                 end for
14:             end for
15:         else
16:             rootnodes ← getRootNodes(g)
17:             leaves ← getLeaves(g)
18:             for all rootnode ∈ rootnodes do
19:                 for all leaf ∈ leaves do
20:                     graphpaths ← allPaths(rootnode, leaf)
21:                     for all path ∈ graphpaths do
22:                         singlegraphpaths ← singlegraphpaths + path
23:                     end for
24:                 end for
25:             end for
26:         end if
27:         clusterpaths ← clusterpaths + singlegraphpaths
28:     end for
29:     findMinimumBehavior(clusterpaths, clusternumber, graphs)
30: end procedure
```

- **Graphs**: the graphs of the cluster, in order. This is required for parameter modeling process in section 4.2.5

In addiction to the facts exposed above, cluster paths and graphs are ordered: as they are both memorized in a list, the first element of graphs contains the graphs associated to the paths of the first element of cluster paths. Is also easy to notice that the complexity of the algorithm to generate paths is very high: a single path can be found in $O(e+n)$ complexity where e are the edges and n are the nodes but number of paths in a graph can be very large: in case of a complete graph of n order, complexity become $O(n!)$. [21]

Due to the fact we have to compute all the possible paths between nodes and roots, we need to establish some requirements in the graph representing behaviors we analyze and see if they are satisfied

- **Aciclicity:** graphs should present no cycles. Due to this fact, one or more root node can be found, and one or more leaf node can be found.
- **Sparse:** our graphs are not completely connected. They have only a few edges.
- **Node are few:** if nodes are many, computational power for executing the algorithm will be very high

The first requirement is always true: graph are always acyclic because they represent a flux of API calls obtained by a dynamic analysis: we have a temporal sequence of the API calls executed. The truth of the second and third requirements is verified experimentally, as we are going to see in Chapter 5 a large part of graphs meets those hypotheses, but there is still a little part of big graphs we cannot process.

4.2 Common Behavior Search

The search for the common behavior between all the cluster graphs starts from the assumption, already stated in the previous chapter, that the behavior we want to extract will be a graph smaller in terms of API calls than each one of the graph we actually have in the cluster. For this reason, after have generating all the paths between root and leaves in each graph of the cluster, our minimization process is done starting from the first graph and its relative generated path list. Algorithm 4.2 takes in input this list in variable *clusterpaths* and the cluster graphs in variable *graphs.*

Algorithm 4.2 essentially is the external shell of our algorithm. From this procedure are launched the various sub-algorithms that we are going to see in the next sub-sections to process the cluster producing the common behavior. Those sub-algorithms are explained as follows: after a path initialization in paragraph 4.2.1, the last element of the *clusterpath* list is taken as model for our calculation: from now

Algorithm 4.2 Minimum Behavior Search

```
1: procedure findMinimumBehavior(clusterpaths, clusternumber, graphs)
2:     initializePaths(clusterpaths, clusternumber, graphs)
3:     modelgraph ← clusterpaths.pop
4:     finalgraph ← list
5:     finalparams ← list
6:     for i ← 1 to len(modelgraph) do
7:         path ← modelgraph[i]
8:         if isPathPresentInOtherGrpahs(path, clusterpaths) then
9:             pathToMatch ← list(path)
10:            allMatchedPaths ← getAllMatchedPaths(pathToMatch, clusterpaths)
11:            markAsReadNodesInLists(pathToMatch, clusterpaths)
12:            markAsReadNodes(pathToMatch, modelgraph)
13:            pathparameters ← getParameters(path, allMatchedPaths, graphs)
14:            finalgraph ← finalgraph + path
15:            finalparams ← finalparams + pathparameters
16:        end if
17:    end for
18:    exportInGraph(finalgraph, finalparams, clusternumber)
19: end procedure
```

that graph will be referred as «model graph» and the other cluster graphs as «other graphs». In the act of the final behavioral model creation algorithm instantiates two lists (*finalgraph* and *finalparams)* that will contains the paths of the final graphs and their occurrences in the cluster graphs in the act of parameters models generation, as explained in section 3.4.

After this initialization phase, algorithm goes into its core. In sub-section 4.2.2 we are going to see how for is searched a match in other graphs for each path in *model graph*. If the search succeeds, in sub-section 4.2.3 we are going to see how matching phase is done and what post-processing procedure are done on all the paths that have matched like node flagging.

In this part it is important to set the read flag firts in the nodes in other graphs than in our model graph. This is because during the node marking phase we are going to repeat the match and if nodes in model graph are flagged before node in other graphs, matching fails. When node marking has ended, parameters for the path are calculated with *getParameters* procedure as explained in sub-section 4.2.5.

At the end we have to generate the output: lists *finalgraph* and *finalparams* are

populated. From their values we can build the behavioral model: *exportInGraph* procedure is called for the final export.

4.2.1 Node initialization

Paths between all roots and leaves are provided by *networkx* API calls[2]: calling the particular API call used in this code snippet we obtain as output all the possible paths between the two nodes in input. The initialization process shown in Algorithm 4.3 takes this node list and adapts it into a suitable format.

Algorithm 4.3 Node initialization

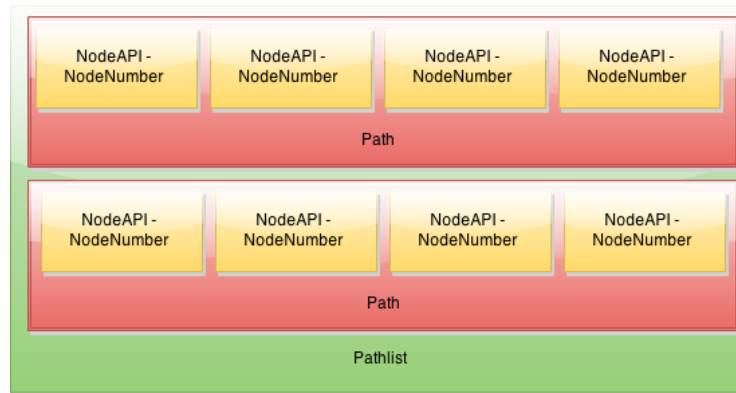
```
1: procedure INITIALIZEPATHS(clusterpaths, clusternumber, graphs)
2:     for k ← 1 to LEN(clusterpaths) do
3:         for j ← 1 to LEN(clusterpaths[k]) do
4:             for i ← 1 to LEN(clusterpaths[k][j]) do
5:                 newnode ← list()
6:                 newnode ← newnode + clusterpaths[k][j][i]
7:                 newnode ← newnode + False
8:                 clusterpaths[k][j][i] ← newnode
9:             end for
10:            newgraphpaths ← list()
11:            newgraphpaths ← newgraphpaths + clusterpaths[k][j]
12:            newgraphpaths ← newgraphpaths + COMPUTEPOINTS(graphs[k],
   clusterpaths[k][j])
13:            clusterpaths[k][j] ← newgraphpaths
14:        end for
15:    end for
16: end procedure
```

This process transforms the data structure from a simple list of paths in a list of paths that is associated to an integer, which represents the amount of points of the paths calculated by the heuristic explained in section 3.3.2. Algorithm used in our code is exposed in Algorithm 4.4. The neighbors function exposed in the algorithm is managed by *networkx* API calls[3] and returns the number of neighbors of the node given.

