Ciprian OPRIȘA

Machine learning techniques for the analysis and detection of malicious software

PhD Thesis

UTPRESS
Cluj-Napoca, 2021
ISBN 978-606-737-511-4

Ciprian Oprișa

MACHINE LEARNING TECHNIQUES FOR THE ANALYSIS AND DETECTION OF MALICIOUS SOFTWARE

PhD Thesis

UTPRESS
Cluj-Napoca, 2021
ISBN 978-606-737-511-4

Editura U.T.Press
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.: 0264-401999
e-mail: utpress@biblio.utcluj.ro
https://biblioteca.utcluj.ro/editura

Director: ing. Călin Câmpean

Recenzia: prof.dr.ing. Gheorghe Sebestyen Pal
prof.dr.ing. Nicolae Țăpuș
prof.dr. Bazil Pârv
prof.dr.ing. Rodica Potolea

ISBN 978-606-737-511-4
Bun de tipar: 12.05.2021

Contents

List of Figures

List of Tables

Foreword

The current work is my PhD thesis that was written at the Technical University of Cluj-Napoca between 2013 and 2016 under the supervision of Prof. Dr. Eng. Gheorghe Sebestyen Pal. The thesis was defended on June 7$^{\text{th}}$ 2016 to a committee presided by Prof. Dr. Eng. Liviu Miclea. The committee members were Prof. Dr. Eng. Gheorghe Sebestyen Pal, Prof. Dr. Eng. Nicolae Țăpuș, Prof. Dr. Bazil Pârv and Prof. Dr. Eng. Rodica Potolea. The thesis was accepted with the distinction *summa cum laude*.

Cybersecurity was an important topic during my PhD years and it continued to grow with new threats and new defenses. Malicious software, or *malware*, continues to be a threat in 2021 and its scale continues to require the use of machine learning and big data techniques. While this techniques have evolved over the years, many concepts from this thesis like software similarity and clustering of binary files are still used today.

The thesis focuses on the practical aspects of malware detection while also addressing the theoretical issues. Malware detection is more than a classification problem and the client side of an anti-malware program is just the tip of the iceberg. In a security company there are many internal processes for building robust detection models and to analyze the new malware families and cybercrime campaigns. The experiments from the thesis were performed on real-world data provided by the security company Bitdefender, where I also work, some of them in collaboration with my colleagues. Some of the results in the thesis were deployed to ensure the security of hundreds of millions of customers and are still being used in 2021. The theoretical results have been disseminated in more than 15 research papers on which this thesis is based. These papers received more than 50 independent citations during the thesis years and after.

I would like to express my gratitude to my coordinator for his valuable advice during the thesis years, to my colleagues from the Technical University of Cluj-Napoca and from Bitdefender that I worked with, to all my teachers that helped me develop a rigorous thinking and to my family for their constant support.

The Author

Abstract

From the abstract concept of self-reproducing automata to recent cases of cyber-terrorism, malicious software is one of the greatest challenges to security. In the last few years we have witnessed an exponential growth in the number of new malware samples, reaching 400,000 unique files in a single day. In order for the security solutions to handle these emerging threats, all new samples needs to be analyzed and categorized, a process that can no longer be performed manually. This thesis proposes solutions inspired from the machine learning area for automating most of the analysis tasks and reducing the amount of manual work.

The malware landscape is divided into malware families, each family containing many variants that perform the same actions but are rewritten or obfuscated to look syntactically different. This thesis proposes several approaches for comparing binary samples and finding similarities that will help identifying samples that belong to the same malware family and also find plagiarized code.

Being able to compute the distance between two programs is not enough to analyze large collections. Cluster analysis, as a form of unsupervised machine learning can group together similar samples. The shortcoming of classical clustering algorithms is that their running time is quadratic in the number of inputs, which is unacceptable in real-world scenario. This shortcoming is addressed by proposing new clustering algorithms that produce a very close approximation of the required clusters while dramatically improving the running time.

One recurring problem in malware research that cannot be fully automated is deciding whether a new sample is clean or malicious. Based on the assumption that similar programs are likely to share the same verdict, this thesis presents algorithms for selecting the most relevant samples from a collection to be analyzed, inferring the correct verdict for the rest of them. Combined with the fast clustering algorithms, these proposal can considerably reduce the amount of manual analysis.

Another problem solved in this thesis is identifying plagiarism cases in large collections, such as Android applications markets. This problem is difficult because it requires more than simply performing pairwise comparison between all applications. Some practical aspects like eliminating the legally shared code such as common libraries or ignoring pairs of applications developed by the same entity must be considered. The scalability problem is also addressed, as applications markets are continuously increasing.

Chapter 1

Introduction

1.1 Motivation

Malicious software or *malware* is a type of program created with the purpose of violating a system's security. Regardless of form, this software is a threat for every user and must be detected before causing any harm.

By Bishop's definition [Bis02], "*malicious logic* is a set of instructions that causes a site's security policy to be violated". We will consider a program to be malicious if it contains any malicious logic. As the author of [Bis02] pinpoints, this malicious logic is difficult to detect even by humans. For instance, a sequence of instructions that encrypt the user's personal files might be contained in a benign program that helps the user to protect his privacy. The same sequence of instructions could be used to encrypt the user's personal files in order to demand a ransom for restoring them. The only difference is that in the first case, the user desires the effect of the program (i.e. the encryption of the files) while, in the second case, the user is rather *tricked* into running the malware.

The difficulty of malicious logic detection was shown more formally by Cohen. His paper [Coh89] focuses on a particular type of malicious logic, called *computer virus* (formally defined as a sequence of symbols that replicate themselves in a Turing machine). The paper states that "it is undecidable whether an arbitrary program contains a computer virus". The statement is proved by reducing the virus detection to the halting problem [Tur36].

Various attempts have been made to design a system where malicious logic could cause no harm [Bis02]. Unfortunately, such systems could not eliminate malicious logic completely, while the usability was dramatically reduced.

The current solution for protecting against malware is to use anti-virus programs. Such programs usually detect if a software is malicious by checking it against a database of known malicious samples. This type of detection will be called *signature-based detection* since the malware database contains the signatures extracted from the known malware. The problem with this approach is the lack of proactivity. A new malware sample cannot be detected if it hasn't been previously seen and categorized by the anti-virus vendor. Since the number of new malware samples that appear everyday has surpassed more than 300,000 [AT16], simply working with malware collections has become a difficult task.

Figure 1.1 divides malware detection into three components:

- source channel detection: the malware is blocked before it reaches the target system by blacklisting known malware channels (like web sites that contain malware). **A program is detected if it comes from a malicious source.**

- static detection: the malware is detected after it gets downloaded into the system but before it has the chance to run. At this point, the binary program is available but it is in a passive state (it doesn't run yet). **A program is detected if it looks like malware.**
- dynamic detection: the malware is detected by the actions performed at run-time. **A program is detected if it behaves like malware.**

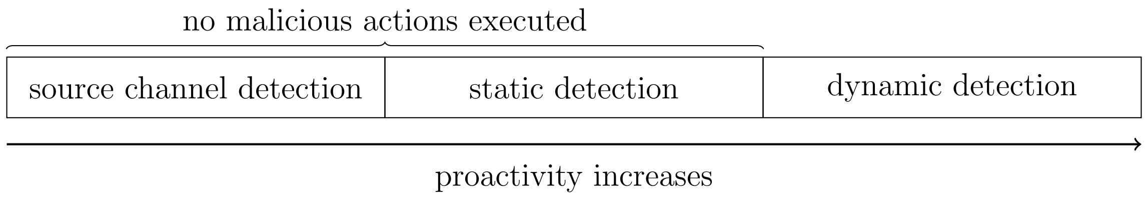


Figure 1.1: Proactivity by detection method

If the malware is detected in the first two phases, it cannot cause any harm to the user, as no malicious action gets to be executed. In the third phase, the malware is already running and some of the malicious actions cannot be undone. For example, if the malware sends the victim's passwords to the attacker, the malicious purpose has already been achieved.

As the figure shows, proactivity grows from left to right. A source channel cannot be blacklisted until it was previously seen, so this type of detection has a low proactivity. Dynamic detection is based on the actions that the malware performs on the victim's system. By detecting the malicious behavior a higher proactivity can be achieved, but with a cost: by the time of detection, some harm has already been done.

Static detection comes with its advantages and challenges. An anti-virus scanner has the chance to examine the binary program before it runs. Unfortunately, malware authors do everything they can to make detection harder. Obfuscation techniques [LD03] are used to modify an existing program in order to make its binary content difficult to recognize while maintaining the semantics. Most research efforts focus on detecting obfuscated programs by finding invariant features in existing malware. Using these features as signatures, new malware samples from the same family can be proactively detected. However, the task of clustering malware samples by family is getting harder, giving the increasing number of new samples.

As the malware landscape is getting more crowded with new samples and more sophisticated with new techniques, it is imperative to find new defense solutions. This thesis aims to provide such solutions by employing techniques from the machine learning area, based on features extracted through static analysis on binary programs.

1.2 Thesis Overview

The following chapter will present the concepts and definitions used along this thesis. This presentation is divided into two sections. Section 2.1 presents an overview of the anti-malware research field, while section 2.2 will outline the machine learning concepts used throughout this thesis.

Chapter 3 describes other efforts into applying machine learning for anti-malware tasks and pinpoints the gaps filled by the current thesis. The first section of this chapter highlights related efforts in identifying similar content for both source code and binary programs, including applications where machine learning classifiers were used to predict malware based on similar features. Section 3.2 describes other attempts to cluster large datasets using hashing techniques.

Chapter 4 presents techniques for analyzing programs and measuring their similarity. Our approach is based on code analysis and the first section outlines the process of extracting code features from various program formats. Section 4.2 shows how to design distance functions based on these features. The code features extracted from a program can be treated as a large sequence of operations and represented as a string or can be split into smaller sequences and represented as a set. Both approaches can produce viable distance metrics. Some binary programs may exhibit structural characteristics (like the division into packages, classes and methods). A distance function that takes such features into consideration may provide better results, as shown in section 4.3. Section 4.4 considers the problem of feature selection. We will provide a technique to decide what code features should be used to discriminate between clean and malware samples and to get a higher similarity between obfuscated versions of the same program. The features and distances proposed in this chapter will be tested in section 4.5 that provides the experimental results.

Chapter 5 uses the distance functions discussed in the previous chapter to perform cluster analysis on large collections of binary programs. The chapter emphasizes the difficulty of clustering large collections, as classical algorithms are based on computing the similarity for each pair of samples. We will propose two approaches for avoiding these expensive computations. The one in section 5.1 takes advantage of the Suffix Tree data structure used to compute the *deq distance* from section 4.2, while section 5.2 shows how to use Locality-Sensitive Hashing to filter the pairs of samples where distance computations is required. The experimental results in section 5.3 will show that our algorithms can cluster 10 million samples in a couple of hours.

Chapter 6 shows how to use clustering in malware analysis. Two techniques for reducing the amount of human work are proposed in sections 6.1 and 6.2. The first one selects the most representative samples from a cluster to be further analyzed, while the second one infers the verdicts (clean or malicious) for new samples based on verdicts already given on similar ones.

A complete system for detecting plagiarism cases in large collections, like an Android applications market is presented in chapter 7. The system is based on the results from previous chapters but also deals with practical issues. The chapter takes into account the attack vectors used for avoiding detection in section 7.1 and avoids flagging applications developed by the same entity as a plagiarism case. The system's architecture based Map-Reduce paradigm is presented in section 7.3, which is followed by the experimental findings in section 7.4.

The thesis conclusions, along with the main contributions are presented in chapter 8.

Chapter 2

Concepts and Definitions

2.1 Concepts and Definitions from Anti-malware Research

As stated in the Introduction chapter, malware is a category of software created with the purpose of violating a system's security, by harming users, computers or networks. Anti-malware research deals with the analysis of malicious software in order to understand how it works and combat future attacks of the same type. This section will provide the basic terminology and techniques used in this area of research.

2.1.1 Samples fingerprinting through hashing

The main data sources in anti-malware research are collections of files. A file can be anything, like a binary program, a document, a picture or an archive. Every file has characteristics and metadata. The characteristic are intrinsic properties like the file size or the number of pixels for an image. Some other properties like the provenience of the file or the file name cannot be derived from the file content and we will call them metadata.

Files collections are usually large. For instance, AV-TEST [AT16] collected more than 500 million malware samples over the years. Similar collections of clean files can be estimated to have the same order of magnitude.

Whenever a new sample is encountered, the first problem that arises is whether the sample already exists in a given collection or is a new sample. The naive solution for this problem would be to compare the content of the new file with the content of each sample existent in the collection. Such an exhaustive search may determine the appartenance of a sample to a given collection but is unpractical for large collections. One may argue that we can stop at the first different byte when comparing two samples, so most of the times we will stop after a few bytes. In practice, the bytes in the file content are not uniformly distributed. For instance, self-extracting archives start with the extractor code, having a large number of common bytes at the beginning.

For efficient comparison, we would need a mapping function that transform arbitrary-length content into a fixed-size value, uniformly distributed over the values set. Such a function is called a hash function and is formally defined in Equation 2.1, the concept dating back from 1953 [Knu98].

$$h : \mathcal{S} \rightarrow \mathcal{V} \tag{2.1}$$

The set $\mathcal{S}$ is an infinite set, consisting of all binary contents, regardless of length. If $\Sigma = \{0, 1, 2, \ldots 2^8 - 1\}$ is the alphabet containing all bytes values, $\mathcal{S} = \Sigma^\star$ can be defined as the set of all words built with symbols from Σ. The set of values $\mathcal{V}$, on the other hand, is a finite set that can be represented on a fixed number of bits. We will say that a hash is n bits long if $|\mathcal{V}| = 2^n$.

Since $\mathcal{S}$ is infinite while $\mathcal{V}$ is finite, no hash function can be injective. This means that $\exists X, Y \in \mathcal{S}$, such that $X \neq Y$ and $h(X) = h(Y)$. Such a case is called a *collision* and is unavoidable in hashing. However, the collision probability is very low for hashes with a sufficient number of bits. For example, the MD5 hash [WY05] is 128 bits long and the probability to find a random collision in a collection of 10^{10} samples is smaller than 10^{-18}.

If a large collection of samples is indexed by the content hash, samples retrieval becomes an easy task. Finding the hash value of a sample in a given collection can determine the appartenance with a very high probability.

The utility of hash functions can be extended in order to detect malicious samples on a user's computer. If we compute the hash on a given sample and find it in the malware hashes collection, we can infer that the found sample is malware. This approach has two fundamental issues. The first issue is performance. In order to compute a hash value for a file, the entire file content must be read from the disk. Usually disk access operations are costlier than other computations so they need to be avoided. A possible approach is to select only some *representative* regions of the file for hash computations. Hash values computed on these regions are called *signatures* and are widely used in anti-virus products [SH12]. The second issue is the lack of proactivity for such signature-based detection. If a new version of the same malware has a single byte changed, the hash value will be different and the signature will not match. One of the contributions of this thesis is the extraction of features that are more robust to changes (section 4.1).

2.1.2 Structural analysis of programs

Every sample is represented as raw binary data. Although some useful information (like file size or strings) can be extracted from such data, most features are extracted by parsing the sample's structure.

The first information of interest is the file type. Some tools for identifying the file format already exist, the most notable being the `file` command and `libmagic` library in Unix environments. These tools have a high accuracy in identifying most known file formats but they are limited to existing patterns. Also, some files can belong to more than one format. For example, a self-extracting archive is both an executable file and an archive. An Android application (APK file format) is also a JAR (Java ARchive) and a ZIP archive.

In anti-malware research, the most important file formats are those that contain executable code. Example of such file formats are:

- Microsoft Portable Executables (Windows executable files)
- ELF files (the most common program format on Linux and Unix distributions)
- Android APK files
- script files (Javascript, Visual Basic Script, Python, Perl)
- documents with active content (PDF files with Javascript, Microsoft Office documents with macros)

In theory, any data file can have malicious content, as it can contain an exploit specially crafted to thwart and take control of the executable used to process the data. Exploits are an important topic in security research but they will not be detailed in this thesis.

Binary programs usually contain code, data and metadata. Most features used in this thesis will be extracted from code, but the other two components must be parsed as well. Metadata will provide high-level information and will help the analysis tool to split the program into logical parts. A program's data can provide valuable insights, like the language of the program's author, Internet resources that might be accessed at runtime or encryption keys used in the program's activity.

2.1.2.1 The Portable Executable file format

Most malware threats are found on the Windows platform, so, the most explored file format will be the Portable Executable. In [Pie02], Matt Pietrek provides an in-depth look of this file format.

One of the key ideas of the format is that the file layout on the disk resembles the memory layout. Every PE file begins with a header that contains most of the file metadata and is followed by several sections. The section form a logical division of the program, as each section can contain code, data or resources.