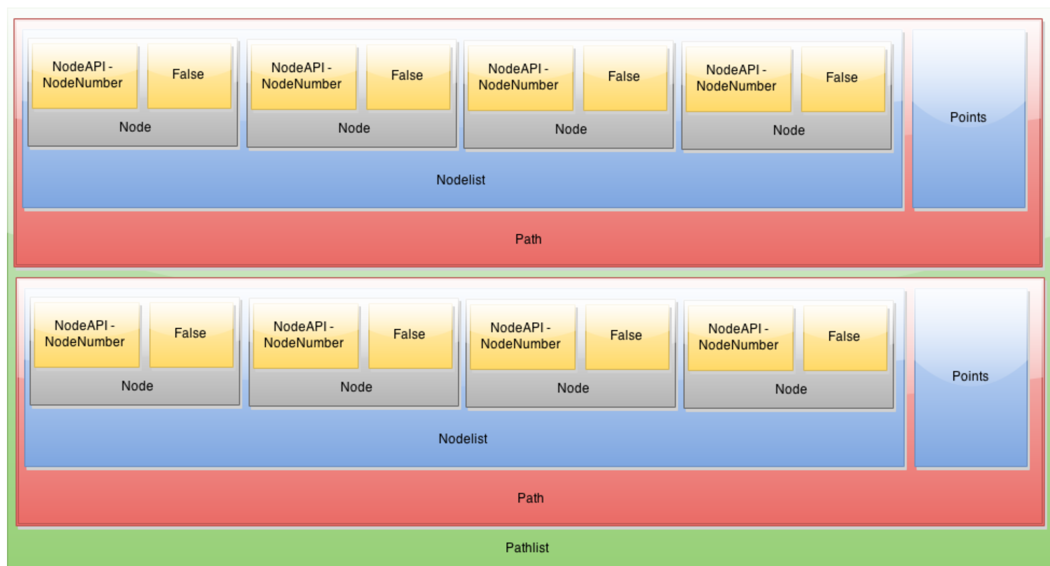
A «read» flag is added to each node. The transformation is displayed in Figure

[2]https://networkx.github.io/documentation/development/reference/generated/networkx.algorithms.simple_paths.all_simple_paths.html

[3]https://networkx.github.io/documentation/latest/reference/generated/networkx.Graph.neighbors.html



(a) Data structure Before Initialization



(b) Data structure After Initialization

Figure 4.1: Data structure initialization

4.1. Read flag is automatically set to *False* at initialization. The correct structure is automatically returned in the *clusterpaths* variable passed as input.

4.2.2 Path searching

The most relevant function of our program, is, without doubts, the search for paths in other graphs: it is shown in Algorithm 4.5. The procedure *cleanpath* called in this way initializes the nodes for matching procedures creating a new list with node names and relative reading flags that is processed in the further parts of the

Algorithm 4.4 Points Computation

```
1: procedure COMPUTEPOINTS(graph, path)
2:     points ← 0
3:     deep ← 0
4:     for all node ∈ path do
5:         deep ← deep + 1
6:         neighbors ← GRAPH.NEIGHBORS(node[name])
7:         points ← points + LEN(neighbors) ∗ deep ∗10+ deep
8:     end for
9:     return points
10: end procedure
```

algorithm: this is done because we want to match nodes not by their names, but by their API call functions.

Algorithm 4.5 Searching for path in other graphs

```
1: procedure ISPATHPRESENTINOTHERGRAPHS(pathtomatch, clusterpaths)
2:     cleanpathtomatch ← LIST
3:     for all node ∈ pathtomatch[0] do
4:         cleanpathtomatch ← cleanpathtomatch + CLEANPATH(node)
5:     end for
6:     for all graphpaths ∈ clusterpaths do
7:         hasGraphPath ← False
8:         if !ISPATHPRESENTINGRAPH(cleanpathtomatch, graphpaths) then
9:             return False
10:        end if
11:    end for
12:    return True
13: end procedure
```

Before execution of Algorithm 4.6 we have in variable *cleanpathtomatch* the API call list of the path, with their read flags. This is passed as input to procedure *isPathPresentInGraph* with the graph to search for the path in. If the search of the path in one graph fails, the whole procedure halts immediately returning false. Procedure *isPathpPresentInGraph* in Algorithm 4.6

Algorithm 4.6 cleans again every path and checks if there is a match in the specified graph for all the generated paths. At the end, the algorithm returns whether the path was found or not. Notice that in those two steps we found out if there is a match, not what is the best match. If the procedure succeeds the *getAllMatchedPaths* procedure is called. This procedure does only the matching: it does not modify read

Algorithm 4.6 Path searching in a single graph

```
 1: procedure isPathPresentInGraph(cleanpathtomatch, graphpathsrpaths)
 2:     hasGraphPath ← False
 3:     for all path ∈ graphpaths do
 4:         for all node ∈ path[0] do
 5:             cleanpath ← cleanpath + cleanPath(node)
 6:             if cleanpathtomatch == cleanpath then
 7:                 hasGraphPath ← True
 8:             end if
 9:         end for
10:     end for
11:     return hasgraphpath
12: end procedure
```

flag values for each node. Matches between two paths obviously succeed if path is composed by API calls of the same type, in the same order and with same values of the read flag. If match in a graph succeeds, function immediately terminates returning True as output.

4.2.3 Path Matching

As stated before, these procedures are called only if there is a match. Our purpose now is not just to get the match, but also to get the best match according to the heuristic we created. Read state of the node does not have to be checked in this part because we know that a match exists, so what we simply find the best match calling for every graph the *matchPath* procedure. The *matchPath* procedure is shown in algorithm 4.7

Match succeeds when conditions in section 4.2.3 are met. For every graph the set of paths that have matched are returned as a list into *pathresults* variable.

In the second part evaluation of optimal path between this list is done using points system we explained in section 3.3.2. Final result is returned as output.

4.2.4 Node flagging

After a successful matching phase for a path nodes of this path have to be marked as read in every graph of the cluster to avoid redundant results for the next algorithm iterations. The two procedures for this purpose are shown in Algorithm 4.1. Like

Algorithm 4.7 Path Matching in a Graph

```
1: procedure matchPath(pathToMatch, graphpaths)
2:     pathResults ← List
3:     for all node ∈ pathtomatch[0] do
4:         cleanpathtomatch ← cleanpathtomatch + cleanPath(node)
5:     end for
6:     for all path ∈ graphpaths do
7:         for all node ∈ path[0] do
8:             cleanpath ← cleanpath + cleanPath(node)
9:             if cleanpathtomatch == cleanpath then
10:                pathResults ← pathResults + path
11:            end if
12:        end for
13:    end for
14:    for all path ∈ pathresults do
15:        finalpath ← List
16:        difference ← MaxInt
17:        actualdifference ← Abs(path[1] - pathtomatch[1])
18:        if actualdifference < difference then
19:            finalpath ← path
20:            difference ← actualdifference
21:        end if
22:    end for
23:    return finalpath
24: end procedure
```

algorithms for path searching even in this case there are two Algorithms: the first to search in all path lists and the second to effectively mark nodes as read in the lists.

Procedure set the read flag of a node as shown in Algorithm 4.8. In the operation of node flagging, matching between paths is still used: this time its purpose is not to do node matching but simply to check that the node to mark as read in *graphpaths* is exactly the one that matched before.

4.2.5 Parameters model creation

The only aspect that remains out is the parameters model creation. This is done by the *getParameters* procedure which purpose is to get the model parameters for each API node, the procedure is shown in Algorithm 4.9

This is one of the most complex algorithms of our code: procedure is explained in abstract in section 3.4. It takes as input the path of the model graph, all the

Algorithm 4.8 Node flagging in a path

```
1: procedure MARKASREADNODES(pathToMark, graphpaths)
2:     pathToMark ← pathToMark[path]        ▷ Choose the nodelist, discard points
3:     for i ← 1 to LEN(pathToMark) do
4:         for all path ∈ graphpaths do
5:             nodeList ← path[path]
6:             node ← nodeList
7:             if CLEANPATH(pathToMark[i]) == CLEANPATH(node) then
8:                 nodeindex ← nodeList.INDEX(node)
9:                 nodelist ← nodeList.REMOVE(node)
10:                node ← SETREAD(node, True)
11:                nodelist ← nodeList.INSERT(node, nodeindex)
12:            end if
13:        end for
14:    end for
15: end procedure
```

paths that matched with it and the cluster graphs: API calls of the paths, grouped by type, are searched in the graphs object to retrieve their parameters. For each attribute, immediately at the end of parameters retrieval phase model is created calling *getAttributeModel* procedure. At the end of this procedure, model for each API call of the final graph is stored in *finalattributelist* variable that is returned on the main procedure.

4.3 Graph Export

Last part of our algorithm concerns the export of our list of common paths and their parameters into a graph: this is done by Algorithm 4.10. The Algorithm is logically divided in three parts. In the first one, the graph structure is created adding all the nodes of every common paths: if a node is in two or more paths, during the addition phase will be automatically discarded. Edges are then created. As nodes should have different names by design, a number is added to distinguish different nodes characterized by same API call.

In the second part of the algorithm, models of the parameters generated with algorithm 4.9 are inserted in the final model. In the third one the created behavior

is exported in both dot[4] (for readability) and gpickle[5] format (for re-usability) are done.

[4] http://www.graphviz.org/
[5] https://docs.python.org/2/library/pickle.html

Algorithm 4.9 Parameters model creation procedure

```
1:  procedure getParameters(originalpath, allMatchedPaths, graphs)
2:      allMatchedPaths ← allMatchedPaths + originalpath
3:      finalAttributeList ← List
4:      for i ← 1 to len(originalpath[0]) do
5:          g ← graphs[0]
6:          path ← allMatchedPaths[0][0]
7:          node ← path[i]
8:          globalnodeAPI ← node[0]
9:          attributelist ← attributelist + g.node[globalnodeAPI]
10:         nodeAttributeValues ← List
11:         for all attribute ∈ attributelist do
12:             attributeValues ← List
13:             for k ← 1 to len(allMatchedPaths) do
14:                 g ← graphs[k]
15:                 path ← allMatchedPaths[k][0]
16:                 node ← newpath[i]
17:                 nodeAPI ← node[0]
18:                 attributevalues ← attributevalues + g.node[nodeAPI][attribute]
19:             end for
20:             finalattrib ← getAttributeModel(attribute, attributevalues)
21:             if finalattribute then
22:                 attributevalues ← list()
23:                 attributevalues ← attributevalues + attribute
24:                 attributevalues ← attributevalues + finalattrib
25:                 nodeattributevalues← attributevalues + nodeattributevalues
26:             end if
27:         end for
28:         newnode ← list()
29:         newnode ← newnode + nodeAPI
30:         newnode ← newnode + nodeattributevalues
31:         finalattributelist ← finalattributelist + newnode
32:     end for
33:     return finalattributelist
34: end procedure
```

Algorithm 4.10 Graph Export

```
 1: procedure exportInGraph(modelgraph, finalparams, clusternumber)
 2:     exportgraph ← nx.DiGraph
 3:     for all path ∈ modelgraph do
 4:         path ← path[0]
 5:         previousnode ← null
 6:         for all node ∈ path do
 7:             nodename ← getNode(node)
 8:             exportgraph.AddNode(nodename)
 9:             if previousnode then
10:                 exportgraph.AddEdge(previousnode, nodename)
11:             end if
12:             previousnode ← nodename
13:         end for
14:     end for
15:     for all params ∈ finalparams do
16:         for all node ∈ params do
17:             nodename ← getNode(node)
18:             for all attribute ∈ node[1] do
19:                 nodeatt ← attribute[0]
20:                 valueatt ← attribute[1]
21:                 exportgraph.node[nodename][nodeatt] ← valueatt
22:             end for
23:         end for
24:     end for
25:     nx.WriteDot(exportgraph)
26:     nx.WriteGpickle(exportgraph)
27: end procedure
```

Chapter 5

Experimental results

In this chapter we report the experimental result we obtained and we explore reasons for those results and limitations of our work. In section 5.1 we define our dataset, show the results of our behavior clustering and see how they are affected by our experimental hypothesis exposed in chapter 3. In section 5.2 we are going to see how our model are composed in details and see differences and comparison with our old behavioral system. In section 5.3 we are going to see concretely how our approach generates more distinction in models than Jackdaw one and how this model are different between themselves. At the end, in section 5.4 we are going to analyze the performance of our experiment.

5.1 Dataset

The dataset used to validate this work is composed by 3112 random malware samples downloaded from VirusTotal Intelligence service[1]. They are portable exe format samples for an x86 Windows Architecture.

We performed static and dynamic analysis on each sample of our dataset using Jackdaw. Dynamic analysis is performed by Cuckoo with an execution timeout of 10 minutes. We choose samples that produce a valid analysis, provide API calls executed in the malware sample and their relative fingerprints. Fingerprints can be present in part of the API set tracked in the sample or in the whole set. We then

[1] https://www.virustotal.com/intelligence/

run clustering as described in [20] .

To meet the hypothesis exposed in paragraph 3.3 part of those behaviors are discarded in clustering phase, in particular:

- Behaviors with only one node
- Behavior with 100 or more nodes
- Behavior with no fingerprints

In Table 5.1 there are the results of the clustering analysis. 1269688 behaviors are generated from 3112 samples, and after the pre-filtering phase only 5122 are accepted. Some behaviors can present no fingerprint due to the fact that not all malware samples has a properly execution. Some samples returns an empty or corrupt analysis for the following reasons:

- **Cuckoo is a sandbox and sandboxes can be detected.** This makes for us impossible to track or terminate many malware behaviors causing the failure of the process or an API report with no or few API.
- **Some malware samples requires human interaction.** As dynamic analysis hooks only API effectively called, if malware requires an interaction to be executed - for example an installer - those actions won't be tracked by an automatic Sandbox
- **Some malware samples requires other programs.** Some malware infected the system using other software, for example opening RTF document or a PDF document or images: even if executable are chosen, we cannot exclude that those executable didn't simply download other type of malware and execute them. With this approach loading of those samples is tracked, but nothing else.
- **Some malware samples are packed**. As seen in previous chapters malware packed is a problem because brings static analysis to generate fingerprints not on the real sample but on the encrypted / compressed one. This causes less

fingerprints in analysis, bringing wrong clusterization and leading the process to an empty results.

We also see, from this analysis, that our hypothesis of graphs that are sparse and with node low in numbers are verified. If we take into accounts all the behaviors - from every sample 6,53 behaviors are generated in average and only 0,63 of them are big, meaning a rate of 9,6%.

From those 5122 accepted behaviors, 607 clusters are generated.

At this point of the analysis we try to find how many clusters are generated according to samples analyzed. We tracked the number of cluster generated every 100 samples processed and we found that the progression is instead linear: results of this analysis are shown in Figure 5.1 where on x-axis is represented the number of sample processed and on y-axis the number of the clusters generated at that point of execution

We can see that cluster progression is linear in the order of numbers of samples analyzed.

5.1.1 Cluster distributions

At this point of the experimental results, behavior are divided into cluster. Not all the clusters are suitable as input for our algorithm: some clusters are composed by only one behavior so common behavior is already modeled, and some clusters presents one or more graphs that does not meet hypothesis exposed in paragraph 4.1 so their execution could not terminate in acceptable time. According to those

Table 5.1: Behaviors Analysis Results

Result Type	**#Behaviors**
One node	1249339
100 or more nodes	1969
No Fingerprints	13258
Accepted	5122

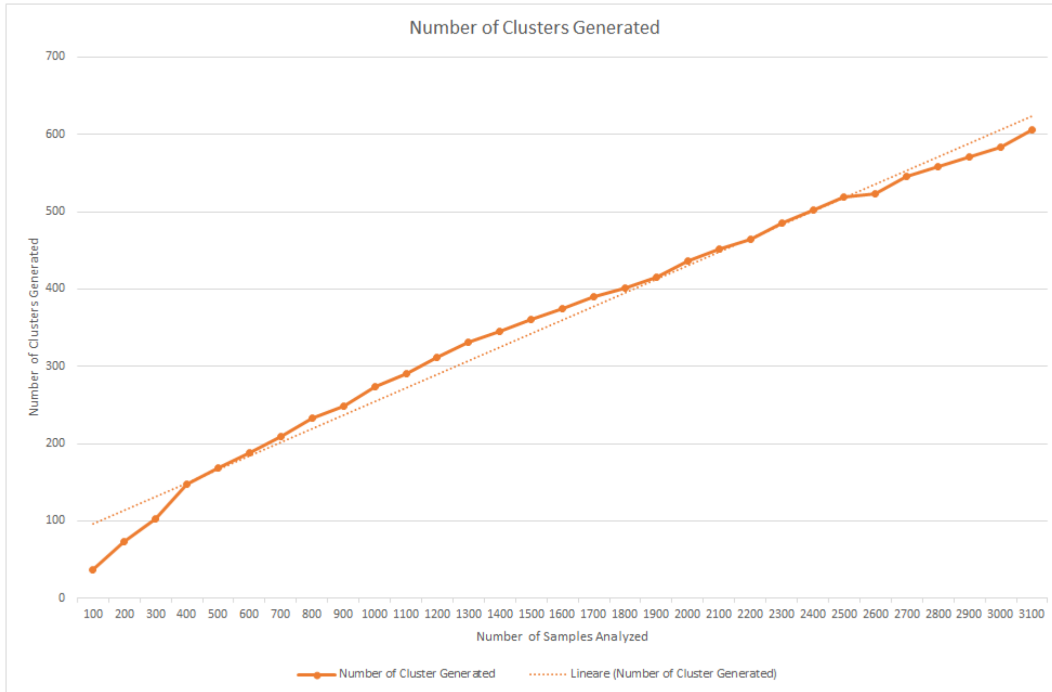


Figure 5.1: Cluster generation progression

hypothesis, we choose to distinguish four different types of outcomes for each cluster regarding the analysis termination:

- **Not Valid:** in those clusters our hypothesis of little behaviors expressed in section 3.3 is not valid due to big graphs generated by taint analysis process. Computation won't terminate in appreciable time and execution of those clusters must be interrupted. Behavior with more than 100 API calls are discarded during the clusterization phase, but this act only as a filter and does not exclude the fact that some cluster cannot terminates due to big graphs.