Some differences between the file layout and the memory layout may occur. Some section may have different sizes on disk than in memory, due to alignment differences (on disk, the sections are sector-aligned, which is usually 512 bytes and in memory the sections are page-aligned, which is usually 4096 bytes) or due to uninitialized data (such data will not be present on disk). To speed-up run-time operations, most addresses in the PE file format are expressed as Relative Virtual Addresses (RVA), which are memory addresses relative to the Image Base (the address at which the module will be loaded into the virtual memory). To transform a RVA into a file address, one needs to verify which section the address falls into, then subtract the section's RVA and add the section's file address.

Each section is characterized by a name, file address and RVA, file size and memory size and some characteristics that may hint the section's contents. Based on these characteristics we can determine if a given section contains code or not. These characteristics can be modified at run-time, a technique often used by packers to obscure the program's code from static analysis.

2.1.3 Disassembling and decompiling

Disassembling is the process that reverses binary code to Assembly instructions. Decompiling is a similar process, that reverses binary code to original source code.

Given the fact that Intel x86 assembly instructions map one to one to binary code [Int13], the disassembly process is easy to perform. As for the decompiling, it is usually impossible to revert a native program to the original source code. This is because the variable and function names are usually lost and different C syntax constructs can be translated to the same binary code.

The challenges of code disassembly are presented in [SDA02]. The paper states that "code disassembly routines form a fundamental component of software systems that statically analyze or modify executable programs". Two challenges are identified for Intel x86 disassembly: indirect jumps and presence of data inside code sections. Such challenges will not be present on other platforms like the Common Intermediate Language in the .NET framework [Int12].

The most used disassembly strategies are:

- linear sweep: decodes each instruction and proceeds to the next one, by adding the current instruction's length to the buffer position. The method is straightforward but will also try to disassemble any data found among the assembly instructions, rendering inaccurate results.

- recursive traversal: this strategy follows the program flow by considering instructions like jumps, returns or branches. In case of conditional branches, both alternatives are taken into account. Indirect jumps (jumps where the destination is only computed at run-time) may mislead a recursive traversal disassembly tool.

The most popular disassembly tool is IDA pro [Eag11], an interactive disassembler that uses many heuristics and expert knowledge to deal with the aforementioned issue. IDA pro users can also decompile the analyzed code using the Hex-Rays Decompiler, a plugin able to infer many high-level language constructs.

2.1.4 Programs obfuscation and detection resistance

In order to detect and classify malware, we must first understand how malware authors protect it.

An older but still relevant synthesis work on modern malware belongs to Ször [SF01]. The paper is dealing with different types of file infectors (also called viruses) but the employed techniques are used in other malware types, not only at the time of writing the paper (2001) but even nowadays.

The paper starts by describing the evolution of virus code. In order to avoid detection, the malware payload (or the malicious logic) was encrypted and a small decryption stub was used to decrypt it at run-time. Since the decryption key was different each time, hash-based detection will fail. Indeed, computing a hash on the binary zone where the malicious code resides and checking the result against a malware database cannot work, since that zone will be different for every sample. A possible solution is to sign the code stub that unpacks the payload. This will work if the stub is specific enough so we won't have false positives on clean files. Also a necessary condition is that the stub doesn't change.

The first attempt to resist this kind of detection was the use of oligomorphic viruses. A Windows 95 virus called Memorial was able to change its decryption stub during the infection of new samples. However, he could only generate 96 distinct patterns. Even if the number is finite and not very large, it is not practical for an anti-virus engine to use 96 signatures in order to detect a single malware. The malware evolution continued with polymorphic viruses. They also change the decryption stub dynamically like an oligomorphic virus but the number of variations is not limited. A solution to detect both oligomorphic and polymorphic malware is to use code emulators. These emulators can interpret the decryption stub and execute it in a virtual environment. If the code is interpreted correctly, at some point the malicious payload will get decrypted.

The next step for malware writers was the *metamorphic virus*. As [SF01] explains, "metamorphic viruses do not have a decryptor, or a constant virus body. However, they are able to create new generations that look different. Metamorphic viruses do not use a constant data area filled with string constants but have one single code body that carries data as code". This means that the original code of the malware will never be revealed because the processor runs the modified code directly, as there is no need to unpack it. The paper presents first some examples of metamorphic engines. It starts with simple techniques like swapping the instruction's registers, permuting entire blocks of code or adding garbage instructions. The most notable piece of malware presented was the Zmist virus that could disassemble the binary code of a program and insert it's own instructions between them in such a way that both the host and the virus would be executed.

Several detection techniques are presented but none of them can be considered *the magic bullet* against this type of malware. Geometric detection is based on observing the structural particularities of the malware file. Unfortunately, such a method can have a large number of false positives. Disassembling techniques looked more promising but they can't detect any type of malware. Some amount of manual work is also required for deciding which instructions can cause a sample to be malicious. Emulators can also be used to observe the malicious behavior in an isolated environment. Unfortunately, nowadays malware comes with a bag of tricks to detect emulators.

The limits of the emulators are further discussed in a more recent paper [PMRB09]. In order to evade detection, malware samples use pieces of code called *red pills* that behave differently in an emulated environment than in a real one. This emulators limitation comes from the fact that the processor and the operating system are quite complex and emulating them completely would come with a great performance decrease. The paper showed how these *red pills* can be generated automatically. More attacks on emulators and virtual machines were presented in [Fer07]. The conclusion of the author is that in some cases there is no method to prevent the malware from detecting that it runs in a controlled environment. Fortunately, as more and more systems run in virtual environments, the attacker will have to change their tactics or risk not running at all on the victims systems.

The malware authors also try to make their creations resistant to static analysis by *thwarting* the disassemblers. The paper [LD03] proposes two disassembly methods called "linear sweep" and "recursive traversal". The disassembly process is generally difficult because the Intel x86 assembly instructions vary in length [Int13]. The linear sweep approach starts at a given point in the binary code then determines the position of the next instruction by determining the length of the current disassembled instruction. The problem with this approach, as stated in [LD03] is that "it is prone to disassembly errors resulting from the misinterpretation of data that is embedded in the instruction stream". The recursive traversal tries to fix this by considering the control flow of the program and by following the encountered branches. The paper shows several techniques to thwart both disassembly approaches. The result is that anti-virus scanner might incorrectly disassembly some programs and fail to detected them.

Because the task of hiding the malicious payload is different from the task of writing it, some malware authors prefer to do the hiding part by using *packers*. A packer is a program (free or commercial) that can be used to shrink the size of an executable, to hide the program's code from reverse engineers and sometimes to detect if the program is run in a controlled environment like an emulator or a virtual machine. We can't say a program is malicious just because it's packed. Many legitimate applications use packers to protect their intellectual property. An anti-virus program must be able to unpack a program while scanning it or at least discern between a malware and a clean application when they are both packed. If they fail to do so, they either can't detected the packed malware or cause a large number of false positives. Unfortunately, as the authors of [SH12] concluded, "there are no good publicly available automated dynamic unpackers. Many publicly available tools will do an adequate job on some packers, but none is quite ready for serious usage".

2.2 Concepts and Definitions from Machine Learning

2.2.1 Distance functions and similarity computation

The third chapter of [RU12] explores the concept of "Finding similar items", a fundamental concept in machine learning and data mining. Several applications like plagiarism detection, mirror web pages or collaborative filtering are presented.

The basic similarity function is the Jaccard similarity [Jac01], that examines the intersection and the union of two sets for computing their similarity. Equation 2.2 formally defines this similarity as the ratio between those measures.

$$sim(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{2.2}$$

One problem with Jaccard similarity is that it works on sets and not all real-world items are represented as sets. A technique called n-grams extraction or *shingling* can transform a document into a set. If we are analyzing text documents, an n-gram can be defined as a sequence of n consecutive letters or a sequence of n consecutive words. With binary code, we can replace letters or words by assembly instructions or operations.

In order to detect near-duplicates in a collection without performing pairwise comparison, "similarity-preserving summaries of sets" can be used. Such a summary can be obtained by employing locality-sensitive hashing [IM98], a technique that is further explored in section 5.2.

Other commonly-used distance functions are:

- Euclidean, Manhattan or cosine distance for vector spaces
- edit or Levenshtein distance for strings
- Hamming distance for sets

2.2.2 Cluster analysis

Clustering is the unsupervised learning process of grouping items in clusters based on a distance function, in such a way that similar items will end up in the same cluster, while dissimilar items will end up in different clusters.

According to [RU12] there are two main clustering strategies:

- *point assignment* - for this strategy, some initial estimation of the clusters is performed at first, like starting with some centroids. Then, each point is assigned to the cluster that it fits best.
- *hierarchical or agglomerative clustering* - algorithms from this class start by assigning each point to a different cluster then proceed by combining smaller clusters into bigger ones, based on similarity.

The most popular clustering approach that fits into the first category is k-means [M$^+$67]. The standard algorithm was published by Lloyd [Llo82]. Unfortunately, the algorithm requires an input parameter k that is hard to estimate for some collections.

Hierarchical clustering, also called linkage, seems more appropriate for grouping together binary programs that are similar to each other. The first element required for performing this kind of clustering is a metric distance that computes how dissimilar two items are. Any metric distance discussed in the previous subsection can be used with hierarchical clustering algorithms.

The linkage criteria defines the strategy of joining together two existing clusters in order to form a bigger one. Based on this criteria, the hierarchical clustering can also be divided into several strategies:

- *complete linkage* - in this case, the distance between two clusters will be defined as the maximum distance between an element belonging to the first cluster and an element belonging to the second cluster. Two clusters will be joined if their distance is smaller than a given threshold. In other words, we will only join two clusters if the resulting cluster will have each pair of elements similar to each other. The complete linkage idea was firstly introduced in [Sør48]. The naive algorithm has $O(n^3)$ complexity but an $O(n^2)$ algorithm was given by Defays in [Def77].
- *single linkage* - two clusters will also be joined based on their distance, but the distance definition is different. Instead of computing the maximum distance between elements of the two cluster, we will consider the minimum distance. This definition favors clusters joining more than the complete linkage case, usually producing a smaller number of larger clusters. The single linkage algorithm was introduced by Sibson in [Sib73] and has $O(n^2)$ complexity. The author proved that the algorithm is optimal, so we cannot achieve a better performance (in asymptotic terms) and guarantee to obtain the same results. One drawback of this approach is the chaining effect. If an item A is similar to an item B and item B is similar to an item C, A and C are not necessarily similar but will still end up in the same cluster. This may lead to heterogeneous clusters.
- *average linkage* - same as above, the criteria for joining two clusters is based on their distance. In this case, the distance is defined as the average distance between every pair of samples where one item belongs to the first cluster and the other item to the second cluster. The method was introduced in [Sok58] and an $O(n^2)$ algorithm based on UPGMA trees was given in [Mur84].

Other clustering approaches like distribution-based clustering and density-based clustering also exist. Malware collections are difficult to model as distribution-based clusters especially because they are not a natural phenomenon. They are artificially created by humans with the explicit purpose of being as different as possible from each other in order to avoid anti-virus detection. Density-based clustering shows promising algorithms like DBSCAN [EKSX96] and a more recent version called OPTICS [ABKS99], but they haven't been tried yet for clustering binary programs or malware samples.

2.2.3 Classification

Classification is the supervised machine learning process that learns *classification rules* based on labeled training data and uses such rules to make predictions on unlabeled data. Usually, classifiers learn two classes that we will call positive and negative. Such a classifier can be extended to learn any number of classes, by training a separate classifier to predict whether a sample belongs to a specific class or not.

Some popular classifiers are:

- Naive Bayes classifiers [FS97] are probabilistic classifiers based on the assumptions that all features are independent.
- Decision Trees [Qui86] are tree-like predictive models where internal nodes represent decisions based on features and the leaves are class labels. One advantage of Decision Tree Learning is that it builds a white box model that is easy to understand and interpret by humans.
- Support-Vector Machines [BGV92] classify data by building a hyperplane that separates positive instances from negative ones. The hyperplane is chosen in such a way to minimize the separation margin (the distance from the closest data point to the hyperplane). When linear separation is not possible, the features can be mapped into higher dimensions using kernel functions [STC04].
- Artificial Neural Networks [MP43] use a model inspired by biological neural networks where interconnected *neurons* exchange messages between each other. A neural network can be trained using algorithms like backpropagation [RHW88].

Even poor classifiers are useful. Boosting techniques can combine classifier with accuracy slightly better than random into a powerful and highly accurate classifier. An example of such a boosting algorithm is AdaBoost [FS97].

2.2.4 Quality evaluation

Machine learning algorithms should be cross-validated on independent data sets in order to asses their quality. Usually, the data samples are partitioned into *training set* and *validation set* (also known as *testing set*). The algorithm is trained on the first set and validated on the second one.

In case of classifiers, we are interested whether the correct class was predicted or not, for each sample. In case of clustering algorithms, each pair of items can belong to the same cluster or not. We will use each pair as a validation sample. Since clustering belongs to non-supervised machine learning, we don't need labeled data for training so the entire dataset can be used for validation.

For each sample in the validation set, we will assign one of the labels from the confusion matrix [KP98], as in Table 2.1.

Table 2.1: Confusion matrix

		Predicted	
		Similar	Dissimilar
Actual	Similar	TP	FN
	Dissimilar	FP	TN

Based on the relation between the prediction and the actual label, each sample will be counted into one of the following four categories:

- True Positive (TP) - Both predicted and actual value are positive. For clustering, both tested algorithm and expert methods placed the pair of samples into the same cluster.

- False Positive (FP) - The algorithm predicted a positive result but expert classification decided otherwise.
- True Negative (FN) - The actual result is positive but the algorithm failed to predict that. In case of clustering, the two items should belong to the same cluster but were placed in different ones.
- False Negative (TN) - Both predicted and actual value are negative.

Several quality metrics can be derived from the number of samples with each label:

- Accuracy - is the fraction of correctly classified samples.

$$ACC = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.3}$$

  This metric can be biased by the large number of True Negative instances, especially for assessing clusters quality. For instance, a clustering algorithm that places every item into a different cluster will have a high Accuracy. For the reason, this metric is not used too often.

- Precision - is the fraction of retrieved instances that are relevant.

$$P = \frac{TP}{TP + FP} \tag{2.4}$$

- Recall - is the fraction of relevant instances that are retrieved.

$$R = \frac{TP}{TP + FN} \tag{2.5}$$

- F-Measure [Rij79] - is a measure of a test's accuracy. It considers both the Precision P and the Recall R of the test to compute the score. The F-Measure can be interpreted as a weighted average of the precision and recall, and it reaches its best value at 1 and the worst at 0. A constant $\beta \geq 0$ can be used to balance the contribution of false negatives.

$$F_\beta = \frac{(\beta^2 + 1) \cdot P \cdot R}{\beta^2 \cdot P + R} \tag{2.6}$$

  The usual value chosen for the constant β is 1, as Precision and Recall are equally important.

In order to illustrate how the Precision and Recall vary with the similarity threshold, in some cases we will analyze the receiver operating characteristic (ROC) [Faw06]. The threshold will be varied from 0% to 100%, using some increment. At each step, we computed the number of True Positives (TP), False Positives (FP) and False Negatives (FN).

The true positive rate is identical to the Recall index previously discussed and can be computed as $\frac{TP}{TP + FN}$. The false positive rate can be calculated as 1 - Precision, or $\frac{FP}{FP + TP}$.

A ROC curve basically plots the dependency between the two indices. Each point on the curve corresponds to the false positive rate and true positive rate for a given threshold. A perfect classifier would pass through the point $(0, 1)$ - that corresponds to 100% true positives and 0% false positives. A random guess would result in a point along a diagonal line (called *line of no discrimination*) from the left bottom to the top right corners. A good classifier should be above this line.

Chapter 3

State of the Art

3.1 Identifying Similar Content

The issue of comparing binary files for finding similar but non-identical items have been previously studied.

A tool called *sif* was presented in [M$^+$94] and was able to find similar files in a large file system. It was one of the first attempts to apply fingerprinting techniques in order to find matches in a large collection. The authors considered two files to be similar if "they contain a significant number of common substrings that are not too small". The technique proposed by the author was to extract several fingerprints from each file. A fingerprint was defined as a hash on a substring. In order to avoid extracting a very large number of fingerprints, only the ones containing certain "anchors" were considered.

The advantage of this approach is that most of the fingerprints of a file will remain the same, even if some random characters are inserted or deleted. If substrings starting from certain positions would have been considered for fingerprints, a simple insertion at the beginning of the file would change all of them.

The method presented in [M$^+$94] will work on many file formats because it does not consider the semantics of the file while extracting fingerprints. Unfortunately, this will be a weakness for identifying similar programs, because simple modifications may change the operands of most instructions, leaving only small common substrings. The approach in this thesis is more robust because it disassembles the binary code and only considers the operations performed, which are less likely to change.