- **Singleton:** cluster contains only one graph. In this case algorithm is valid

Table 5.2: Behavior results having more than one API sample

Result Type	#Behaviors	Average Behaviors per sample
100 or mode nodes	1969	0,63
No Fingerprints	13258	4,26
Fingerprints	5122	1,64
Total	20349	6,53

and the behavioral model is simply the model of the only graph presented in the cluster.

- **Fail:** those clusters are composed by behaviors API calls that are totally different graph by graph, so API calls set intersection of the cluster is empty. To understand why this can happen: fingerprints are generated from assembly code and sequences of different API calls can express the same behavior and generated from same or similar assembly code.
- **Synthesized**: those clusters has more than one graph in it and the algorithm works correctly producing the common cluster behavior we expected

The result of the distribution of the 607 cluster produced from the analysis are represented in the Table 5.4

Our algorithm fails only 14% of the times, and succeeds in 75% of the clusters[2]. In those 75% a valid result is produced, modeling correctly the common behavior of the cluster. We can't say anything on the rest 10% because is composed by behaviors that doesn't meet our hypothesis. In figure 5.2 the numerical distribution of graph per cluster is shown considering the 457 clusters that have produced a valid result: most of the clusters have two, three or four behavior to synthesize: distribution is clearly hyperbolic. Obviously most of the clusters are alone, because we choose malware randomly: using this approach with specific malware family samples produces a different distribution.

In Figure 5.3 we can see the distribution of the nodes (i.e. API) in the final

[2] If clusters contains only one behavior, theoretically algorithm succeed

Table 5.3: Clusters data

Data Type	Data Value
$\frac{\#clsusters}{\#samples}$	$0,20$
$\frac{\#graphs}{\#clusters}$	$8,43$

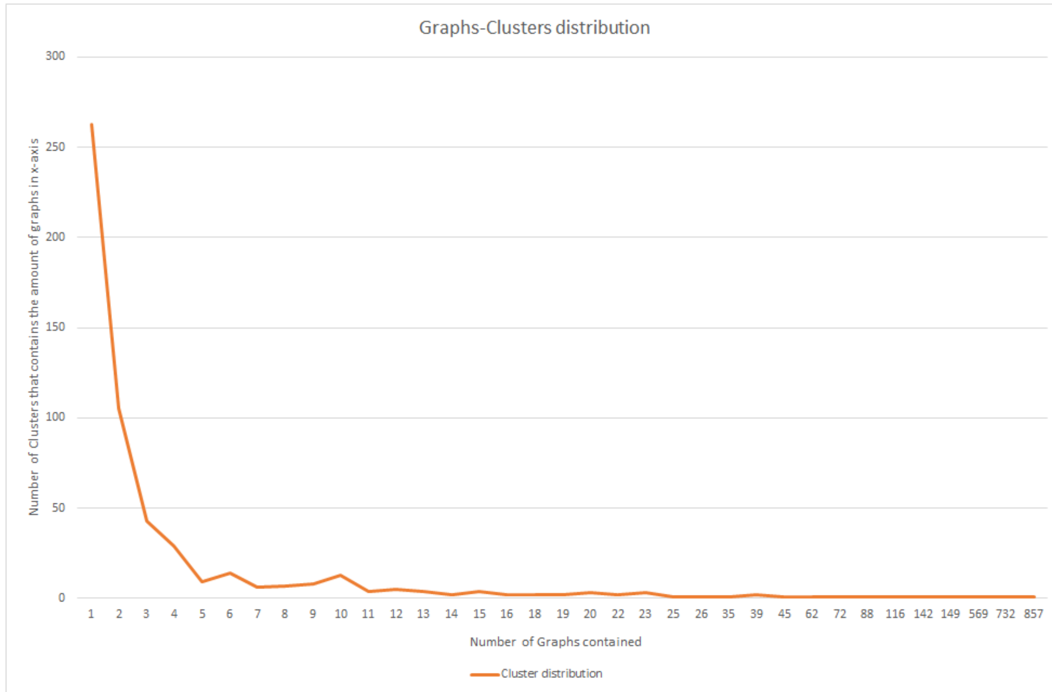


Figure 5.2: Graphs distribution

behavioral models: the majority of the graphs has just two nodes as model: only 50 models have 5 or more nodes in final graph. This means that our hypothesis of small graphs is true.

Confronting for every cluster the number of nodes before and after the algorithm application we can say that approximately 16% of nodes for each graph in cluster are deleted by the modeling algorithm.

In most of cases (62%) the number of the nodes in final result is the average of the cluster graph nodes: this means that in those type of cluster behaviors were already very similar. In the other clusters, instead, modeling algorithm produces a result that is radically different by shape and dimension from each one of the cluster behaviors, but shares a common core that is detected by the algorithm.

Table 5.4: Cluster distribution

Cluster Type	#Cluster Produced	Percentage on total
Singleton	263	43,33%
Fail	88	14,50%
Not Valid	62	10,21%
Synthesized	194	31,96%

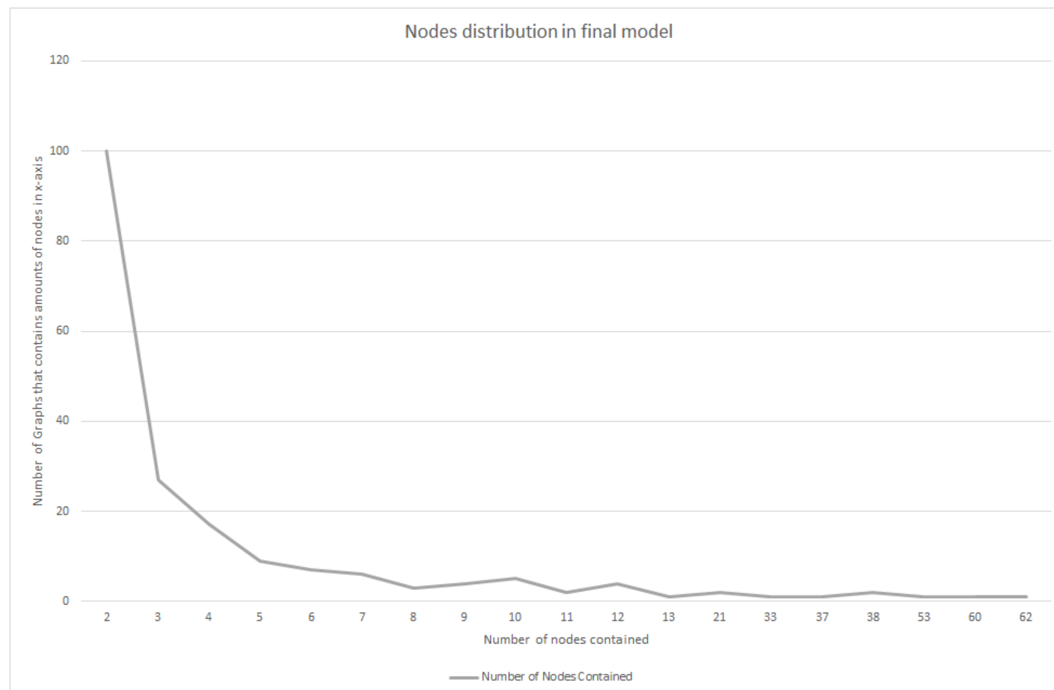


Figure 5.3: Nodes distribution

5.2 Model Analysis

At the end of process, behavioral models have been produced for each valid cluster. Here we present two of these models to understand differences and similarity between our models and the ones in Jackdaw. In sub-section 5.2.1 we present a simpler sample, with no API calls repetition, to understand how our behavioral model works. In sub-section 5.2.2 instead we present a more complex model, and make a comparison between them to better understand the differences between new and old behavioral models.

5.2.1 Example 1

In listing 5.1 there a human-readable, dot code of one of synthesized models, while its graph is represented in Figure 5.4. Dot code representation is used by *Graphviz*.

We can see that this sample creates one file and writes something on it.

This model is generated by 2 graphs in cluster: we clearly see 2 API calls and one dependency between them. Looking at API call parameters, we can see that

parameters in behavioral model are different only in buffer and in filename due to the fact that there are different elements in those fields and only one in the other. About the parameter filename, is clear is that file is stored in *C:\Windows\Rescache* folder. In the first sample is in *wip\ResCache.hit* folder, in the second sample is called *ResCache.mni*. Those are the values that will be modeled using Jackdaw models.

About the other information, *createdisposition* field has this parameter set to 5 for all the behaviors modeled and *shareaccess* is set also to 3. This couple of choices means that a new file with that name of 0 bit size is created and this file cannot be deleted while is in an open state (example, currently in use by another process).

Listing 5.1: Sample 1 dot Code

```
1   strict digraph G {
2       "NtCreateFile - 80843" [
3           category="[u'filesystem']",
4           ShareAccess="[u'3']",
5           repeated="[0]",
6           api="[u'NtCreateFile']",
7           CreateDisposition="[u'5']",
8           FileName="[
9               u'C:\Windows\rescache\wip\ResCache.hit',
10              u'C:\Windows\rescache\ResCache.mni']"
11          ];
12      "NtWriteFile - 80844" [
13          category="[u'filesystem']",
14          Buffer="[
15              u'RESMANI\x00\x00\x00\x00\x00\x00\x00\x00\x00\x88\x13\x00\x00\x00\x00\x02\x00\x00\x00\
                    x00\x002\x00\x00\x00\x01\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
                    xff',
16              u'RESCHIT\x00\x01\x00\x00\x00\x0c\x02\x00\x00\x10E\xec`\"\\xd0\x01\x00\x00\x00\x00\x00\
                    x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
17          ]",
18      repeated="[0]",
19      api="[u'NtWriteFile']"
20      ];
21      "NtCreateFile - 80843" -> "NtWriteFile - 80844";
22  }
```

5.2.2 Example 2

The second sample is in listing 5.2 that corresponds to the graph in Figure 5.5. We are going to see the step immediately after the graph exposed in previous paragraph:

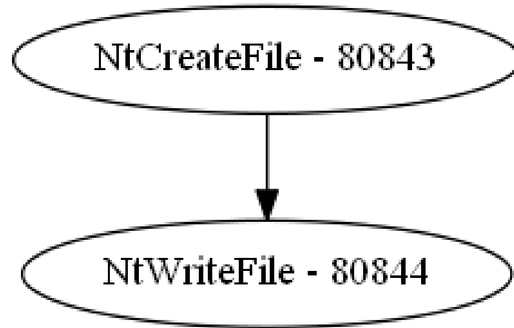


Figure 5.4: Sample 1 Graph

two different writes on the same file. Also in this file, the behavioral results is generated from two clusters.

We can see that the file created, this time, is *C:\Windows\rescache\wip\Segment[number].cmf* where [number] can be 0 or 1. Then on the same file two different writes are performed. The text of the two writes API calls is in the parameters of the buffers.

Listing 5.2: Sample 2 dot Code

```
1  strict digraph G {
2      "NtWriteFile - 80852" [
3          category="[u'filesystem']",
4          Buffer="[
5              u\"RESCSEG\x00\x01\x00\x00\x00\x01\x00\x00\x005\x00\x00\x005\x00\x00\x00\x14\x02\x00\
                   x00I\xa4\x1f\x00\xa9\xb4\x1f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
                   x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff
                   \xff\xff\xff\xff\xff\xff\xff\x01\x00\x00\x00\x00\x00\x01\x00 \x00\x00\x00\xb1\x15\
                   x00\x00\x00\x00\x00\x00\xe91\x00\x00sb\x00\x00\x8a\x96\x00\x00\x9b\xb6\x00\x00\x0e\
                   xe6\x00\x00\xed\x1e\x01\x00\x86_\x01\x00\x12\x8a\x01\x00\xa0\xc0\x01\x00o\xf3\x01\
                   x00\x1e#\x02\x00bS\x02\x00\x13}\x02\x00O\xac\x02\x006\xe9\x02\x00\xb6)\x03\x00\x10
                   `\x03\x00\xf0\x9d\x03\x00\xf1\xd1\x03\x009\x04\x04\x00mC\x04\x00\xb0\x85\x04\x00M\
                   xc8\x04\x00M\x05\x05\x00\x18T\x05\x00\x06\x9f\x05\x00n\xe6\x05\x00'.\x06\x00\x821\
                   x06\x00o\xcf\x06\x00t5\x07\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
                   x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                   \x00\x00\",
6              u\"RESCSEG\x00\x01\x00\x00\x00\x00\x00\x00\x00^\x00\x00\x00^\x00\x00\x00\xe7\x03\x00\x00
                   \xa1\x18?\x00\x01)?\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
                   \x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
                   xff\xff\xff\xff\xff\x01\x00\x00\x00\x00\x00\x01\x00@\x00\x00\x00y\x08\x00\x00\x00\
                   x00\x00\x00^C\x00\x00j\x9a\x00\x00\xe7\xd4\x00\x00v\x19\x01\x00\xeeR\x01\x00\x1d\
                   x96\x01\x00\xe9\xce\x01\x004\xf3\x01\x00\xefD\x02\x00q\x84\x02\x00\x86\xc2\x02\x00\
                   xc8\xf6\x02\x00\xf1.\x03\x00)m\x03\x00\xc4\xa4\x03\x00\xcc\xea\x03\x00\xdb'\x04\x00
                   \xa5i\x04\x00r\x9d\x04\x00w\xe9\x04\x00\xa1,\x05\x00c~\x05\x00\xec\xbc\x05\x00\x15\
                   xfe\x05\x00<D\x06\x00\xbb\x8b\x06\x00y\xc7\x06\x00\xe6\x07\x07\x00\xe0J\x07\x00Nx\
                   x07\x00\xb9\xc5\x07\x00T\xf9\x07\x00\xf43\x08\x00\xe3u\x08\x00\xa2\xb4\x08\x00\xe3\
                   xef\x08\x002H  \x00\xd0\xa0    \x00\xd0\xa0\n\x00\xd0\xa0\x0b\x00\"
7          ]",
8          repeated="[0]",
```

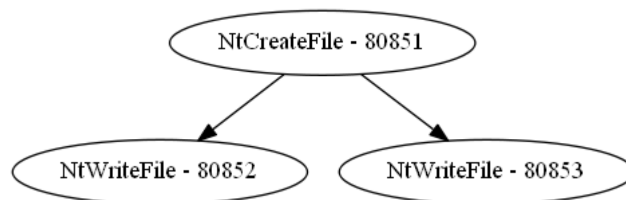


Figure 5.5: Sample 2 Graph

```
9            api="[u'NtWriteFile']"
10       ];
11       "NtCreateFile - 80851" [
12           category="[u'filesystem']",
13           ShareAccess="[u'0']",
14           repeated="[0]",
15           api="[u'NtCreateFile']",
16           CreateDisposition="[u'5']",
17           FileName="[
18               u'C:\Windows\rescache\wip\Segment1.cmf',
19               u'C:\Windows\rescache\wip\Segment0.cmf'
20           ]"
21       ];
22       "NtWriteFile - 80853" [
23           category="[u'filesystem']",
24           Buffer="[
25               u'\x00\xb8A\x00MZ\x90\x00\x03\x00\x00\x00\x04\x18\x00\xwritten ff\xff\x00\x00\xb88\x00\
                     x01\x00@<\x00\x07\x00O\x03Y\x01\x0e\x1f\xba\x0e\x00\xb4  \xcd!\xb8\x01\x00\x00\x00\
                     x00L\xcd!This program cannot be run in\x05\x01\x02\x00 DOS mode.\r\r\n$\x1c\x02\
                     x01u\xf5\xd9E\x14\x9b\x8a\x1d\x00Ll\x0f\x8aD:\x00\n:\x00  \x10\x00\x0eRich\xb9\x00t
                     \x01\x06\x00PE\x00\x00L\x01\x01\x00\xd5\xc6[J\x94\x00\x00\xe0\x00\x02!\x0b\x01 c\
                     x00\xf4\x02\xac\x00\xe6/^\xbc\x00\x00\x10\x1c\x00)\x009\x00\xd0\x00\x06\x00\x01\x1a
                     \x00\x05\xb2\x00\xc9\x00(\x08\xb8\x00\xb3\x9e\x030\x00@\xb0\x00@\x01\x9a\x01=\x006\
                     x00\x07\x00\xa0\x00\x90\xf3_\x03\xf1\x07\x00F.^\x0b\xfa\x0frsrcJ\x00x\x05\x81\x049\
                     x07\xf7\x03\xf5\xd8\r\xf7\r\x0b\x07\x00\xd3\xf9\x0f\x01@\x02\xe8\x00\x00\x80',
26               u'\x00\xb8A\x00MZ\x90\x00\x03\x00\x00\x00\x04\x18\x00\xff\xff\x00\x00\xb88\x00\x01\x00@
                     <\x00\x07\x00O\x03Y\x01\x0e\x1f\xba\x0e\x00\xb4  \xcd!\xb8\x01\x00\x00\x00\x00L\xcd
                     !This program cannot be run in\x05\x01\x02\x00 DOS mode.\r\r\n$\x1c\x02\x01u\xf5\