Heintze, in [H$^+$96] approached the scalability problem. The presented solution only takes into account a fixed number of fingerprints, in order to preserve storage space. The paper states that the total number of substrings of length α from a document of length l is $l-\alpha+1$, which is too many for efficient storage. Although both storage capacity and speed have increased in the last 20 year, the paper still gives some interesting approaches for selecting a small number of relevant fingerprints. The substrings frequency is considered, as frequent fingerprints will lead to a large number of false positives. The substring were selected based on the frequency of the first 5 letters. Substrings with the lowest frequency for these 5-letter groups will lead to a smaller number of false alarms.

Plagiarism detection has been approached in other works as well. Any kind of data can be plagiarized, including documents, multimedia files or programs (both at source code level or at binary level). We are not interested in verbatim copies, as they are easy to spot by hashing an entire collection of documents and comparing the documents with the same hash. Near-identical copies, where the plagiarizer makes some modifications are harder to detect. Most approaches relied on the concept of n-grams. An n-gram can be considered as a group of n consecutive items from a sequence.

3.1.1 Source code plagiarism detection

The ideas behind Moss, a publicly available solution, are presented in [SWA03]. The system is designed for plagiarism detection in programming assignments. The authors also host a web server where users can upload a collection of source codes and receive a similarity report. Moss also includes a visualization component that shows similar regions in the source code.

The fingerprinting technique used in Moss is called Winnowing and is based on n-grams (called k-grams in the paper). The authors argue against a popular technique, of selecting only the n-grams that are divisible by a number p as a fingerprint. Although this technique reduces the number of fingerprints to $\frac{1}{p}$, there may be large gaps between two selected n-grams. The proposed Winnowing techniques ensures that at least one n-gram is selected from every window of w consecutive n-grams. This property is ensured by selecting the minimum value from each window of w n-grams. In case more than one n-gram with the minimum value is encountered, the rightmost value is selected.

In this thesis we also use n-grams for fingerprinting but instead of working directly on the source code, we use binary code. The compiler will perform the basic normalization on the analyzed code, like eliminating variable and function names or rewriting slightly different functions. These normalizations will make the fingerprints more robust and ensure a higher detection rate.

3.1.2 Mobile applications plagiarism

Mobile applications plagiarism detection is also an interesting research topic, as we identified several recent papers dealing with this subject.

In [PNNRZ12], the focus was on cases where the attacker added malicious code to existing applications. The authors showed that 29.4% applications from a collection of 158000 are more likely to be plagiarized because they already have the permissions that an attacker needs. For deciding if two programs are similar or not, three schemes were proposed: *Symbol-Coverage*, *AST-Distance* and *AST-Coverage*. The first one relies on the symbol names extracted from the application and works only if no form of obfuscation has been involved. The other two schemes are based on *Abstract Syntax Trees*, a model that considers for each method only the number of arguments and the other invoked methods. Based on these trees, feature vectors can be built for the entire application or at method-level. *AST-Distance* finds plagiarism cases in a collection by comparing the feature vector extracted from a given application with the feature vectors of the other applications in the collection, while *AST-Coverage* matches the method-level feature vectors of two applications and searches for the maximum coverage. All these schemes work by comparing a given application with all the others in the collection. The tested collection was small, containing only 7600 samples.

The authors in [CGC12] attempted to eliminate the pairwise similarity computations by clustering the applications in the first phase. The clustering is performed based on the application's attributes. Only the applications situated in the same cluster will then be pairwise verified for plagiarism by comparing the Program Dependence Graphs [FOW87]. The graphs are compared in two phases. The first phase tries to filter out the pairs that are too different while the second one performs the more costly operation of finding subgraph isomorphisms. The paper also addresses the problem of eliminating the library code and the author identification so that two programs belonging to the same author won't be flagged as plagiarism. The framework was tested on a collection of 75000 free Android applications and managed to check for plagiarism an average of 0.71 application pairs per minute.

Juxtapp is another system proposed by [HHW$^+$13] that detects vulnerable code reuse, malicious samples and piracy (plagiarism) cases. This system uses OpCode n-grams like ours but doesn't split the code into methods. Since the number of n-grams extracted from an application is quite large, a feature hash is produced, represented as a bit vector. Two such bit vectors can be compared using the Jaccard distance. Finding all cases of similar applications is still a quadratic problem, as each pair must be checked. The system was tested on a collection of 58000 applications from the Android official market and from the Anzhi third party market.

The three systems above employ various methods for detecting if two applications are similar but are working on small samples collections (less than 100000). The focus of our system is scalability, as it manages to deal with a collection bigger than one million samples, while still correctly identifying plagiarism cases. Also, the work described in this thesis presents some practical considerations when dealing with the applications ecosystem. One such consideration is that in many Android applications, the quantity of library code exceeds the quantity of application-specific code, making library code identification a top priority.

3.1.3 Training classifiers for malware detection

The authors of [SMF$^+$12a] used features extracted from programs code in order to build classifiers that discriminate between clean and malicious samples.

The paper compares bytes n-grams with OpCode n-grams. The first type of features is obtained by extracting n-grams directly from the binary file, without further processing. The OpCode n-grams are based on disassembled instructions and take into consideration only the operation, not the operands. The second approach is proven to be more robust because the instructions are less likely to change.

The test collection contained more than 30000 files, for which 8 different classifiers were trained. The classifiers achieved a True Positives Rate above 95% and a False Positive Rate below 10%. The imbalance problem was also discussed, because in real-life scenarios the number of malicious files is smaller than the number of clean samples.

Machine learning algorithms such as Hidden Markov Models and linear classifiers were also used in [CBG14] for detecting JavaScript malware. This kind of malicious scripts are responsible for spreading further binary executables and relay on heavy obfuscation. By examining a large dataset of malicious scripts, the machine learning algorithms found new patterns for detecting this type of malware.

A comparative study between various machine learning techniques for malware classification was presented in [VCGL15]. The study used real-world data, comprising of 2 million clean files and 200,000 infected files, that is considered "a realistic quantitative mixture". The first issue identified was the rate of false positives. If a benign file is wrongly labeled as malicious, the file will be blocked or even deleted by anti-virus engines, leading to data losses or work disruption.

The techniques employed included One Side Class learning (training a classifier that will correctly classify all the records from one class), various classifiers like perceptrons or decision trees and ensemble techniques. The experimental results showed that the best classifier can achieve a detection rate of 76.46%, while maintaining the false positive ratio around 0.5% and the training time below 7 hours.

3.2 Cluster Analysis on Massive Collections

Clustering large collections is a problem well studied in the literature as it has been noticed that classical algorithms like SLINK [Sib73] render poor performance on big datasets. Since SLINK has been proved to compute single linkage in optimal time and is still quadratic, improvements that achieved higher performance lost some of the algorithm's accuracy. In malware analysis, avoiding pairwise distance computations between millions or even billions of samples may worth misplacing a few samples in different clusters.

3.2.1 Clustering through hashing techniques

Identical samples can be identified through classical hashes, as stated in subsection 2.1.1. Unfortunately, near-identical or similar samples will not have the same hash value if we are using regular hashes.

A simple hash function that would output the same value for similar items and different values for dissimilar ones is impossible to design in Euclidean spaces. Let h be such a hash function and two samples A and B, such that $d(A, B) = \frac{3}{2}\theta$, where θ is the distance threshold. Since the distance between A and B is greater than the threshold, they should have different hash values. In any Euclidean space, there exists a point C situated at half the distance between A and B. This means that $d(A, C) = d(B, C) = \frac{3}{4}\theta < \theta$. The last equation implies that $h(A) = h(C)$ and $h(B) = h(C)$, which contradicts the fact that A and B have different hash values.

The clustering problem has been approached by using several hash functions, each of them having a high probability of outputting the same value for similar items.

The first approach that involved MinHash functions was done by Broder in [Bro97] who used the terms *resemblance* and *containment* in order to describe relationships between documents. The *resemblance*, expressed as a Jaccard similarity between two sets was shown to be related with the concept of MinHashes. The author also realized that for a good estimation for the similarity, several MinHash values are required. However, the approach in [Bro97] was slightly different than ours: instead of considering the minimum values according to different permutations, the author considered the smallest s elements in the set (s was a fixed parameter).

A generalized approach called locality-sensitive hashing was introduced by Indyk and Motwani [IM98] and was used to retrieve similar items from a large collection. The novel idea was to put together several independent hash functions, each of them being more likely to give the same value for similar items than for dissimilar ones. Combining these functions properly will lead to similar items identification with a very high probability.

Koga addressed the problem of clustering in [KIW07], by employing the technique of locality-sensitive hashing. He showed that the complexity of the clustering algorithm can drop from $O(n^2)$ in the case of SLINK algorithm [Sib73] to $O(nB)$, where B is "the maximum number of points going into a single hash entry". The author argues that the value B is much smaller compared to the number of samples n. Still, depending on the dataset this value can be large enough to cause performance issue. Our proposed algorithm will try to address this issue by further dividing the points going into the same hash entry, if their set is too big.

Previous attempts to optimize locality-sensitive hashing were also done. In 2012, Slaney [SLH12] presented a method for selecting the optimal parameters for the locality-sensitive hashing, in order to improve the speed of search and also maintain a good recall rate. The author's approach was based on the data distribution. By sampling the dataset and inferring a probability distribution for the distance between samples, Slaney managed to build a model that selects the best parameters.

3.3 Chapter Conclusions

The continuous research for identifying binary similarity and for performing cluster analysis shows that static analysis is a domain of interest in malware research, plagiarism detection and information retrieval. The existing solutions contain interesting ideas that provide some of the basis for the research in this thesis. However, some solutions were only tested on small datasets and lack the ability to scale, while other solutions are not adapted to deal with the obfuscation employed by malware authors.

Some early approaches attempted a general solution for finding near-duplicates. The research in this thesis focuses on binary programs and proposes solutions for specific files formats but can easily be extended to cover any other formats. By taking into account the particularities of binary programs, the quality of the results improves. Source code plagiarism detection can benefit from the methods we propose for dealing with binary programs. The fingerprinting techniques from the state of the art detection systems will not completely overcome the syntax differences in plagiarized coding assignments. Some advanced techniques for comparing Android applications were also proposed in the recent years but the presented systems worked on collections of 100,000 samples or less. This thesis proposes a system that dealt with more than 1,000,000 applications.

The literature review also found some interesting techniques for clustering large collections, based on hashing techniques. Some general guidelines for employing these technique are given. This thesis completes the guidelines by proposing formal solutions to select the optimal parameters for fast clustering algorithms.

Chapter 4

Measuring Software Similarity

4.1 Code Analysis for Features Extraction

A program's essence resides in it's code. Most programs are distributed in binary format, as the source code is usually unavailable. This section describes the process of analyzing binary programs in order to build an abstract model of their code. Such abstract models will be used in later sections from this chapter to compute software similarity. There are also programming languages that don't get compiled (usually scripting languages like JavaScript, VBScript or Python). Although this section doesn't show how to extract abstract models from these type of programs, some models extracted by other works like [VGB12] can also be used as features for the software similarity task.

4.1.1 Extracting native code from Windows programs

Microsoft Windows uses the Portable Executable file format [Pie02] for binary programs, most of them running on the Intel x86 [Int13] platform.

The main parts of a Windows portable executable are the headers and the sections. The headers contains informations about the file, including the characteristics for each section. For code extraction, we are only interested in the sections marked by the compiler as *code containers*. The header format also contains the offset and the size of each section in file. These executable sections will then be treated as raw code buffers.

Having the raw code buffers, the next step is to disassemble them in order to obtain the sequence of OpCodes. The code buffers can also contain data or invalid instructions, besides the binary code, but the good news is that the Intel x86 instruction set has the property of *self-repairing disassembly* [LD03]. This property ensures that even if some invalid instructions occur in the disassembly flow, the flow will "eventually re-synchronize with the actual instruction stream".

The actual disassembly process consists in decoding the instruction at the current position in the buffer, extract any necessary information from it, then advance to the next instruction. Intel x86 instructions have a variable length, so the only way to advance to the next instruction is to decode the current instruction and determine it's length. The diStorm3 library [Dab13] offers support for this task, as it offers a structured-output interface that outputs each disassembled instruction as a structure, instead of a textual representation.

The instructions from Intel x86 architecture [Int13] can contain up to four prefixes, an operation code (OpCode), a ModR/M field, a SIB (Scale, Index, Base) field, a displacement and an immediate constant (some of the components are not present in all the instructions).

We considered the most important part of the instruction to be the OpCode, that defines the operation performed by the instruction. This OpCode can tell if we are dealing with a `MOV`, a `PUSH` or an `ADD` instruction, for example.

In order to obtained the desired OpCode sequence, we will proceed as in Algorithm 1.

Algorithm 1 EXTRACT-OPCODE-SEQUENCE(S)

```
Require: A binary code buffer $buf$
Ensure: A sequence of OpCodes $S$
 1: $pos \leftarrow 0$
 2: $S \leftarrow$ ""
 3: while $pos < |buf|$ do
 4:    $crtInstr \leftarrow$ DECODE-INSTRUCTION($buf, pos$)
 5:    if IS-VALID-INSTRUCTION($crtInstr$) then
 6:       $S \leftarrow S + crtInstr.opCode$
 7:       $pos \leftarrow pos + crtInstr.length$
 8:    else
 9:       $pos \leftarrow pos + 1$
10:    end if
11: end while
12: return $S$
```

The current position is initialized at the beginning of the buffer. While it doesn't exceed the buffer length, we attempt to decode the instruction that begins at that position. If the current instruction is valid, we will add it's OpCode to the sequence S. The current position then advances with the instruction's length, since Intel x86 instructions have a variable length. If there's an invalid instruction, we simply advance to the next byte.

For practical reasons, some of these OpCodes should be ignored. For instance, an obfuscation tool can add `NOP` instructions in a given code buffer, to obtain a different version of it. `NOP` instructions can also be added by compilers for alignment purposes. Other instruction like `MOV` or `PUSH` are too common (according to [Bil07], these two instructions together form more than 40% of the code). Some other OpCodes, although different, should be treated as they were equal. For example, conditional jumps like `JZ` (jump if zero) and `JNZ` (jump if not zero) are easy to interchange.

To formalize things, we will introduce a function tf, that transforms OpCodes into other symbols to work with. We will denote by $\mathcal{O}$, the set of all possible OpCodes found in x86 instructions. The transform function tf defined in Equation 4.1 transforms a given OpCode into a symbol from the set Σ or into the empty symbol ϵ.

$$tf : \mathcal{O} \to \Sigma \cup \{\epsilon\} \tag{4.1}$$

The statements made above, about certain OpCodes can be expressed as constrains on the function tf:

$$\begin{aligned} tf(\texttt{NOP}) &= \epsilon \\ tf(\texttt{MOV}) &= \epsilon \\ tf(\texttt{PUSH}) &= \epsilon \\ tf(\texttt{JZ}) &= tf(\texttt{JNZ}) \end{aligned}$$

We will define as $\Sigma^\star$, the set of all strings with elements from Σ. In the code analysis process, we will work with strings from $\Sigma^\star$, instead of strings from $\mathcal{O}^\star$.

4.1.2 Working with source code

Although source code is generally unavailable, there are some tasks that require source code analysis. One such task may be checking students homework for plagiarism. In order to apply the same analysis techniques, we need to set a common ground between source code and binary code. Since it is easier to compile sources files and obtain binary executables, than decompiling the former back into source code, we have chosen the binary executables as a common format.

For mass compiling several programs from source code, the command-line version of the Visual C++ compiler [Cor13] was used to produce Windows portable executables [Pie02]. We wanted the following principles to be enforced in the compilation process:

- The sources should be compiled on release mode. An executable compiled on debug mode contains a lot of debugging information embedded, that is not relevant for our goal.
- There should be no compiler optimizations. If the compiler was allowed to perform optimizations, a piece of code written differently by two students might end up in the same form.
- No library code should be included in the result. The external libraries used (like *msvcrt*, that contains the *printf* function) should be linked dynamically. Although we will show how to ignore the library code while computing similarities, it is better to start without it.

To follow the principles above while compiling, we will use the following command line arguments for `cl.exe`:

```
cl.exe /EHsc /Od /MD source.cpp
```

The `/Od` switch tells the compiler to disable optimizations, while `/MD` “causes the application to use multithread- and DLL-specific version of the run-time library”. This means that external libraries won't be included in the compiled binary, only “a layer of code that allows the linker to resolve external references”.

After obtaining the binary executable format from the source code, we can treat it in the same way as the regular binaries. One may argue that some students can submit uncompilable source code in order to defeat such a plagiarism checking system. To prevent this, an instructor should not accept source code that doesn't compile.

4.1.3 Extracting CIL code from .NET binaries

Since the release of Windows Vista, the .NET framework is installed by default on any computer that runs this operating system [Wan06] or later versions (Windows 7, 8). The .NET framework will also run on Windows Phone [Bra12]. This opened a gate to malware creators, as they are now able to create new, portable malware in an easier way. For this reason, we have decided to focus on .NET malware.