                     xd9E\x14\x9b\x8a\x1d\x00Ll\x0f\x8aD:\x00\n:\x00\x0b\x10\x00\x0eRich\xb9\x00t\x01\
                     x06\x00PE\x00\x00L\x01\x01\x00\x0c\xbf[J\x94\x00\x00\xe0\x00\x02!\x0b\x01   c\x00\
                     x06\xa4\x00\x01\x00`\x7fiu\x10\x1c\x00)\x009\x00\x02\xf1\x00\x01\x1a\x00\x05\xb2\
                     x00\x00\x00 \xbb\x00N\x1d1\x00@\xb0\x00@\x01\x9a\x01=\x006\x00\x07\x00\xa0\x00\xf8\
                     xe1   \x07\x00/N.rsrc\xae_\xa1\xffA\x00\x9d\x069\x07\xb8\x05\x07\x00\xd8\r\xf7\r?\
                     x0b\x07\x00\x11\x06\x01\x08\x00\xa0\x00\x00\x80P\x08\x80\xd1\x028'
27           ]",
28           repeated="[0]",
29           api="[u'NtWriteFile']"
30       ];
31       "NtCreateFile - 80851" -> "NtWriteFile - 80852";
32       "NtCreateFile - 80851" -> "NtWriteFile - 80853";
33   }
```

5.2.3 Comparison with Jackdaw

The previous 2 behavioral models we described are not distinguishable with the old modeling system in Jackdaw. In fact, using the old modeling system, both samples are modeled using this formula

$$B = NtCreateFile \wedge NtWriteFile$$

Using graph modeling with taint dependencies, we are able to distinguish the fact that in the second example two distinct operations *WriteFile* are present. In fact is obviously applicable the following rule: *all the behavioral model with the same API set and different number, of API or different API dependencies, are distinguishable only with the graph behavioral model.*

We have 545 clusters that produce behavioral models, but only 69 sets of API calls in those models - meaning that with Jackdaw old modeling system we can distinguish at most 69 different behaviors. Obviously, not all 545 models we produced are different: some of them can be equal, so also our dataset can be shrunk. In the next paragraph, we will see how we can distinguish our models and what are and how are differences between them.

5.3 Behavior matching comparison

Starting from the same cluster, the API call set presented in the old modeling system may be different from the API calls set presented in new modeling system. For Jackdaw design choices, in fact, for being present in the model an API call must present one or more fingerprints and has to be present in at least 70% of the clusters behaviors. In the new behavioral model, the clustering algorithm uses the whole graph as we have taint dependencies: if an API call has no fingerprints but has dependencies (of every grade) from other API calls that have fingerprints, this API call is still present in the behavioral model. On the other hand new model considers only API calls that are present in 100% of cluster graphs. Table 5.5 recaps what we have just explained.

5.3.1 To compare the old and the new system

We define «group» *a set of clusters whose behaviors are not distinguishable using the old modeling system.* To evaluate if we can distinguish more behavior with new modeling system, we divide all the clusters into groups.

286 clusters out of 607 belong to a group. 242 using the old modeling system would present an empty behavior, other 28 present a model that is unique in the set of behaviors of this clusterization experiment. The remaining 88 clusters marked as «not valid» are not considered from now.

Table 5.6 lists the behavioral groups formed from the experiment. Every group is composed by a set of behavioral models, represented by a cluster number, having in common the API set presents in the old modeling system. In Group Behavior column, we writeAPI_1, API_2 to mean $B = API_1 \wedge API_2$

In each group we generate all the possible couples between graphs.

In each group, for each couple we evaluate if we can distinguish new behavioral models associated to each cluster of the couple. We can summarize this process with this formula

$$GroupDifference = \frac{\sum_{i=1}^{n} \delta_i}{n}$$

Where n is the number of couples in group, and δ_i is 0 if the two behavioral graphs representing the new models are not distinguishable, 1 otherwise. To perform this calculation, we have to redefine graph equality applying the definition for our specific case. In fact, graph equality is a sub-problem of graph isomorphism, that is an *NP-Hard* problem [7]. Due to this fact, Algorithm 5.1 is used to confront if two behaviors are distinguishable. There can be some rare false positive, but they will affect our results only in negative.

Table 5.5: API call presence in models

	Fingerprinted	**Not Fingerprinted, with dependency**
Present in 100% of cluster samples	In old and new model	In new model only
Present in 70%-100% of cluster samples	In old model only	Discarded
Present in less than 70% of cluster samples	Discarded	Discarded

Table 5.6: Behavioral Groups

Group	Number of Graphs	Group Behavior
1	6	'RegEnumValueW', 'RegOpenKeyExW'
2	4	'LdrGetProcedureAddress', 'NtCreateMutant'
3	5	'NtSetInformationFile'
4	26	'ZwMapViewOfSection'
5	24	'NtWriteFile', 'NtCreateFile'
6	6	'NtSetInformationFile', 'NtOpenFile'
7	11	'LdrGetProcedureAddress'
8	15	'NtQueryKey'
9	12	'NtCreateFile', 'NtReadFile'
10	3	'RegCreateKeyExW'
11	11	'NtSetInformationFile', 'NtCreateFile', 'NtReadFile'
12	5	'RegEnumKeyExW', 'RegOpenKeyExW'
13	3	'RegOpenKeyExW', 'RegEnumKeyW'
14	36	'NtCreateFile'
15	8	'NtSetInformationFile', 'NtCreateFile'
16	26	'NtOpenFile'
17	3	'NtOpenKey'
18	11	'RegOpenKeyExW'
19	2	'NtWriteFile', 'NtCreateFile', 'NtSetInformationFile', 'NtOpenFile'
20	4	'NtOpenFile', 'NtReadFile'
21	8	'NtWriteFile'
22	6	'NtDeviceIoControlFile'
23	6	'NtReadFile'
24	2	'NtCreateSection'
25	2	'RegOpenKeyExA'
26	3	'NtQueryDirectoryFile'
27	3	'RegCreateKeyExW', 'RegOpenKeyExW'
28	3	'NtQueryKey', 'LdrGetProcedureAddress'
29	6	'NtSetInformationFile', 'NtReadFile'
30	2	'NtSetInformationFile', 'NtCreateFile', 'NtWriteFile', 'NtOpenFile', 'NtReadFile'
31	3	'NtSetInformationFile', 'NtWriteFile'
32	2	'ZwMapViewOfSection', 'NtFreeVirtualMemory'
33	9	'NtWriteFile', 'NtCreateFile', 'NtSetInformationFile'
34	2	'NtSetInformationFile', 'NtCreateFile', 'NtWriteFile', 'NtReadFile'
35	2	'GetCursorPos', 'LdrGetProcedureAddress'
36	4	'GetCursorPos'
37	2	'GetSystemMetrics'

Algorithm 5.1 Graph inequality algorithm

```
1: procedure areTheSame(G1, G2)
2:     G1nodes ← G1.nodes
3:     G2nodes ← G2.nodes
4:     G1edges ← G1.edges
5:     G2edges ← G2.edges
6:     if Len(G1nodes)!=Len(G2nodes) then
7:         return False
8:     end if
9:     if Len(G1edges)!=Len(G2edges) then
10:        return False
11:    end if
12:    for all e1 ∈ G1edges do
13:        for all e2 ∈ G2edges do
14:            if e1 == e2 then
15:                G1edges.remove(e1)
16:                G2edges.remove(e2)
17:            end if
18:        end for
19:    end for
20:    if Len(G1edges)!=Len(G2edges) then
21:        return False
22:    end if
23:    return True
24: end procedure
```

Algorithm first confront number of nodes and edges, for performance reasons, then confronts the lists of the edges removing all the edges that perform the same API calls. Checking the number of nodes permits also to avoid ambiguities of graphs having same edges but different nodes as our graphs are completely connected components.

Performing this algorithm on every couple, results on the groups are the ones in Table 5.7

Average of the group means is set to $90,31\%$, but looking more significantly to total couples number distinguished behaviors are

$$\frac{1801}{2116} = 85,11\%$$

Those percentages indeed proof that due to the new modeling algorithm granularity of behavior we are able to distinguish is indeed growth. Validity of solution pro-

vided in this thesis is so proved, so in the next paragraph we are going to quantify qualitatively what are the differences between those models.

5.3.2 Differences in behavioral models

In the previous paragraph we have detected how many behaviors we are able to distinguish now thanks to new modeling system. In this paragraph we are now going to quantify how much those new behavioral models differs from the old behavior models and between themselves.

5.3.2.1 Node similarity

First of all, we have to consider that every group has an old behavioral model and a list of new behavioral model that are not distinguishable with the old system. Some of them may be equals between themselves but, for what said in paragraph 5.3.1, most of those behaviors are also different. The first similarity concept we introduce to measure this fact is node similarity. Node similarity measures how much the new behavioral models in each group are different from the old behavioral model that the group defines.

We are aware of the fact that the old behavioral model is characterized by a set of API calls: for this calculation, we approximate the API set of the old model with a graph that has one node for every API in the set, no matter the edges. Then we are going to compute Node similarity from this graph to each one of the other graphs in the group in this way

$$NS_{group} = \frac{1}{n} \sum_{i=1}^{n} NS_{0,i}$$

Where NS_{group} is the node similarity of the group, n is the number of graph in the group, $NS_{0,i}$ is the Node similarity between graph 0 (old behavioral graph) and the i-esime graph of the group. Going one step ahead, we have to define this similarity.

$$NS_{a,b} = \frac{1}{n}\sum_{i=1}^{n}\frac{APIpresent_i}{APItotal_i}$$

Where n is the length of the set of API calls of the two graph in exam[3], $APItotal_i$ is the number of times this API call is present in the new model, and $APIpresent_i$ is a coefficient that is set to 1 if the API is present in old and new model, 0 otherwise. What this means in concrete? It means that we are calculating similarities between the two models using similarity between API calls and we are computing similarity on the number of calls. The new models introduce the possibility to store more than one API call, so we assume that if API calls are identical (100% similarity) there is only one API call in both models. If there is more than one API call, similarity is calculated with the number of API calls in old model (1 or 0, *APIpresent*) on the total API calls in the new model (*APItotal*). So, for example, if there are two API calls on the new models, similarity is 50% for that API call.

With this model, 100% Node similarity in the group means that all the graphs have exactly one API call for each API call from the API call set of the old behavioral model. On the other hand, 0% Node similarity means that old and new model are composed by totally different API calls - this can happen for what we saw in paragraph 5.3.1.

5.3.2.2 Arc similarity

We want now to measure similarity between models of the same group and we can do it measuring similarity on arcs. As there are no arcs in old behavioral models, and we have no informations about taint dependencies, this similarity is only to compute differences in arcs between the new models and, for this reason, is not computed between the old behavioral model and the new ones, but for every couple of the group. The fact to define similarity between graph is, obviously, to complete the approach and quantify better differences between behavior we can distinguish. We can define the arc similarity of the group with this formula

[3] as we saw an API call can be present only in one of two models or in both

$$AS_{group} = \frac{1}{n} \sum_{i=1}^{n} \sum_{i=1}^{i} AS_{i,j}$$

Where n is the number of couples, $AS_{i,j}$ the arc similarity of the couple containing behaviors i and j. The Arc similarity between a couple of graphs is so calculated in this way

$$AS_{i,j} = \frac{ARCS_{commons}}{ARCS_{total}}$$

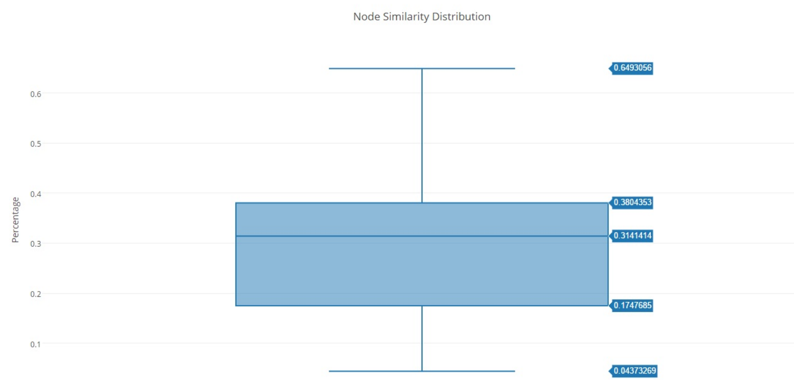
Where $ARCS_{commons}$ is the number of arcs graphs have in common. An arc is considered in common if it has the same API calls at the beginning and at the end. Algorithms considers also if a couple of arc in common is already processed or not - so, for example, if in the first graph the arc A->B is present twice and in the second graph once, similarity will be 2/3 (two arcs in common on a total of three arcs).

Arc similarity set to 0%, means that dependency between the two models are totally different in terms of API calls (or, in another way, every dependency in the first graph is not present on the second one and vice-versa). Arc similarity set to 100% means that the two graphs presents exactly the same dependencies: this does not guarantee the equality between the two graphs.

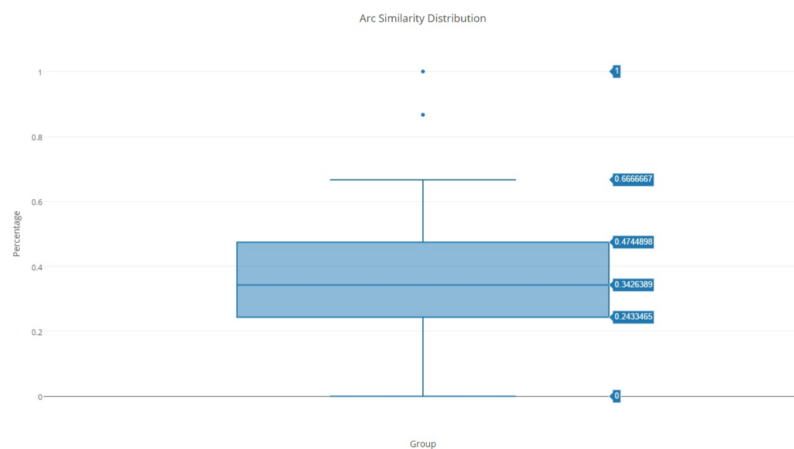
5.3.2.3 Similarity results in models

With the formulas exposed above, similarity are computed for each group. Results are exposed in table 5.8

In average, average of node similarity of the groups is *29,42%* and average of arc similarity of the groups is *37,24%*. In figure 5.6 we have a look of how this two variable are distributed in the groups using box plots. For what regards node similarities, distribution is around *31,4%* with *17,4%* as first quartile and 38% as third quartile: differences between original models are in most of the case less than *40%*: this means that using our models we have a lot of API calls that are presented more than once in a graph and that we cannot distinguish before. Values of arc similarity are between



(a) Node Similarity Distribution



(b) Arc Similarity Distribution

Figure 5.6: Similarity distributions

21,3% as first quartile, *34,26%* as mean, *66,66%* as third quartile, meaning that also dependencies between graphs of the same group are different.

5.4 Performances

The system provided is scalable due to the parallelization of the sample processing after the clustering phase and the fact we set a timeout for the algorithm execution. The main bottle-neck of the process is, without any doubt, the first part: dynamic analysis of the samples in cuckoo.

Analysis in cuckoo can be done in parallel creating more virtual machines but

this requires sufficient hardware capabilities: every Sandbox is a Virtual Machine with high resource demand in particular an high memory capability is needed. In our process our VMs hosts Windows 7 with 2GB of RAM and for our purposes, we analyzed the samples with 8 Virtual Machines, with 2 GB of RAM each.