.NET programs are Portable Executables [Pie02], having a file format similar to other Windows programs. Instead of x86 assembly, Common Intermediate Language (CIL) is used. The complete Common Language Infrastructure (CLI) of .NET programs is described in [Int12]. The document shows how "applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments".

More precisely, CLI "provides a specification for executable code and the execution environment". The executable code is divided into methods that consist of variable length CIL instructions, just like native code that runs on x86 processors. However, CIL instructions are different from the native code. For instance, they do not use registers, they work with the stack. [Int12] divides the CIL instructions in two categories: *base instructions* and *object model instructions*.

Most of the instructions from the first category have equivalents in x86 native instructions. We have the following subcategories:

- instructions that move data around: Since CLI works with the stack, these instructions are somehow limited. Examples of such instructions are `ldc` that loads a constant on the stack, `pop` that removes the top of the stack or `ldarg` that loads the specified argument on the stack.

- arithmetic and logic instructions: `add`, `div`, `or`, `and`, `xor`, ... These instructions are very similar to the x86 equivalents, except that they read their arguments from the stack instead of registers and store the result on the stack. The comparation instructions will also be classified here. Instead of instructions that set the flags (`cmp` and `test` in x86), we have different instructions for different kinds of comparisons. Some examples are: `ceq` - compare equal, `cgt` - compare greater than. The result of the comparison (0 or 1) will be pushed on the stack.

- instructions that modify the control flow: such instructions are method calls (`call`), method jumps (`jmp` - just like calls, but they never return) or branches. A branch is an instruction that may transfer the control to another instruction from the same method, different from the instruction that follows. The address of the target instruction is computed by adding the branch's argument to the address of the instruction that follows the current one. These branches are unconditional (`br`) or conditional (`ble` - branch less or equal, `blt` - branch less than, `brtrue` - branch when true). A CIL branch is equivalent to a relative jump from the x86 instruction set.

Some of these instructions have multiple versions, depending on the argument type. If the instruction name is suffixed by an `.s`, we have the *small* version, meaning that the instruction's operand has only one byte instead of four (for example, the offset for a branch can fit in a single byte). The `.un` suffix specifies that the operand must be regarded as an unsigned number. The two suffixes can be combined. Note that the suffixes appear only in the instruction's name. Binary, they are encoded as two different operations.

Object model instructions are special instructions that deal with the object oriented part of the CLI. Examples of such instructions are `newobj` that creates new object or `throw` that throws an exception.

If we look at the binary encoding, an instruction can start with a prefix and has an OpCode (Operation Code) and 0 or more operands. The OpCode is the portion of a CIL instruction that specifies the operation that must be performed. A CIL instruction also contains the instruction's operands, but they will be disregarded, as they are very easy to modify.

In order to get access to the code buffers, we need to parse the .NET structures, as detailed in [Pis08]. The first relevant information about these structures are found in the .NET Directory, a data directory from the Portable Executable format [Pie02], that replaces the old COM Directory. This directory contains an `IMAGE_COR20_HEADER` structure, also known as the *CLI header*.

This structure contains several sections, but we will focus here only on the Metadata Section. Here, there are usually 5 streams:

- `#Strings` - An array of ASCII strings. The strings in this stream are referenced by Metadata Tables.
- `#US` - Array of unicode strings. The name stands for User Strings, and these strings are referenced directly by code instructions (ldstr).
- `#Blob` - Contains data referenced by Metadata Tables.
- `#GUID` - Contains 128 bits long unique identifiers. Also referenced in Metadata Tables.
- `#∼` - The most important stream. It contains the Metadata Tables.

The Metadata Tables (or the `#∼`) stream contains a set of variable-length tables, found in Table 4.1, from [Pis08].

The table that we need for extracting the code buffers is the 6th one, the MethodDef table. There, each row represents a method in a specific class. Among others, the row contains the Relative Virtual Address, where the method's code is located inside the file. Since the tables above have a variable length, it is necessary to parse them in order to reach the required table.

After reaching the MethodDef table, we are able to disassemble the methods code. The instruction set is detailed in [Int12]. Each instruction has an OpCode represented on 1 or 2 bytes (in the second case, the first byte is always `0xFE` for the current version of CLI). After the OpCode, CIL instructions "can be followed by zero or more operand bytes" [Int12].

In Figure 4.1, we have a listing of some CIL (Common Intermediate Language) code disassembly. On each line we have one instruction with the following information:

- position in file, specified as RVA (Relative Virtual Address)
- instruction's bytes, between square brackets
- operation's name

Table 4.1: The Metadata tables [Pis08]

00 - Module	01 - TypeRef
02 - TypeDef	04 - Field
06 - MethodDef	08 - Param
09 - InterfaceImpl	10 - MemberRef
11 - Constant	12 - CustomAttribute
13 - FieldMarshal	14 - DeclSecurity
15 - ClassLayout	16 - FieldLayout
17 - StandAloneSig	18 - EventMap
20 - Event	21 - PropertyMap
23 - Property	24 - MethodSemantics
25 - MethodImpl	26 - ModuleRef
27 - TypeSpec	28 - ImplMap
29 - FieldRVA	32 - Assembly
33 - AssemblyProcessor	34 - AssemblyOS
35 - AssemblyRef	36 - AssemblyRefProcessor
37 - AssemblyRefOS	38 - File
39 - ExportedType	40 - ManifestResource
41 - NestedClass	42 - GenericParam
44 - GenericParamConstraint	

- optionally, one or more operands

The first instruction in the listing is a `nop` instruction. It starts at the address `0x254C` and has a single byte: `0x00`. The second instruction is a `call`, that starts at the address `0x254D` and occupies 5 bytes. The first byte (`0x28`) represents the OpCode for the `call` instruction, while the remaining 4 bytes form the instruction's operand `0x0A00000E` (the value is written in little endian). For each instruction we can determine the number of bytes from the OpCode (with one exception - `switch`, that has a variable size).

Many malware authors apply obfuscation techniques to the code, such that it behaves identically, but the sequence of operation is changed. We would like to perform such a transformation to the code buffers, that several versions of the same method, obtained through obfuscation will be as similar as possible.

To do this, we will perform two steps: the first one is to eliminate the unreachable code, and the second one is to normalize the OpCodes.

4.1.3.1 Eliminating the Unreachable Code

Some code obfuscation tools will add random instruction to the existent code, in such a way that they will never be reached. For example, they can split the instruction sequence, add an unconditional branch instruction and also some garbage bytes after it. Provided that all the code references are fixed, the garbage bytes will never be reached.

We may consider the sequence of instructions: i_1, i_2, i_3, i_4, i_5. An obfuscation tool may transform it into i_1, i_2, b, g_1, g_2, g_3, g_4, i_3, i_4, i_5, where g_1, g_2, g_3, g_4 are random garbage instructions, while b is an unconditional branch instruction that tells the interpreter to skip the length of the garbage code. For example, if each garbage instruction is one byte length, b might be `br.s` 4. `br.s` (*branch small*) is an unconditional branch instruction (equivalent to the x86 `jmp` instruction), that tells the interpreter to jump over the number of bytes specified in the 1-byte argument.

In order to design an algorithm that eliminates the unreachable code, we must take into account all the possible modifications of the instruction flow:

```
=== Method 4: name='mpress._::Main'; RVA=0x0000254C;
              FA=0x0000074C; size=0x9A ===
  = Exception handlers: 000025D6; =
0000254C: [00] nop
0000254D: [28 0E 00 00 0A] call 0x0A00000E
00002552: [12 00] ldloca.s 0x00
00002554: [28 03 00 00 06] call 0x06000003
00002559: [13 06] stloc.s 0x06
0000255B: [11 06] ldloc.s 0x06
0000255D: [2D 16] brtrue.s 0x16
0000255F: [00] nop
00002560: [72 01 00 00 70] ldstr 0x70000001
00002565: [72 23 00 00 70] ldstr 0x70000023
0000256A: [28 0F 00 00 0A] call 0x0A00000F
0000256F: [26] pop
00002570: [15] ldc.i4.m1
00002571: [13 05] stloc.s 0x05
00002573: [2B 6E] br.s 0x6E
00002575: [00] nop
00002576: [06] ldloc.0
00002577: [28 10 00 00 0A] call 0x0A000010
0000257C: [80 01 00 00 04] stsfld 0x04000001
00002581: [7E 01 00 00 04] ldsfld 0x04000001
00002586: [6F 11 00 00 0A] callvirt 0x0A000011
0000258B: [0B] stloc.1
0000258C: [14] ldnull
0000258D: [0D] stloc.3
0000258E: [07] ldloc.1
0000258F: [6F 12 00 00 0A] callvirt 0x0A000012
00002594: [8E] ldlen
```

Figure 4.1: Example of code disassembly

- returning instructions (`call`, `callvirt`, ...): although these instructions modify the flow by calling another method, the flow will return to the next instruction after the called method returns. We can treat these instructions as regular instructions, that do not modify the flow.

- unconditional branches (`br`, `br.s`): these instruction will always add a (positive or negative) value to the instruction pointer. If the argument is 0, they are equivalent to the `nop` instruction.

- conditional branches (`brtrue`, `brfalse`, `breq.s`, ...): the instruction flow might continue normally, or it might be altered. Since we cannot determine statically if the branch condition is met or not, both alternatives must be considered. This means that we will mark as reachable both the code that follows after these instructions and also the code that would be reached by adding the argument to the instruction pointer.

- flow disruptive instructions (`jmp`, `ret`, `throw`, ...): the `jmp` instruction is similar to `call` (it calls jumps to the specified method) but it doesn't return the control flow to the next instruction. `ret` and `throw` will also disrupt the instruction flow by ending the current method or by jumping to an exception treating block. The code following these instructions will not be marked as reachable, unless referenced by other reachable instruction.

The proposed Algorithm 2 will receive a code buffer (*codeBuf*) and will return another buffer (*reach*) of the same size. The returned buffer will contain only 0 and 1 values, for each byte. A value of 1 on the position i in the *reach* buffer means that the instruction on the corresponding position in *codeBuf* can be reached by the instruction flow. A value of 0 means that the corresponding instruction is unreachable.

Algorithm 2 MARK-REACH-CODE(*codeBuf*, *excList*)

Require: A buffer *codeBuf* containing a method's code.
Require: A list of starting positions for the exception handlers *excList*.
Ensure: A buffer *reach* where all the reachable bytes are set.

```
1: for $i = 0 \to |\ codeBuf\ | -1$ do
2:     $reach[i] \leftarrow 0$
3: end for
4: $Q \leftarrow \{\}$
5: ENQUEUE($Q, 0$)
6: for $i = 1 \to |\ excList\ |$ do
7:     ENQUEUE($Q, excList[i]$)
8: end for
9: while $|\ Q\ | > 0$ do
10:     $ip \leftarrow$ DEQUEUE($Q$)
11:     while $ip < |\ codeBuf\ |$ and $reach[ip] = 0$ do
12:         $len \leftarrow$ GET-INSTR-LENGTH($codeBuf, ip$)
13:         for $i = 0 \to len$ do
14:             $reach[ip + i] \leftarrow 1$
15:         end for
16:         $type \leftarrow$ GET-OP-TYPE($codeBuf, ip$)
17:         if $type =$ BRANCH_UNCOND then
18:             $distance \leftarrow$ GET-OP-ARG($codeBuf, ip$)
19:             $ip \leftarrow ip + len + distance$
20:         else if $type =$ BRANCH_COND then
21:             $distance \leftarrow$ GET-OP-ARG($codeBuf, ip$)
22:             ENQUEUE($Q, ip + len + distance$)
23:             $ip \leftarrow ip + len$
24:         else if $type =$ INSTR_DISRUPT then
25:             break
26:         else
27:             $ip \leftarrow ip + len$
28:         end if
29:     end while
30: end while
31: return $reach$
```

This algorithm parses the code, instruction by instruction and marks all parsed instructions as reachable. When a conditional branch is reached (as stated above, both alternatives must be taken into account), one path is taken and the other one is enqueued.

The current parsing will stop either when the end of the buffer or some already parsed code is reached, either when a flow-disruptive instruction is encountered. Then, if the working queue is not empty, a new address is dequeued and the instructions starting there will be processed. The initial queue contains the entry point of the method and the addresses of the exception handlers. These exception handlers are portions of a method's code that are invoked when various exceptions occur. Although the normal instructions flow might not reach them, they must be marked as reachable code, because we can't determine statically if an exception will occur or not.

Lines 1-3 will initialize the *reach* buffer with zeros (we start with no reachable instructions), while lines 5-8 will enqueue all the possible starting points for code (the method's entry point and the exception handlers). From each starting point in the queue, the code will be parsed sequentially and marked as reachable (lines 13-15), until some already marked code or the end of the buffer is reached. When an unconditional branch is reached (lines 17-19), the argument of the command is added to the instruction pointer (*ip*). If the branch is conditional (lines 20-23), the parsing continues with the next instruction, and the possible target is enqueued. A flow disruptive instruction will stop the current parsing and will continue with the next element from the queue.

The function GET-OP-TYPE (get operation type) receives a code buffer and a position inside it and returns the type of the instruction at the given position. The function GET-OP-ARG (get operation argument) receives similar parameters and returns the value of the first argument of the instruction. In case of branch instructions, the branch distance is returned, that is relative to the following instruction.

The running time for MARK-REACH-CODE, on a buffer $codeBuf$ of length n is $O(n)$, because in the worst case scenario, every instruction is reached, and every instruction is processed at most once. Considering the average length for an instruction to be the constant $c \geq 1$, the number of parsed instructions will be $\frac{n}{c}$, and each instruction will be parsed in $O(1)$ so the final complexity will be $O(n)$. The algorithm will always finish because each branch instruction will be reached as most once, so it will get the chance to enqueue a value at most once.

4.1.3.2 OpCodes Normalization

After marking the reachable code, all we have to do is extract the OpCodes from every reachable instruction. Instead of extracting the raw OpCodes, we will perform some transformations, similar to the one applied to Intel x86 instructions.

A tf function similar to the one in Equation 4.1 will be defined for CIL instructions. In this case, $\mathcal{O}$ is the set of all CIL instructions. Some constraints enforced on the tf function will be:

$$tf(\texttt{br 0}) = tf(\texttt{nop}) = \epsilon$$
$$tf(\texttt{brtrue}) = tf(\texttt{brtrue.s}) = tf(\texttt{brfalse})$$

The various scenarios that start from a binary program or a source code and obtain strings from the Σ alphabet are depicted in Figure 4.2.

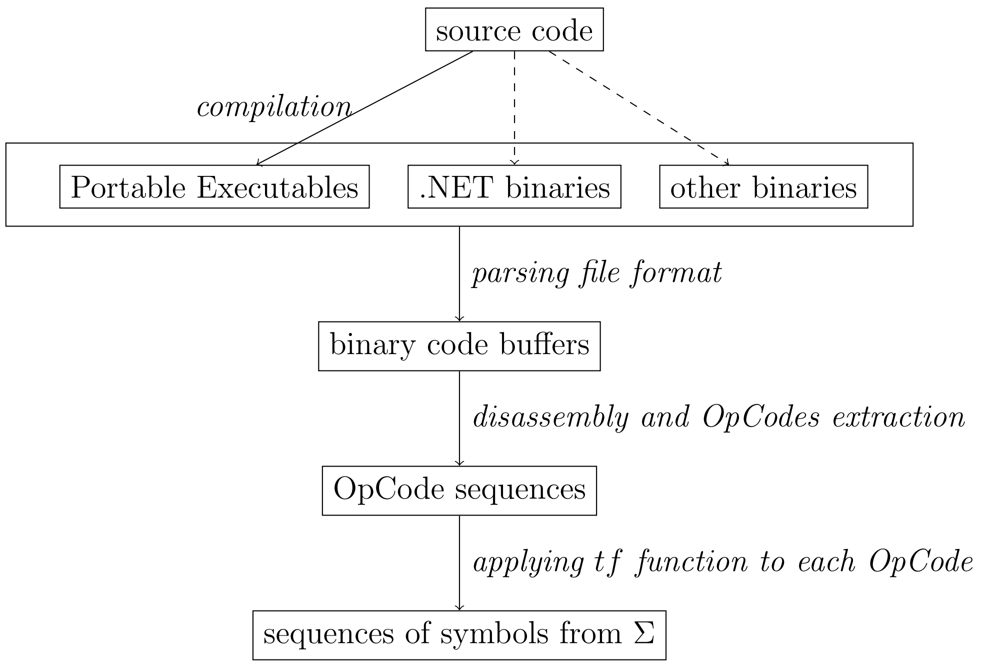


Figure 4.2: The workflow for transforming a binary or source file into a sequence of symbols

4.2 Distance Metrics and Similarity: Set or String Semantics?

In this section we will describe several techniques for computing the distance between binary programs, in order to decide if two programs are similar or not. We will propose several distance metrics, that we will compare with common metrics from the literature. We are interested in two aspects: how well does a certain distance can express the similarity between two programs and how fast can we compute it.

4.2.1 Distance and similarity definition

In the previous section, we showed how to get an abstract model of a binary program as a sequence of operations represented by a string from $\Sigma^\star$. In this sections, we will treat $\Sigma^\star$ as a metric space, by defining distance functions between such strings.

A distance function on $\Sigma^\star$ can be defined as in Equation 4.2: a function taking as inputs two strings from $\Sigma^\star$ and outputting a non-negative real number.

$$d : \Sigma^\star \times \Sigma^\star \to [0, \infty) \tag{4.2}$$

Such a function is considered a metric, if it follows the following properties:

1. non-negativity: $d(X, Y) \geq 0, \forall X, Y \in \Sigma^\star$.
2. identity of indiscernibles: $d(X, Y) = 0 \iff X = Y$.
3. symmetry: $d(X, Y) = d(Y, X), \forall X, Y \in \Sigma^\star$.
4. triangle inequality: $d(X, Y) + d(Y, Z) \geq d(Z, X), \forall X, Y, Z \in \Sigma^\star$.

The distance between two programs gives a measure of dissimilarity between them. The second metric property assures that the distance between two programs is 0 if and only if they are identical. However, the numeric value of the distance can only be used to tell if a pair of items is more similar than another pair. Indeed, if the distance between X and Y is 2 and the distance between X and Z is 10, we know that Y resembles X more than Z does, but it is hard to tell if none, only Y or both Y and Z have a significant part of the code common with X. In order to make such statements, we will employ similarity functions, which also use two elements from $\Sigma^\star$ as inputs, but output a value in the $[0, 1]$ interval (Equation 4.3).

$$sim : \Sigma^\star \times \Sigma^\star \to [0, 1] \tag{4.3}$$

The advantage of similarity functions is that we can use them to express distances in a manner that is easier to understand by humans. A sentence like "X is 70% similar to Y" is easier to comprehend than "the distance between X and Y is 14.3". A value of 1 (or 100%) means perfect similarity, while a value of 0 means no similarity at all.

If the distance defined in Equation 4.2 takes values only in a finite interval $[0, MAX]$, then a distance functions can easily be converted into a similarity function:

$$sim(X, Y) = \frac{MAX - d(X, Y)}{MAX}$$

In the particular case where $MAX = 1$ (e.g. the Jaccard distance takes values between 0 and 1), we have $sim(X, Y) = 1 - d(X, Y)$.

4.2.2 Proposed metrics based on string semantics

4.2.2.1 Descriptional Entropy

The first similarity function will be based on the concept of Descriptional Entropy from [PG12].

In order to explain this concept, we will first need to recall Shannon's definition of entropy from [SW48]:

Let Σ be an alphabet, and let S be a string from $\Sigma^\star$. For each $s \in \Sigma$, $p_S(s)$ will be the computed probability that a random symbol from S is s.

The entropy of the string S is:

$$E(S) = -\sum_{s\in\Sigma} p_S(s) \cdot \log p_S(s) \tag{4.4}$$

Unfortunately, the entropy, by itself, is not able to differentiate two strings very well. By the definition above, the entropy of "*aaabbb*" and "*ababab*" is the same, even if they appear quite dissimilar.

The authors in [PG12] introduced a new concept of entropy, called the Descriptional Entropy. It is defined as above, but instead of considering each symbol from S, each substring is considered, with it's probability to appear in S. If for each unique substring we associate an index i from 1 to N, and a probability of appearance p_i, the Descriptional Entropy is defined as follows:

$$DE(S) = -\sum_{i=1}^{N} p_i \cdot \log p_i \tag{4.5}$$

Below, we have some examples of strings, with their Descriptional Entropy.

- For $S = "aaabbb"$, $DE(S) = 2.59861$.
- For $S = "ababab"$, $DE(S) = 2.30963$.
- For $S = "abbaab"$, $DE(S) = 2.66462$.

We can observe that $DE(S)$ varies with the string's complexity.

While computing Descriptional Entropy on the code buffers, we will make a small modification to the method described in [PG12]. The substring sizes will be limited to a constant `DE_SS_SIZE` that has been chosen empirically. This approach has two advantages. First, the method will be faster as it only has to deal with smaller substrings. Secondly, if some code portions are permuted, the computed value will remain the same, as most of the substrings will not change.

With this Descriptional entropy, we can now compute the similarity of two files, with buffers S_1 and S_2, with the following formula:

$$DEsim(S_1, S_2) = 1 - \frac{\mid DE(S_1) - DE(S_2) \mid}{\max(DE(S_1), DE(S_2))} \tag{4.6}$$

$DEsim(S_1, S_2)$ will be very close to 1 when $DE(S_1)$ is close to $DE(S_2)$ and will decrease to a minimum value of 0, when they are too dissimilar.

By computing the Descriptional Entropy, after we have extracted the sequence of symbols, we have manage to project an entire file on a single dimension. Although we cannot expect surprising results by representing a file as a single real number, we have included this method, to see how well it performs. In some cases, it can be used for pre-filtering similarity candidates in order to decide if it worths applying a more precise but costly similarity function.

4.2.2.2 Normalized Compression Distance

The concept of Normalized Compression Distance is related to the Kolmogorov complexity [Kol68].

In algorithmic information theory, the Kolmogorov complexity of an object is a measure of the computational resources needed to specify the object. Unfortunately, the Kolmogorov complexity of a buffer is not computable, so we will approximate it.

Let us consider a buffer S, that we need to compress. Various compression algorithms will output various compressions, but the output length will depend every time on the buffer's complexity.

Usually, if we concatenate a buffer with itself, the compression length of the new buffer will be very close to the compression length of the original one. Also, if buffer S_1 is very similar to buffer S_2, the compression length of the concatenation of these buffers will be close to the compression length of S_1.

Based on these observations, we will define the following metric, between two buffers S_1 and S_2:

$$NCDsim(S_1, S_2) = 1 - \frac{CL(S_1 \mid\mid S_2) - \min(CL(S_1), CL(S_2))}{\max(CL(S_1), CL(S_2))} \tag{4.7}$$

where $||$ is the concatenation operator and CL is the Compression Length function, that computes the length of the compressed buffer, given an input buffer.

For the Compression Length we have used the *compress* function from the zlib library [Adl13]. This library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. The compression is fast, and it is easy to integrate in the project.

4.2.2.3 The deq distance

Visually, we can consider two sequences of operations to be similar if we manage to align them in such a way that the correspondence between identical blocks of code can be seen. Such an alignment can be performed by computing the longest common subsequence between the two strings. A distance based on this approach already exists and it is called *longest subsequence distance* [NW70, Nav01]. Unfortunately, the best known algorithm for computing this distance is quadratic.

The *deq distance* that we will propose in this subsection uses the longest common substring instead of the longest common subsequence. Empirically, this means that we will consider two pieces of code to be similar if they share a large chunk of consecutive instructions. The advantage here is that the longest common substring can be computed in linear time. The two distances will also be tested with a clustering algorithm, in order to see if they produce similar results.

In what follows, a group of consecutive symbols from a given string will be called a substring. If the symbols are not necessarily consecutive (but they are still in the same order as in the original string) we have a subsequence.

The *deq distance* is a special case of edit distance, meaning that it measures the number of operation required to transform a given string into another. Levenshtein distance [Lev66] is the most common type of edit distance and it allows the following operations to be performed on a string:

- *insertion*: a character can be inserted anywhere in the given string
- *deletion*: any character from the given string can be deleted
- *replacement*: any character from the given string can be replaced by another character

The *longest subsequence distance* [NW70, Nav01] allows only the *insertion* and *deletion* operations. Other distances introduce the *transposition* (swapping of two characters) as an operation.

Let Σ be a set of symbols or characters. Such a set will be called an *alphabet*. The closure of the set Σ, denoted by $\Sigma^\star$ is the set of all strings that can be obtained by concatenating symbols from Σ. The string distances will be defined on this set, as functions that receive two points from $\Sigma^\star$ and output a positive real number.

The *longest subsequence distance* is formally defined through the concept of *longest common subsequence*: the longest sequence of (not necessarily consecutive) characters that appear in two strings ($LCS : \Sigma^\star \times \Sigma^\star \to \Sigma^\star$). Given the LCS function, this distance is defined in Equation 4.8.

$$\begin{gathered} lsd : \Sigma^\star \times \Sigma^\star \to [0, \infty) \\ lsd(S_1, S_2) = |S_1| + |S_2| - 2|LCS(S_1, S_2)| \end{gathered} \tag{4.8}$$

The algorithm for determining this distance (which is basically reduced to computing LCS) is based on dynamic programming [Nav01] and has a time complexity of

$O(m_1 \times m_2)$, where m_1 and m_2 are the lengths of the two strings, or $O(m^2)$, if m is the average string length.

Our proposed *deq distance* allows only *insertions* and *deletions* at the beginning or at the end of the string (like in a double-ended queue). This differs from the previous distance, where *insertions* and *deletions* could be performed anywhere. To transform a string S_1 into S_2, one would delete characters from the beginning and from the end of S_1, until a substring S' remains, that is also a substring for S_2. From this point, new characters are inserted at the beginning and at the end of S', in order to obtain S_2. The smallest number of operations is performed if S' is the longest common substring between S_1 and S_2. The longest common substring is defined in a similar manner to the longest common subsequence, but we add the restriction that the characters are consecutive in the two strings.

We will denote the function that computes the longest common substring as lcs (not to be confused with LCS - the function that computes the longest common subsequence).

Now we can define the *deq distance* in the following way:

$$\begin{gathered} d : \Sigma^\star \times \Sigma^\star \to [0, \infty) \\ d(S_1, S_2) = |S_1| + |S_2| - 2|lcs(S_1, S_2)| \end{gathered} \tag{4.9}$$

Theorem 1. *The deq distance d is a metric on* $\Sigma^\star$.

Proof. Given two strings S and T, we will denote by $S \subseteq T$ the fact that S is a substring of T.

To prove that d is a metric, we have to prove that the metric properties hold:

1. *non-negativity:* $d(S_1, S_2) \geq 0$.

   Since $lcs(S_1, S_2) \subseteq S_1$ and $lcs(S_1, S_2) \subseteq S_2$, we have that:

   $$\begin{aligned} |lcs(S_1, S_2)| &\leq |S_1| \\ |lcs(S_1, S_2)| &\leq |S_2| \end{aligned}$$

   By summing the inequations above, we get:

   $$\begin{gathered} 2|lcs(S_1, S_2)| \leq |S_1| + |S_2| \\ \Rightarrow 0 \leq |S_1| + |S_2| - 2|lcs(S_1, S_2)| \\ \Rightarrow 0 \leq d(S_1, S_2) \end{gathered}$$

2. *identity of indiscernibles*:
   $d(S_1, S_2) = 0 \iff S_1 = S_2$.

   If $S_1 = S_2 = S$, then $lcs(S_1, S_2) = S$.

   $$\begin{aligned} \Rightarrow d(S_1, S_2) &= |S_1| + |S_2| - 2|lcs(S_1, S_2)| \\ &= |S| + |S| - 2|S| \\ &= 0 \end{aligned}$$

   If $d(S_1, S_2) = 0$, then $|S_1| + |S_2| = 2|lcs(S_1, S_2)|$. Since $|lcs(S_1, S_2)| \leq |S_1|$ and $|lcs(S_1, S_2)| \leq |S_2|$, in order to have equality, we have $|lcs(S_1, S_2)| = |S_1| = |S_2|$.

From $|lcs(S_1, S_2)| = |S_1|$ and $lcs(S_1, S_2) \subseteq S_1$, we obtain $S_1 = lcs(S_1, S_2)$. Similarly, $S_2 = lcs(S_1, S_2)$, so $S_1 = S_2$.

3. *symmetry*: $d(S_1, S_2) = d(S_2, S_1)$.

   Because $lcs(S_1, S_2) = lcs(S_2, S_1)$,

$$|S_1| + |S_2| - 2|lcs(S_1, S_2)| = \\ |S_2| + |S_1| - 2|lcs(S_2, S_1)|$$

   $\Rightarrow d(S_1, S_2) = d(S_2, S_1)$.

4. *triangle inequality*:
   $d(S_1, S_2) + d(S_2, S_3) \geq d(S_1, S_3)$.

   The inequality above can be expanded to:

$$|S_1| + |S_2| - 2|lcs(S_1, S_2)| + \\ |S_2| + |S_3| - 2|lcs(S_2, S_3)| \geq \\ |S_1| + |S_3| - 2|lcs(S_1, S_3)|$$

   After reducing the terms, we obtain:

$$|S_2| + |lcs(S_1, S_3)| \geq |lcs(S_1, S_2)| + |lcs(S_2, S_3)| \tag{4.10}$$

   Since $lcs(S_1, S_2) \subseteq S_2$ and $lcs(S_2, S_3) \subseteq S_2$, we will consider the three cases in Figure 4.3, depending on their relative position. In fact there are six cases but $lcs(S_1, S_2)$ and $lcs(S_2, S_3)$ can be swapped without loss of generality.

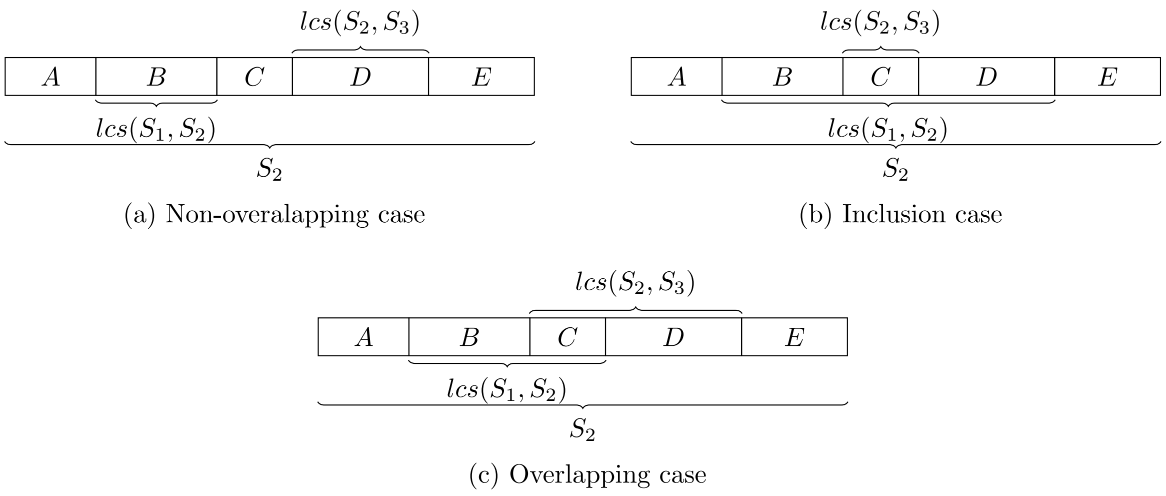


Figure 4.3: Composition of the string S_2

In all the cases, the string S_2 can be written as a concatenation of 5 substrings: $S_2 = ABCDE$, so $|S_2| = |A| + |B| + |C| + |D| + |E|$. Any of those strings can be empty.

If the two common substrings don't overlap (Figure 4.3a), we have $lcs(S_1, S_2) = B$ and $lcs(S_2, S_3) = D$. Since $|S_2| \geq |B| + |D|$, we also have that $|S_2| + |lcs(S_1, S_2)| \geq |B| + |D|$ and we obtain the inequality (4.10).

In the other two cases, the two common substrings overlap. In Figure 4.3b, $lcs(S_1, S_2) = BCD$ and $lcs(S_2, S_3) = C$ so $|lcs(S_1, S_2)| + |lcs(S_2, S_3)| = |B| + 2|C| + |D|$. In Figure 4.3c, $lcs(S_1, S_2) = BC$ and $lcs(S_2, S_3) = CD$ so we also have $|lcs(S_1, S_2)| + |lcs(S_2, S_3)| = |B| + 2|C| + |D|$. In both cases $C \subseteq lcs(S_1, S_2)$ so $C \subseteq S_1$ and $C \subseteq lcs(S_2, S_3)$ so $C \subseteq S_3$. This means that the longest common substring of S_1 and S_3 must be at least as long as C. The inequality (4.10) can be obtained as follows:

$$\begin{aligned}
&|S_2| + |lcs(S_1, S_3)| \\
&\quad \geq |S_2| + |C| \\
&\quad \geq |B| + |C| + |D| + |C| \\
&\quad = |B| + 2|C| + |D| \\
&\quad = |lcs(S_1, S_2)| + |lcs(S_2, S_3)|
\end{aligned}$$

$\square$

The deq similarity The longest common substring and the *deq distance* can give a valid measure of how similar are two strings. However, the numbers alone are not enough to compare two pairs of strings, in order to see which pair has a higher similarity. For instance, if we have a pair of strings of length 20 whose distance is 15, we can say it has a higher similarity than a pair of strings of length 100 whose distance is 20, despite the fact that the first pair has a smaller distance between the strings. Instead, we will use a similarity measure that is related to the distance but outputs values only in the $[0, 1]$ interval. When the similarity between two strings is 0, it means that they have nothing in common, while a similarity of 1 denotes a perfect match.