Clusterization process system is on the other hand really efficient even if clusters are not processed in parallel: to cluster the first 3100 samples 1365 seconds are needed. We can define our clustering rate in our experiment as $2,26\frac{samples}{second}$. Clusterization process is linear too so clustering 10^6 samples takes approximately 122 hours in the verified hypothesis that clusterization time is also linear. As we can see in Figure 5.7 that seems not to be completely true: after a first phase clusterization time seems to follow a slower progression than a linear one.

This does not depend from clusterization algorithm itself: in fact the most expensive part in term of time of this work part is not the clustering action itself but the load of the gpickle files in which information from the sample analysis are stored. More big is the sample analysis, more time is needed to load the file: the time needed to cluster the file can be considered negligible respect to this one.

The decrease of slope in the graph is so due to the fact that the majority of big files are present in the first 2100 samples of the clusterization process: $\frac{1613}{1969} = 81\%$ of the big samples in approximately the first 66% of the dataset. Due to this fact, we can approximate also the clusterization time in a linear way: our graph is caused by a non-homogeneous distribution of sample analysis size in the cluster.

Synthesizing process depends form the cluster nature and strongly from the machine nature: Clusters are processed 4 at once, launched at intervals of 0.5 seconds: after this time, if the cluster doesn't terminate, and additional time of 20 seconds are given to this group of four clusters. When this time is passed process still in execution are killed. This means, in a group of 4 clusters in execution, that the first has 2 seconds to terminate, the second 1.5 seconds, the third 1 second and the fourth 0.5 seconds - after this time, if one of the four cluster is not terminated, other 20 seconds are given to the group.

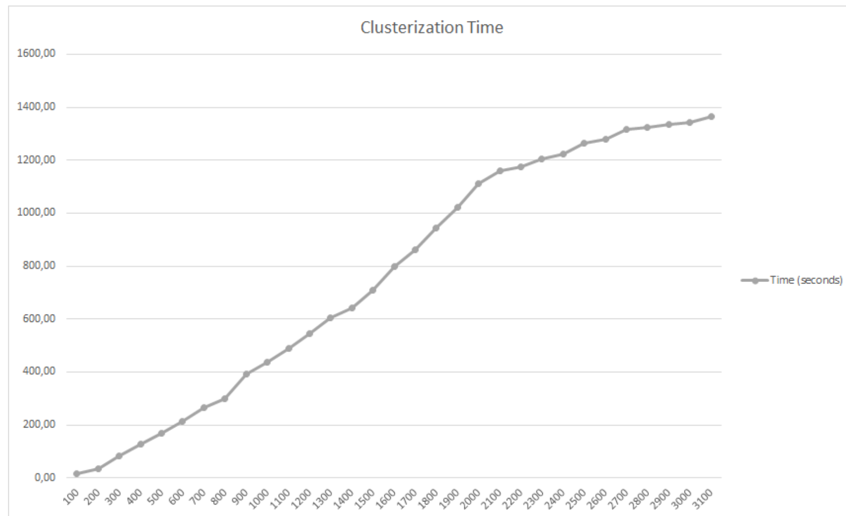


Figure 5.7: Clusterization Time

Due to this design execution a group of 4 cluster can terminate in 2 second or in 22 second. It it sure that a group terminates in 22 second if in the group at least one of four clusters is one of the clusters described as «not valid» in paragraph 5.1.1 because in those clusters execution is not terminated.

Table 5.7: Behavioral Differences

Group	Number of Group Couples	Different Couples Percentage	Different Couples Number
1	15	93,33%	14
2	6	100%	6
3	10	100%	10
4	325	79,69%	259
5	276	56,52%	156
6	15	80%	12
7	55	100%	55
8	105	100%	105
9	66	90,91%	60
10	33	100%	3
11	5	100%	55
12	10	100%	10
13	2	100%	3
14	630	93,33%	588
15	28	96,43%	27
16	325	83,08%	270
17	3	100%	3
18	55	81,82%	45
19	1	100%	1
20	6	50%	3
21	28	96,43%	27
22	10	40%	4
23	10	100%	10
24	1	0%	0
25	1	100%	1
26	3	100%	3
27	3	100%	3
28	3	100%	3
29	15	100%	15
30	1	100%	1
31	3	100%	3
32	1	100%	1
33	36	100%	36
34	1	100%	1
35	6	100%	6
36	1	100%	1
37	1	100%	1

Table 5.8: Similarity Results

Group	Number of Group Couples	Node Similarity	Arc Similarity
1	15	45,09%	52,75%
2	6	4,37%	26,98%
3	10	10,99%	20,67%
4	325	34,47%	53,49%
5	276	56,69%	55,38%
6	15	64,93%	50%
7	55	7,54%	27,92%
8	105	6,67%	43,5%
9	66	45,27%	33,23%
10	33	33,33%	31,43%
11	5	19,44%	24,26%
12	10	15,6%	24,36%
13	2	18,6%	44,76%
14	630	32,81%	30,89%
15	28	33,49%	36,5%
16	325	31,06%	44,88%
17	3	38,89%	19,87%
18	55	31,41%	40,34%
19	1	35,66%	30%
20	6	57,14%	52,38%
21	28	27,71%	34,26%
22	10	43,33%	86,67%
23	10	10,76%	34,9%
24	1	25%	100%
25	1	13,61%	4,35%
26	3	15,74%	52,38%
27	3	30,16%	31,28%
28	3	44,21%	46,6%
29	15	18,62%	27,42%
30	1	32,36%	16,67%
31	3	24,19%	19,35%
32	1	13,73%	66,67%
33	36	37,76%	37,48%
34	1	35,54%	19,05%
35	6	33,98%	22,9%
36	1	40,28%	34,46%
37	1	18,06%	0%

Chapter 6

Conclusions

In this thesis we have presented a new way to synthesize and extract malware behaviors from known malicious samples through automatic tools from large datasets. Models are presented into graphs that permits, respect to the old modeling system, to add some information about presence of multiple API calls of the same type. In addition of those information also in new models are present information about API dependencies that were not stored in the previous models. We have so proved that different malware have behaviors in common and have provided a more efficient way to distinct them than in Jackdaw [20] providing not only a quantitative measure of similarity and differences between the two behavioral models but also qualitative: we can distinguish 85% more behavioral models respect to the old modeling system and we can say that those models are in average 31% similar in a comparison with old modeling system and 34% similar between them. This approach however presents some limitations.

Bibliography

[1] New point-of-sale malware on underground markets for us2000. http://www.techworm.netl, December 2014.

[2] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[3] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[4] Silvio Cesare and Yang Xiang. *Software Similarity and Classification*. Springer Briefs in Computer Science. Springer, 2012.

[5] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim*, 3:2006, 2006.

[6] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[7] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. Algorithms for graph similarity and subgraph matching. 2011.

[8] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *In Proceedings of USENIX Security*, pages 255–270, 2004.

[9] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. In Robert H'obbes' Zakon, editor, *ACSAC*, pages 349–358. ACM, 2012.

[10] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[11] Science Magazine. 160,000 new malware samples arriving every day. `http://www.scmagazineuk.com`, May 2014.

[12] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):381–395, Oct 2010.

[13] McAfee. Infographic: The state of malware 2013. `http://www.mcafee.com/us/security-awareness/articles/state-of-malware-2013.aspx`, April 2013.

[14] Microsoft. Msdn. `http://msdn.microsoft.com`, 2015.

[15] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[16] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, February 2006.

[18] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[19] PandaLabs. Pandalabs annual report 2014. `http://www.pandasecurity.com/mediacenter/src/uploads/2015/02/Pandalabs2014-DEF2-en.pdf`, March 2015.

[20] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 9148 of *Lecture Notes in Computer Science*, pages 121–143. Springer International Publishing, 2015.

[21] Robert Sedgewick. *Algorithms in C - part 5: graph algorithms (3 .ed.)*. Addison-Wesley-Longman, 2002.

[22] Qun Song and Nikola Kasabov. Ecm - a novel on-line, evolving clustering method and its applications. In *In M. I. Posner (Ed.), Foundations of cognitive science*, pages 631–682. The MIT Press, 2001.

[23] Tarkan Yetiser. Polymorphic viruses - implementation, detection, and protection. `http://vxheaven.org/lib/ayt01.htmll`, January 1993.