We will define the *deq similarity* as 1 minus the ratio between the *deq distance* and the sum of the string lengths, as in Equation 4.11:

$$\begin{gathered}
s : \Sigma^\star \times \Sigma^\star \to [0, 1] \\
s(S_1, S_2) = 1 - \frac{d(S_1, S_2)}{|S_1| + |S_2|} = \frac{2|lcs(S_1, S_2)|}{|S_1| + |S_2|}
\end{gathered} \tag{4.11}$$

Indeed, d being a metric, $d(S_1, S_2) = 0 \iff S_1 = S_2$, so the similarity is 1 (or 100%) iff the two strings are identical. $s(S_1, S_2)$ can be 0 only if the two strings have no common characters, but takes values close to 0 for very dissimilar strings.

Computing the distance The *deq distance*, as defined in Equation 4.9 and the *deq similarity* from Equation 4.11 are both based on the longest common substring problem.

One method to solve it is to use dynamic programming [ZCM07], obtaining the complexity $O(m_1 \cdot m_2)$ (m_1 and m_2 are the lengths of the given strings). A faster approach is based on the Suffix Tree data structure that was introduced by Winer in 1973 [Wei73]. The Suffix Tree construction was optimized in 1995 by Ukkonen [Ukk95], that managed to compute a Suffix Tree for a string of length m in $O(m)$ time and a Generalized Suffix

Tree from n strings of lengths $m_1, m_2, \ldots, m_n$ in $O(m_1 + m_2 + \ldots + m_n)$ time complexity (or $O(m \cdot n)$, if we consider m to be the average string length). Ukkonen's original paper and a more detailed description in [Gus97] proved that the space complexity is the same.

A Generalized Suffix Tree is a compressed trie data structure, containing all the suffixes of the given strings. Each node corresponds to a substring and has references to the original strings that contain it. In order to compute the longest common substring between two strings, one has to find the deepest node that has references to both strings.

For two strings of length m_1 and m_2, since both the construction and the traversal of the Generalized Suffix Tree for finding the deepest node are linear, we can say that the *deq distance* can be computed in linear time ($O(m_1+m_2)$) while the best known algorithm for the *longest subsequence distance* is quadratic ($O(m_1 \cdot m_2)$).

4.2.3 Proposed metrics based on set semantics

4.2.3.1 Common n-grams

At the beginning of this chapter we showed how to transform a binary program into a sequence of symbols from an alphabet of operations. Some techniques for computing the similarity based on string semantics were presented in the previous subsection. In the following, we will provide a method for transforming strings into sets and working with set semantics.

The basic idea is to transform sequences into n-grams. The n-grams of a sequence S are all the sub-sequences of n consecutive elements from S.

We will exemplify on the sequence $s_1, s_2, s_3, s_4, s_5, s_6$. For $n = 4$, the 4-grams of this sequence are (s_1, s_2, s_3, s_4), (s_2, s_3, s_4, s_5) and (s_3, s_4, s_5, s_6).

We will define $\mathcal{G}$, the set of all possible n-grams, and $\mathcal{P}(\mathcal{G})$, the set of all subsets of $\mathcal{G}$. If the number of OpCodes symbols is $|\Sigma|$, then the cardinality of $\mathcal{G}$ is the number of distinct sequences of length n with elements from Σ. This number is $|\Sigma|^n$.

We will define $ng : \Sigma^\star \to \mathcal{P}(\mathcal{G})$ as the function that computes the set of n-grams extracted from a sequence of symbols from $\Sigma^\star$.

As an n-gram has a variable-length representation (depending on n), in the implementation we have computed a hash function (CRC32) on each. If $|\Sigma|^n \gg 2^{32}$ (the CRC32 hash takes values from 0 to $2^{32} - 1$) another hash function should be chosen, otherwise hash collisions will decrease the quality of the solution.

The measure of similarity between two sequences S_1 and S_2 will be proportional with the number of common n-grams. If the two buffers are very similar, they will have many common n-grams. If they represent different programs, they will have zero or few common n-grams.

Formally, we will define this similarity measure by the following formula:

$$CNsim(S_1, S_2) = \frac{|ng(S_1) \cap ng(S_2)|}{\max(|ng(S_1)|, |ng(S_2)|)} \tag{4.12}$$

where the function $| \bullet |$ computes the size of set.

The problem with this approach is that there might be some library code, that is common to some of the programs. For the plagiarism detection example, the students might share some code given by the instructor. If the proportion of library / legally shared code does not exceed the proportion of normal code, the similarity function should work well. Unfortunately, we can't rely on this assumption.

4.2.3.2 Weighted common n-grams

To improve the metric above by addressing the issue of legally shared code, we will make some small modifications in similarity function.

So far, we have considered that each n-gram has an equal importance. However, if an n-gram is common to a large number of programs, the corresponding piece of code is legally shared. By contrary, if an n-gram is only common to two or three program, and others don't contain it, it might be an indication that they share a pretty specific piece of code.

Following the idea above, we will compute the number of appearances for every n-gram in the given sequences of symbols. If an n-gram appears several times in a sequence it will only be counted once. Based on the number of appearances, a weight will be computed for each n-gram. This weight must be high for rare n-grams, that appear in just a few sequences and should decrease as the number of n-grams increases. For n-grams where the number of appearances is over some threshold, the weight should be very low, as they probably belong to legally shared code. For the similarity metric, instead of using the sizes of the sets as in Equation 4.12, we will sum on the weights of the n-grams in those sets.

In the definitions that follow, we will consider that we have N sequences of symbols $S_1, S_2, \dots S_N \in \Sigma^\star$. The goal is to re-design the function from Equation 4.12 such that it takes into account the weights of the n-grams found in the N sequences.

The number of appearances for an n-gram $g \in \mathcal{G}$ can also be defined as a function $f : \mathcal{G} \to \mathbb{N}$, as in Equation 4.13.

$$f(g) = |\{i \in \mathbb{N} \mid 1 \le i \le N \wedge g \in ng(S_i)\}| \tag{4.13}$$

Having the f function defined, we are able to compute the weight of an n-gram with the function $w : \mathcal{G} \to \mathbb{R}$, as in Equation 4.14.

$$w(g) = \frac{1}{1 + e^{\frac{1}{4}\cdot(f(g)-\theta)}} \tag{4.14}$$

A plot that describes the evolution of this function (for $\theta = 5$, $\theta = 10$ and $\theta = 20$) is presented in Figure 4.4. On the x axis we have the number of appearances of a given n-gram and on on the y axis we have it's weight. The constant θ influences the threshold for legally shared code and can be a parameter given by the user. We can observe that the weight of an n-gram is a real number that takes values between 0 and 1 and decreases exponentially for a large number of appearances.

Knowing the weight of each n-gram, a new similarity metric will be derived in Equation 4.15, similar to Equation 4.12 but also taking the weights into account.

$$WCNsim(S_1, S_2) = \frac{\displaystyle\sum_{g \in ng(S_1) \cap ng(S_2)} w(g)}{\max(\displaystyle\sum_{g \in ng(S_1)} w(g), \sum_{g \in ng(S_2)} w(g))} \tag{4.15}$$

To highlight the connection between Equation 4.12 and Equation 4.15, we notice that $CSSim$ is just a particular case of $WCNsim$, where $w(g) = 1$, $\forall g \in \mathcal{G}$. However, by using the definition of w from Equation 4.14 instead, we obtained better results, as we will see in the experimental section.

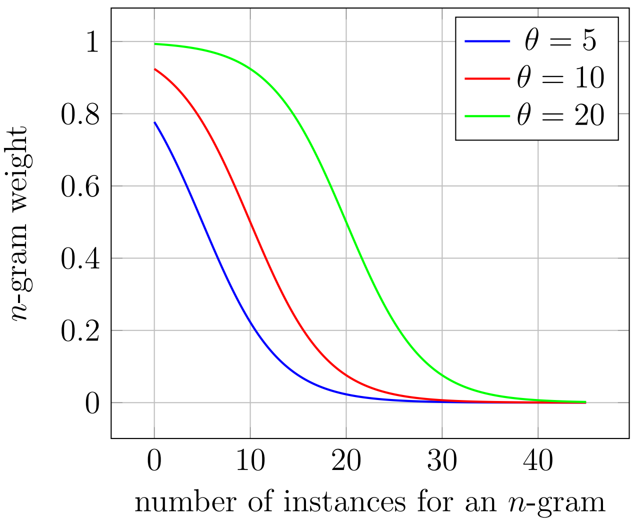


Figure 4.4: The weight function w plotted against the number of appearances $f(g)$

4.3 Proposed Techniques for Dealing with Structured Code

Some compilers, especially those that produce native code do not maintain the structure of the source code, making it harder to separate the binary code into functions or classes. On the other hand, Java compilers [JSGB00], including the one for Android produces binary code that can easily be separated into packages, classes and methods, giving it a hierarchical structure.

The similarity measure presented in this section will take advantage of this property in order to provide more accurate results.

4.3.1 Hierarchical code features extraction

This subsection describes how relevant features can be extracted from an Android binary program, even if it's compiled and we don't have access to the source code.

According to [For14], an Android application package (called *APK*) is a zip archive that contains the application's code, resources and manifest. The entire application binary code is stored in a single file inside the archive, called *classes.dex*. This file is obtained after the Java code is compiled to bytecode, then converted to Dalvik Executable (*dex*) format.

Reverse engineering tools like baksmali [Gru15] are able to extract the *classes.dex* file from an APK, rebuild the classes and packages and disassemble the bytecode into an assembly-like format called *smali*. Figure 4.5 shows the structure of the *classes.dex* file extracted from an android application using the baksmali tool. The gray rectangles with rounded corners correspond to Java packages and will be extracted as a hierarchy of folders. Each package can contain other packages or Java classes, represented as simple rectangles and extracted as `.smali` files. Each such file corresponds to a Java class but instead of Java code, they contain assembly-like instructions. The content of `.smali` files is also divided into methods.

Following an approach similar to the previous section, each smali instruction is associated with a distinct symbol. A sequence of n consecutive instructions from a method will be called an n-gram. Internally, each method will be represented as a set of n-grams instead of a sequence of instructions. The main distinction from the previous approach

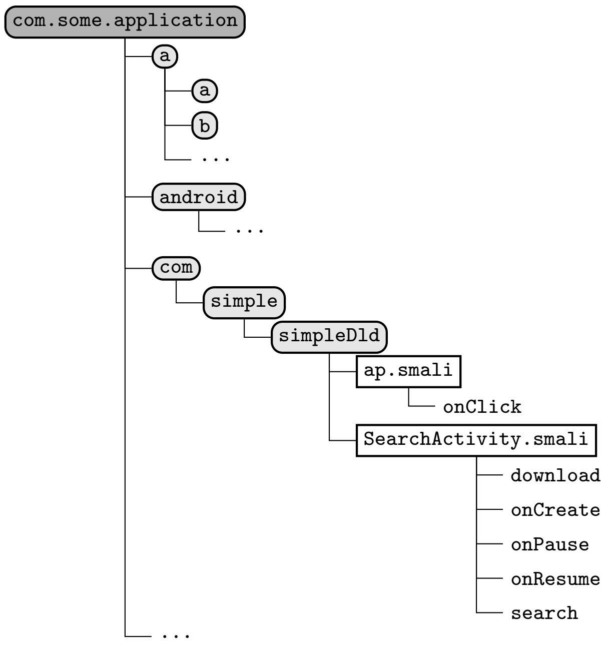


Figure 4.5: *classes.dex* structure as extracted by baksmali

is that for Android applications we have enough information to give the application a hierarchical structure, while in the previous works the entire application was represented as a single set of n-grams.

For formalism, we will make the following notations:

Let $\mathcal{G}$ be the set of all possible n-grams that can be extracted from an application code. Since a method is a set of n-grams, we will define the set of methods as $\mathcal{M} = \mathcal{P}(\mathcal{G})$ (the set of subsets of $\mathcal{G}$). Similarly, a class can be defined as a set of methods, and a package as a family containing classes and other packages. We will not allow a package to contain itself, directly or indirectly. The family of method containers, formed by classes and packages will be denoted by $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ (elements from $\mathcal{C}_1$ are classes and elements from $\mathcal{C}_2$ are packages). An application can be abstracted as a simple package that contain the top-level packets.

4.3.2 Similarity functions description

The basic similarity that we need to compute is the similarity between two methods. Since methods are represented as sets, the most natural similarity function is the Jaccard similarity [Jac01], that is 0 for disjoint sets and computes the ratio between the size of the sets intersection and the size of the sets union, as in Equation 4.16.

$$\begin{gathered} msim : \mathcal{M} \times \mathcal{M} \to [0,1] \\ msim(M_1, M_2) = \begin{cases} 0, \text{ if } M_1 \cap M_2 = \emptyset \\ \dfrac{|M_1 \cap M_2|}{|M_1 \cup M_2|}, \text{ otherwise} \end{cases} \end{gathered} \tag{4.16}$$

The Jaccard similarity has some important advantages in our scenario: first of all, the similarity function will always take values between 0 and 1, with 0 only for methods represented by disjoint sets of n-grams and 1 only for identical sets. Another advantage is that the Jaccard distance, defined as $d_J : \mathcal{M} \times \mathcal{M} \to [0,1], d_J(M_1, M_2) =$

$1 - msim(M_1, M_2)$ is a proper metric and can be used in various clustering algorithms [RU12].

The Jaccard distance may be sufficient for non-hierarchical code features, as each application can be represented as a large set of n-grams. However, for computing the similarity between hierarchical structures, a further step is required. Whether we are computing the similarity between two classes or between two applications, a new similarity function between sets of sets is required, as in Equation 4.17.

$$sim : \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \rightarrow [0, 1] \tag{4.17}$$

Considering two sets of methods, $C_1, C_2 \in \mathcal{P}(\mathcal{M})$, $C_1 = \{m_{11}, m_{12}, \ldots, m_{1k}\}$, $C_2 = \{m_{21}, m_{22}, \ldots, m_{2p}\}$, their similarity should be estimated by the number of methods they have in common:

- if $C_1 = C_2$ (the two sets are comprised exactly of the same methods), then $sim(C_1, C_2) = 1$;

- if the similarity between any method from C_1 and any method from C_2 is very small ($msim(m_{1i}, m_{2j}) < \epsilon$, $\forall 1 \leq i \leq k, 1 \leq j \leq p$), then the similarity between C_1 and C_2 should be 0 or close to 0;

- if some methods from C_1 match totally or partially methods from C_2, the similarity function should reflect the match percentage.

We can start by associating each pair $(m_1, m_2) \in C_1 \times C_2$ with a weight corresponding to their similarity, $w(m_1, m_2) = msim(m_1, m_2)$. At this point, we can find a matching between the two sets of methods by solving the *maximum weighted bipartite matching* problem (also called the assignment problem). Several polynomial algorithms exist for this task, the most notable one being the Hungarian algorithm [Kuh55].

In Figure 4.6, we represented two such methods sets, C_1 containing 5 methods and C_2 containing 4 methods. Methods with a high similarity are connected with lines (dashed or contiguous). The Hungarian algorithm can find a match between the two sets (the contiguous lines), such that each method from C_1 matches at most a method from C_2 and each method from C_2 matches at most a method from C_1 and the sum of the weights of the matches is maximal.

In what follows, we will consider that the Hungarian algorithm takes as input two sets $C_1, C_2 \in \mathcal{P}(\mathcal{M})$ and outputs a set of paired items $bm(C_1, C_2)$. The $bm : \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \rightarrow \mathcal{P}(\mathcal{M} \times \mathcal{M})$ has the following properties:

- $\forall (x_i, y_i), (x_j, y_j) \in bm(C_1, C_2)$, $x_i \neq x_j$ and $y_i \neq y_j$. This property will ensure that no method belongs to more than one match.

- $|bm(C_1, C_2)| = \min(|C_1|, |C_2|)$. In other words, every item from the smallest set will be matched with an item from the largest set. If the sets have different cardinality, some items from the largest set will remained unmatched.

- $\sum\limits_{(x,y) \in bm(C_1,C_2)} msim(x, y)$ is maximal. Since every element of the sum is at most 1 and the number of elements is the smallest between C_1 and C_2, it follows that this sum is smaller than the size of any of the two sets.

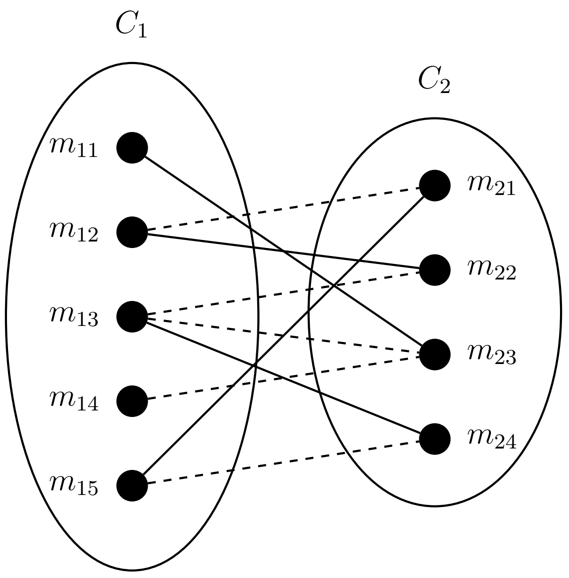


Figure 4.6: Bipartite match between two sets of methods

In the Jaccard similarity formula, we can apply the inclusion-exclusion principle and obtain:

$$sim_J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

The contents of the bipartite match can be seen as an approximate intersection between the sets C_1 and C_2. To devise the new similarity function, we will replace the intersection cardinality in the equation above with a function that depends on the output of the *bm* function:

$$sim(X,Y) = \frac{MatchScore(bm(X,Y))}{|X| + |Y| - MatchScore(bm(X,Y))} \tag{4.18}$$

$MatchScore(match)$, where $match = bm(X,Y)$ should be equal to $|X \cap Y|$ when the weights in the bipartite match are 1 (the matched methods are identical) and should take lower values otherwise. There are two models for computing this function:

- *thresholded match*: for each pair in the bipartite match, add 1 if the pair has a high similarity and 0 if the pair has a low similarity:

$$\begin{aligned} &MatchScore(match) = \\ &\qquad |\{(x,y) \in match \mid msim(x,y) \geq \theta\}| \end{aligned} \tag{4.19}$$

- *contiguous match*: add every similarity from the best bipartite matching:

$$MatchScore(match) = \sum_{(x,y) \in match} msim(x,y) \tag{4.20}$$

4.3.3 Hierarchical similarity algorithm

Algorithm 3 computes the similarity between two method containers (classes or packages). As mentioned in section 4.3.1, applications are also abstracted as packages so the algorithm will successfully compute the similarity between two applications.

Algorithm 3 HIERARCHICAL-SIMILARITY(X, Y)

```
Require: Two method containers $X, Y \in \mathcal{C}$
    $X = \{x_1, x_2, \ldots x_k\}$, $Y = \{y_1, y_2, \ldots y_p\}$
Ensure: the similarity between $X$ and $Y$
 1: if $(X \in \mathcal{C}_1$ and $Y \in \mathcal{C}_2)$ or $(X \in \mathcal{C}_2$ and $Y \in \mathcal{C}_1)$ then
 2:     return 0
 3: else if $X \in \mathcal{C}_1$ and $Y \in \mathcal{C}_1$ then
 4:     for $i = 1 \to |X|$, $j = 1 \to |Y|$ do
 5:         $w(i,j) \leftarrow msim(x_i, y_j)$
 6:     end for
 7: else
 8:     for $i = 1 \to |X|$, $j = 1 \to |Y|$ do
 9:         $w(i,j) \leftarrow$ HIERARCHICAL-SIMILARITY$(x_i, y_j)$
10:     end for
11: end if
12: $match \leftarrow$ COMPUTE-BIPARTITE-MATCH$(w)$
13: return $\dfrac{MatchScore(match)}{|X| + |Y| - MatchScore(match)}$
```

The algorithm begins by examining the type of the received containers. If the two containers are of different natures (a class and a package) they can't be compared, so their similarity is 0 (line 2).

If X and Y have the same type, we will build a weight matrix with the pairwise similarity between all the elements of X and Y. If X and Y are classes, their elements are methods. In this case we will use the *msim* function described in the previous section to compute their similarity (line 5). If they are packages, it means that their elements are also method containers, so we need to make a recursive call to HIERARCHICAL-SIMILARITY in order to compute their similarity (line 9).

Having the weights matrix, we can compute the maximum weighted bipartite matching (line 12) and use it in the similarity function (line 13). The *MatchScore* function can follow any of the two strategies presented in the previous section (*thresholded match* and *contiguous match*).

To compute the complexity for Algorithm 3, we will consider the size of the input as the number of methods the received application or package has. We may consider that the two inputs have roughly the same number of methods. Since we have a divide and conquer algorithm, we can use the master theorem [CLR+01], that computes the complexity for recursions having the form from Equation 4.21.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{4.21}$$

In our case, b is the number of classes or sub-packages a package have (for simplicity, we consider it to be constant, $b = |X| = |Y|$). The recursive call in line 9 is done for every pair of items in the two sets, so $a = |X| \cdot |Y| = b^2$ times.

Finally, after the recursive calls, the Hungarian algorithm is called on the weights matrix, but the complexity of this algorithm will be $O(b^3)$, which doesn't depend on n. It follows that $f(n) = O(1) = O(n^0)$.

To apply the master theorem, we need to compute $\log_b a$ and compare it with c, where $f(n) = O(n^c)$. From the previous paragraph, we have that $c = 0$. $\log_b a = \log_b b^2 = 2$.

Since $0 < 2 \Rightarrow c < \log_b a$ and we are in the first case of the master theorem. In this case, the algorithm's complexity is $O(n^{\log_b a})$. We conclude that the complexity for Algorithm 3 is $O(n^2)$, where n is the average number of methods an application or package has.

As a note, the master theorem stands if $b > 1$. The case $b = 1$ corresponds to the situation where all the application code is in a single class (unlikely but possible). In this case, The algorithm's complexity will be given by the complexity of the Hungarian algorithm and will be $O(n^3)$.

4.4 Automatic Features Selection

In the first section of this chapter we have shown how a sequence of symbols can be extracted from a binary program. Although, at this point we can perform malware detection and clustering based on the list of symbols found, we will go one step further. In this section we will show how to select a subset of these symbols in such a way that the abstract representation of the code keeps its specificity and also loses some of the alterations performed through obfuscation.

As shown before, we will denote by $\mathcal{O}$ the set of all instructions. $\mathcal{O}^\star$ will be then the set of all the finite strings with elements from $\mathcal{O}$. An element of $\mathcal{O}^\star$ can be a *function* or a *method* (a list of instructions).

Similar to the tf function defined above, we can define $TF : \mathcal{O}^\star \to \Sigma^\star$, a function that transforms a method (a string of instructions) into a finite string of symbols from Σ. We will call such an operation *normalization.*

$$TF(i_1 i_2 \dots i_m) = tf(i_{k_1}) tf(i_{k_2}) \dots tf(i_{k_p}) \tag{4.22}$$

$i_{k_1} i_{k_2} \dots i_{k_p}$ is a subsequence of $i_1 i_2 \dots i_m$ that contains all the reachable instructions.

The length of the *normalized* method might not be the same as the length of the original, because some instructions might be skipped because they cannot be reached while others might be normalized into the empty symbol ϵ, which does not alter a string through concatenation.

For $\Lambda \subseteq \Sigma$, a subset of the symbols set, we will define a function $tf_\Lambda : \mathcal{O} \to \Lambda \cup \{\epsilon\}$ in the following way:

$$tf_\Lambda(i) = \begin{cases} tf(i) & \text{, if } tf(i) \in \Lambda \\ \epsilon & \text{, otherwise} \end{cases} \tag{4.23}$$

This function will transform an instruction into its corresponding symbol if that symbol belongs to Λ, otherwise the instruction will be transformed into the empty symbol ϵ.

TF_Λ will have a formula similar to Equation 4.22, with tf replaced by tf_Λ.

The next step is to design a fitness function, $f : \mathcal{P}(\Sigma) \to \mathbb{R}$, where $\mathcal{P}(\Sigma)$ contains all the subsets of the symbols set Σ. For a subset $\Lambda \subseteq \Sigma$, $f(\Lambda)$ will tell us how good is the choice of symbols from Λ.

Having this fitness function, we will search for the best choice of Λ ($\underset{\Lambda}{\arg\max} f$). The size of the search space in this case will be the number of subsets of Σ, $\mid \mathcal{P}(\Sigma) \mid = 2^{|\Sigma|}$.

This search space is too big for an exhaustive search, so we will try a Genetic Algorithm and Particle Swarm Optimization for finding a good solution in a reasonable amount of time.

4.4.1 The Fitness Function

This subsection will help us evaluate how good is a choice of the symbols subset Λ. A good choice would allow us to find common features to similar methods, features that are unique to those methods. Our choice of features is the n-grams.

n-grams are groups of consecutive OpCodes or in our case symbols from Λ, of length n. From every normalized method we will extract all the n-grams. If $l_1 l_2 \dots l_p$ is a normalized method, with $l_k \in \Lambda$, $\forall k \in \{1, 2, \dots p\}$ and $p \geq n$, the n-grams are $l_1 l_2 \dots l_n$, $l_2 l_3 \dots l_{n+1}$, $\dots$, $l_{p-n+1} l_{p-n+2} \dots l_p$.

Some of the n-grams that appear in a method may correspond to library code or may be too general. For this reason, we cannot say that two methods are similar just because they have several n-grams in common. The common n-grams must also not belong to other methods.

We have built a training cleanset of 55230 .NET samples from which we extracted 558695 different methods. We have also created 272 clusters from 1769 different methods. In each cluster we should have similar methods.

The clusters were obtained by two methods:

- By manually selecting similar malware samples. We selected groups of samples from the same malware family and performed reverse engineering on them. Methods that did the same things but didn't have the same code (not even the same sequence of OpCodes) were grouped in the same cluster.

- By using obfuscation tools on existing samples, other than the ones from the cleanset.

A good choice for Λ would allow us to find common n-grams for the clusters that are not present in the methods from the cleanset.

One issue that comes to mind is the choice for the value of n. If n is too small, most of the possible n-grams will already be in the cleanset. For example, if $n = 2$, every combination of 2 symbols might be in the cleanset, so all the common n-grams in the clusters will be invalid. If n is too big, the cleanset will be less filled but it will get harder to find common n-grams. We will deal with this problem by trying all the values of n from a certain range, that was determined experimentally.

The extraction of n-grams from a method is detailed in Algorithm 4. It first performs a filtering of the received method by extracting only the symbols present in Λ (lines 2-7). From the filtered methods, all the substrings of length n are extracted and appended to the *nGrams* set (lines 8-10). The APPEND-CHAR method used at line 5 will append the character specified as the second parameter to the string specified as the first parameter.

Using the EXTRACT-NGRAMS method, we can finally compute the fitness function, in Algorithm 5. This algorithm will compute the fitness of a subset $\Lambda \subseteq \Sigma$, given a cleanset and some clusters.

For each n in a given range, the algorithm will extract $clnNgrs$, a set that contains all the "clean" n-grams extracted from the cleanset's methods (lines 4-7). Using this set, a cluster score will be computed for each cluster (lines 8-10). As shown in the algorithm,

Algorithm 4 EXTRACT-NGRAMS($method, \Lambda, n$)

```
Require: A string from $\Sigma^\star$, $method$.
Require: A set of symbols $\Lambda \subseteq \Sigma$.
Require: An integer $n$.
Ensure: A set of $n$-grams, $nGrams$
 1: $nGrams \leftarrow \emptyset$
 2: $filtered \leftarrow$ ""
 3: for $i = 1 \rightarrow | method |$ do
 4:     if $method[i] \in \Lambda$ then
 5:         APPEND-CHAR($filtered, method[i]$)
 6:     end if
 7: end for
 8: for $i = 1 \rightarrow | filtered | - n + 1$ do
 9:     $nGrams \leftarrow nGrams \cup \{$SUBSTRING$(filtered, i, i+n)\}$
10: end for
11: return $nGrams$
```

Algorithm 5 COMPUTE-FITNESS($\Lambda, cleanset, clusters$)

```
Require: A set of symbols $\Lambda \subseteq \Sigma$.
Require: A list of strings from $\Sigma^\star$, $cleanset$.
Require: A list of lists of strings from $\Sigma^\star$, $clusters$.
Ensure: A real nr. $score$, that evaluates the quality of $\Lambda$
 1: $score \leftarrow 0.0$
 2: for $n = N\_MIN \rightarrow N\_MAX$ do
 3:     $nScore \leftarrow 0.0$
 4:     $clnNgrs \leftarrow \emptyset$
 5:     for $method$ in $cleanset$ do
 6:         $clnNgrs \leftarrow clnNgrs \cup$ EXTRACT-NGRAMS($method, \Lambda, n$)
 7:     end for
 8:     for $cluster$ in $clusters$ do
 9:         $clusterScore \leftarrow$ COMP-CL-SCORE($cluster, clnNgrs, \Lambda, n$)
10:         $nScore \leftarrow nScore + clusterScore$
11:     end for
12:     if $nScore > score$ then
13:         $score \leftarrow nScore$
14:     end if
15: end for
16: return $\frac{score}{| clusters |}$
```

the fitness function will be the maximum average of the cluster scores, as in Equation 4.24.

$$score = \max_n \frac{\sum_{cluster} clusterScore}{| clusters |} \tag{4.24}$$

We have chosen to keep the maximum value because the choice of n can be made after the choice of Λ. If a small set is chosen for Λ, a big n will ensure that we have a big enough number of possible n-grams. For a bigger set, a smaller n should suffice. An average was computed instead of a sum, so that the fitness score will not depend on the number of clusters.

The last detail of the algorithm is how to compute the score for each cluster. This computation will be done in Algorithm 6. If we are able to find enough common n-grams that are not present in the cleanset, in the cluster's methods, we will give that cluster a score of 1.0. If the n-grams are only common to some of the methods, we will divide this score by 2, for each missed method. For example, if we only manage to find enough common n-grams for 4 out 7 methods in a cluster, the score for that cluster will be 0.125, because we missed **3** methods, so $clusterScore = \frac{1}{2^{\mathbf{3}}}$. This approach should give higher fitness values to choices of Λ that make possible the detection of the entire cluster. If not all samples can be detected, the score for that cluster will decrease exponentially because it means we cannot detect all the methods based on the n-grams.

Algorithm 6 COMP-CL-SCORE$(cluster, clnNgrs, \Lambda, n)$

```
Require: A list of strings from $\Sigma^\star$, $cluster$.
Require: A set of clean $n$-grams, $clnNgrs$.
Require: A set of symbols $\Lambda \subseteq \Sigma$.
Require: An integer $n$.
Ensure: A real number $clusterScore$.
 1: $counts \leftarrow \{\}$
 2: for $method$ in $cluster$ do
 3:     for $ngr$ in EXTRACT-NGRAMS$(method, \Lambda, n)$ do
 4:         if $ngr \notin clnNgrs$ then
 5:             ADD-COUNT$(counts, ngr)$
 6:         end if
 7:     end for
 8: end for
 9: $detections \leftarrow$ COUNT-DETECTIONS$(counts)$
10: $clusterScore = 1.0$
11: $sum = 0$
12: for $i =\mid cluster \mid \rightarrow 2$ do
13:     $sum \leftarrow sum + detections[i]$
14:     if $sum \geq$ DETECTION_THRESHOLD then
15:         break
16:     end if
17:     $clusterScore \leftarrow \frac{clusterScore}{2}$
18: end for
19: if $sum <$ DETECTION_THRESHOLD then
20:     $clusterScore \leftarrow 0$
21: end if
22: return $clusterScore$
```

In lines 1-8, we count how many times each n-gram appears in the cluster's methods. If a n-gram appears more than one time in a method it will only be counted once, since the method EXTRACT-NGRAMS returns a set. The method ADD-COUNT, called in line 5 will just increase the count for ngr in $counts$.

The next step is to count the detections, in line 9. This function will return a vector *detections*, where *detections*[k] will tell us how many n-grams are common to k methods. In order to determine how many methods have enough common n-grams we will use a greedy approach: starting from $\mid cluster \mid$ downto 2, we will sum the elements in the *detections* vector (lines 12-13), until the sum gets bigger than a specific threshold. At each step, if the sum is smaller, we divide the $clusterScore$ by 2 (lines 14-17).

Of course, this approach is not always correct. If we have two n-grams, the first being common to the first $\mid cluster \mid -1$ methods and the second to the last $\mid cluster \mid -1$, both will be common to only $\mid cluster \mid -2$ methods. Still, this greedy approach gives us a good approximation. Also, the cluster from the example above can still be detected because the two n-grams will not detect clean methods. Another reason for this greedy approach is the performance. It is much faster than computing the maximum number of methods that have at least `DETECTION_THRESHOLD` common n-grams.

Having the fitness function designed, all we have to do is search for a subset of symbols Λ, that maximizes this function. Unfortunately, the search space is too big for an exhaustive search to be computationally feasible. For this reason we will use two bio-inspired algorithms in order to find a good solution in a reasonable amount of time.

4.4.2 Selection Using a Genetic Algorithm

Genetic algorithms were introduced in 1975 [Hol75] and since then, they were successfully used in many optimization problems. For our problem, we need to optimize the search for the best subset Λ (the one with the greatest fitness score).

In order to solve an optimization problem with genetic algorithms, one must represent a possible solution as an individual. Then, generate a random population that is governed by the principles of evolution: the fittest individuals have greater chances of reproduction, leading to better generations. In the end, the best individual (solution) is selected, as a solution for the optimization problem. In a simplified genetic vision, each individual will be represented as a chromosome - a sequence of genes.

In our case, an individual will be a possible choice for the subset Λ. A random population of subsets will be generated and we will combine existing solutions in order to generate better ones.

The genetic algorithms use two operators on the population, in order to obtain a new generation:

- *crossover*: This operator takes two chromosomes and combines them in order to obtain two others. There is a probability $p_{crossover}$ that the two inputs are combined, otherwise they will be returned unchanged. Usually $p_{crossover}$ is high, about 80%-95%.

- *mutation*: Each chromosome, with a probability $p_{mutation}$ will get altered, by randomly modifying some of its features. Usually, $p_{mutation}$ is small, about 0.5%-1%.

When selecting two individuals for *crossover*, they must be selected with a probability proportional with their fitness. In our implementation, we have used the Roulette Wheel Selection [Bäc96]. This selection method assumes that all the individuals are placed on a circle, each having a sector of size proportional with their fitness function. Then, a point on the circle is chosen randomly, and the individual whose sector contains that point is selected. It is very important that the chances for an individual to be selected for reproduction is proportional with its fitness function, otherwise the algorithm would become a random search.

In order to prevent losing the best solution found, *elitism* is also used. The best `ELITE_SIZE` individuals will always survive to the next generation, unaltered. We will sort the existing population by the fitness function and the top `ELITE_SIZE` individuals will be transferred to the next generation. Note that this individuals will also have high chances of reproduction.

The implementation is detailed in Algorithm 7. For each generation, the following steps are performed:

- the fitness for each individual is computed (lines 3-5)
- the best individuals are passed to the new population (line 6)
- the rest of the new population is filled with new individuals obtained through crossover (lines 7-20)
- all the individuals, except for the elites might be mutated (lines 21-25)

To complete the algorithm's description, we must mention the encoding of the chromosomes and the details of the *crossover* and *mutation* operators.

One chromosome or individual will have one choice of Λ. We have chosen the simplest form of encoding, the binary encoding. For each chromosome, we will have a string of bits of length $\mid \Sigma \mid$. If a bit is set, it means that the corresponding element from Σ belongs to Λ.

The genetic operators are easy to implement using this encoding. We considered the most fit type of crossover to be the uniform crossover. Any other type of crossover would involve some correlation between symbols from Σ and we do not want that.

The uniform crossover receives two parents as inputs and outputs two offspring. For each position in the bit string, one of the parent's bits from that position goes to the first offspring, while the other goes to the second offspring.

If an individual is selected for mutation, some randomly selected bits from his bit string are inverted.

The genetic algorithm presented above should evolve towards a Λ value that maximizes the fitness function from the previous subsection. One problem that might occur is a stuck in a local optimum. To prevent this issue, an extra step was added to the algorithm, that re-initializes the population to random values if they get stuck at the same solution for too long.

The most time consuming operation in this algorithm is computing the fitness function for each individual, since it involves extracting n-grams for all the methods in the cleanset. The algorithmic complexity for the rest of the steps performed for each generation is linear in the population size (if we consider the size of the Σ set to be constant). The number of generation can be fixed, or the algorithm can be left to run for a certain amount of time and the best solution extracted at the end.

4.4.3 Selection Using Particle Swarm Optimization

Particle Swarm Optimization was introduced by [KE95] and is also used for solving optimization problems. This time, the method is inspired by the way the birds in a flock cooperate for finding food. When an individual finds some food, it makes sounds so the

Algorithm 7 GENETIC-ALGORITHM($cleanset, clusters$)

Require: A list of strings from $\Sigma^\star$, $cleanset$.
Require: A list of lists of strings from $\Sigma^\star$, $clusters$.
Ensure: A set of symbols $\Lambda \subseteq \Sigma$.

```
 1: $pop \leftarrow$ GENERATE-RANDOM-POPULATION()
 2: for $generation = 1 \rightarrow$ NR_GENERATIONS do
 3:     for $i = 1 \rightarrow \mid pop \mid$ do
 4:         $fitness[i] \leftarrow$ COMPUTE-FITNESS($pop[i], cleanset, clusters$)
 5:     end for
 6:     $newPop \leftarrow$ SELECT-ELITES($pop$, ELITE_SIZE)
 7:     while $\mid newPop \mid < \mid pop \mid$ do
 8:         $parent_1 \leftarrow$ ROULETTE-SELECT($pop, fitness$)
 9:         if $\mid newPop \mid = \mid pop \mid -1$ then
10:             $newPop \leftarrow newPop \cup \{parent1\}$
11:         else
12:             $parent_2 \leftarrow$ ROULETTE-SELECT($pop, fitness$)
13:             if RANDOM$(0, 1) < p_{crossover}$ then
14:                 $child_1, child_2 \leftarrow$ CROSSOVER($parent_1, parent_2$)
15:                 $newPop \leftarrow newPop \cup \{child_1, child_2\}$
16:             else
17:                 $newPop \leftarrow newPop \cup \{parent_1, parent_2\}$
18:             end if
19:         end if
20:     end while
21:     for $i =$ ELITE_SIZE $+ 1 \rightarrow \mid pop \mid$ do
22:         if RANDOM$(0, 1) < p_{mutation}$ then
23:             $newPop[i] \leftarrow$ MUTATION($newPop[i]$)
24:         end if
25:     end for
26:     $population \leftarrow newPop$
27: end for
28: return $population[\operatorname*{arg\,max}_{i=1 \rightarrow |population|} fitness[i]]$
```

nearby birds will know there's a food source in the area. The neighbours then change their direction of flight towards the calling bird, scouting for other sources of food in the way. Guided by sound, the whole swarm will eventually reach the best food source they could find.

As in a genetic algorithm, we also have a population (called *swarm*) of individuals (called *particles*). Each particle contains a candidate solution (e.g. a choice for Λ, for our problem), that is determined by the particle's position in a multi-dimensional space. The particles move by some pre-defined rules and their position and velocity is influenced by the best personal solution found so far and by the best solution in the swarm.

For our problem, the particle's model is described by the following tuple:

$$p = (X, V, X_{best}, best\_fitness) \tag{4.25}$$

The 4 components are:

- $X \in [0,1]^{|\Sigma|}$ is the particle's current *position*. This position is a point situated in a multi-dimensional cube. The number of dimensions is the size of the Σ set, each dimension corresponding to a certain symbol from Σ. In order to compute $\Lambda(p)$, the solution contained by such a particle, we must find the nearest point of integer coordinates (the only integers in that interval are 0 and 1) because a set either contains an element, either it doesn't. We will consider the projection of X on each dimension. If the value of the projection is greater than 0.5 (closer to 1), the corresponding symbol from Σ will belong to $\Lambda(p)$.

- $V \in [-1,1]^{|\Sigma|}$ is the particle's current *velocity*, and is also a vector in a multi-dimensional space.

- $X_{best} \in [0,1]^{|\Sigma|}$ is the particle's personal best, meaning the position with the highest fitness that the particle has ever reached.

- $best\_fitness$ is the fitness value for X_{best}.

The Algorithm 8 shows how PSO works. It starts by initializing the swarm with random particles (line 1). Then, for each iteration it computes the fitness function of every particle (lines 5-7) and updates the *globalBest* position and its fitness, if necessary. Every particle is then updated (lines 13-15). In the end, the algorithm returns the value for Λ corresponding to the best position found.

Algorithm 8 PSO($cleanset, clusters$)

```
Require: A list of strings from $\Sigma^\star$, $cleanset$.
Require: A list of lists of strings from $\Sigma^\star$, $clusters$.
Ensure: A set of symbols $\Lambda \subseteq \Sigma$.
1: $swarm \leftarrow$ GENERATE-RANDOM-PARTICLES()
2: $globalBestFitness \leftarrow 0$
3: $globalBest \leftarrow 0^{|\Sigma|}$
4: for $iteration = 1 \rightarrow$ NR_ITERATIONS do
5:     for $i = 1 \rightarrow |swarm|$ do
6:         $fitness[i] \leftarrow$ COMPUTE-FITNESS($\Lambda(swarm[i]), cleanset, clusters$)
7:     end for
8:     $bestFitness \leftarrow \max_{i=1\rightarrow|swarm|} fitness[i]$
9:     if $bestFitness > globalBestFitness$ then
10:        $globalBestFitness \leftarrow bestFitness$
11:        $globalBest \leftarrow swarm[\arg\max_{i=1\rightarrow|swarm|} fitness[i]]$
12:    end if
13:    for $i = 1 \rightarrow |swarm|$ do
14:        $swarm[i] \leftarrow$ UPDATE-PARTICLE($swarm[i], globalBest, globalBestFitness$)
15:    end for
16: end for
17: return $\Lambda(globalBest)$
```

A particle's velocity is updated by Equation 4.26, from [SE98a], that is a modified version from the original one specified in [KE95]. The constant ω is called inertia weight and it expresses how much importance is given to the previous velocity. ϕ_1 and ϕ_2

are positive real numbers, called acceleration constants. ϕ_1 shows how important is the personal best, while ϕ_2 represents the weight of the global best. r_1 and r_2 are random constants from the interval $(0, 1)$. If one of the vector's components is outside the interval $[-1, 1]$, it will get saturated. The values for ϕ_1, ϕ_2, ω were chosen following the guidelines from [SE98b].

The equation computes the new value of the velocity vector (V'), given the previous value (V), the current position (X) and the personal (X_{best}) and global best ($global\_best$) positions.

$$V' = \omega V + \phi_1 r_1 (X_{best} - X) + \phi_2 r_2 (global\_best - X) \tag{4.26}$$

To update the particle's position we will add to it the recently calculated velocity vector, as in Equation 4.27. The two vectors are added on the multi-dimensional space. If the projection on any dimension gets outside the interval $[0, 1]$, saturation is performed.

$$X' = X + V \tag{4.27}$$

This method has many similarities with the genetic algorithm from the previous subsection. It also initializes a random population that evolves in time, leading to better solutions. In our case, PSO was adapted for finding the subset Λ that maximizes the fitness function. Similar to the previous method, early convergence to a local optimum is also an issue, so the particles will re-initialize if the global best has been the same for too long. An advantage over the previous solution is that the swarm (population) size can be smaller. Since the bottleneck of both algorithms is the fitness function, it means that PSO can run more iterations in the same amount of time.

4.5 Experimental Evaluation

The concepts described in this chapters have been tested on various datasets extracted from real-world software. Two types of tests were performed:

- quality evaluation: we are interested on how well a distance metrics performs. A good distance metric will output small values for similar items and large values for dissimilar ones.
- performance evaluation: since we are usually working with a large number of samples, we are interested in how fast a given metric can be computed. In practice, we can use fast but unreliable metrics (prone to false positives) for prefiltering then employ costly but more accurate ones.

The quality evaluation can be performed using a confusion matrix similar to the one in Table 2.1.

For a distance metric, we will assign a threshold and will consider a pair of items whose distance is below the threshold to be similar, while a distance above the threshold will mean dissimilarity. The actual decision, whether two items are similar or not can be made by human experts. Such an expert can decide if two malware samples belong to the same family, if two binaries are different versions of the same program or if two coding assignments represent a case of plagiarism. For grouping malware samples into families we can also use a voting system based on multiple anti-virus engines detection names.

Based on the relation between the predicted similarity verdict and the actual similarity verdict, each pair of samples will be labeled into one of the following four categories:

- True Positive (TP) - The two items are similar and the distance metric also predicted they are similar.
- False Positive (FP) - The distance metric predicted similarity but expert classification decided otherwise.
- True Negative (FN) - The two items are similar but the distance metric failed to predict that.
- False Negative (TN) - The items are dissimilar and their distance is above the threshold.

Based on the number of pairs in each of the above categories, quality indices like Precision, Recall and F-Measure can be derived.

4.5.1 Detection of plagiarism cases among students homework

This experiment was performed on a collection of 946 homework submissions (the programming assignments from a semester). Our system outputs all the pairs of assignments that have a similarity over a certain threshold and are not belonging to the same student. To be able to compute the confusion matrix, we need to know which pairs of assignments were correctly classified as plagiarism and which were not. Since the total number of pairs from N assignments is $\frac{N(N-1)}{2}$ (in our case 446985), it is infeasible to take each pair and compare the submissions manually. Instead, we have used Moss, the system described in [SWA03] and manually checked the plagiarism reports. We have also verified the plagiarism cases reported by our application and not reported by Moss and added them to the list, in case we found a true positive. This approach is not perfect, as it misses the cases not detected neither by Moss, nor by our system, but gives a good approximation.

The number of pairs situated in each category of the confusion matrix is represented in Table 4.2.

Table 4.2: Plagiarism detection - simple measurements

Similarity metric	**TP**	**FP**	**FN**
Moss	186	0	119
$DEsim$	56	1399	249
$NCDsim$	153	144	152
$CNsim$	169	79	136
$WCNsim$	250	0	55

Based on these measurements we have computed the three quality indices described in the previous subsection and obtained the results in Table 4.3, that were also represented graphically in Figure 4.7.

By analyzing the results, we observed that Descriptional Entropy performed poorly. Although it managed to detect some of plagiarism cases, the high number of False Positives would make it impractical to use.

The Normalized Compression Distance performed quite better, as it obtained values above 50% for both Precision and Recall.

Table 4.3: Plagiarism detection - quality indices

Similarity metric	**Precision**	**Recall**	**F-measure**
Moss	100.00%	60.98%	66.15%
$DEsim$	3.85%	18.36%	10.47%
$NCDsim$	51.52%	50.16%	50.43%
$CNsim$	68.15%	55.41%	57.56%
$WCNsim$	100.00%	81.97%	85.03%

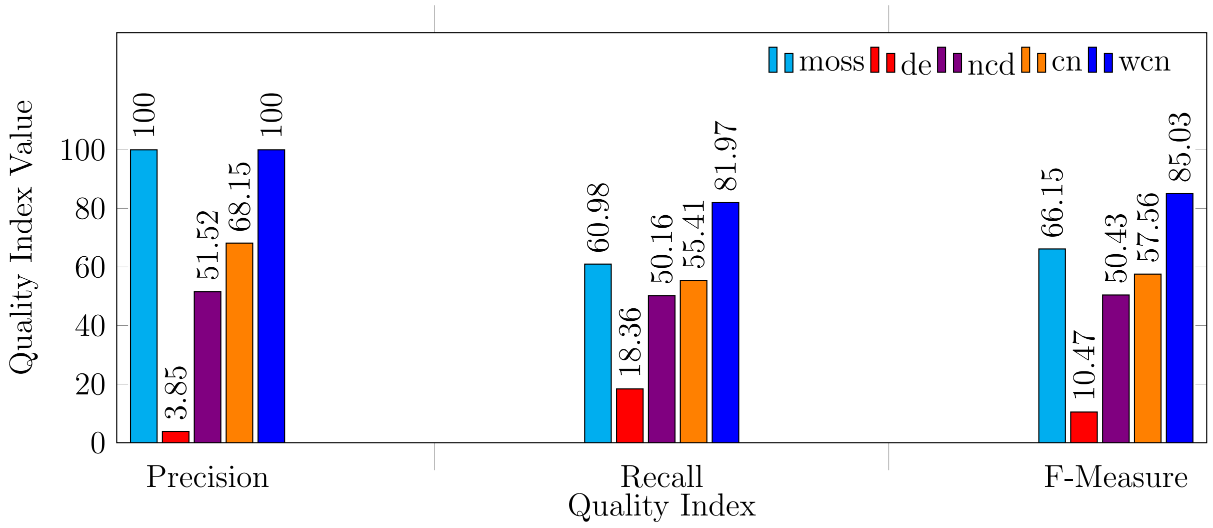


Figure 4.7: Plagiarism detection results

We also observed quite a large difference in the results obtained by the two metrics based on common n-grams. $CNsim$ obtained some promising results, but also a considerable number of False Negatives and False Positives. For a better understanding of this behaviour, we must consider the particularities of the student's assignments. They were given a library for measuring the performance of their programs but the usage was optional. A student that plagiarized someone else's homework might have removed that library from his project. Since the size of that library was larger than most of the assignment sizes, two very different programs that shared this library appeared to be more similar than the programs discussed above.

$WCNsim$ addressed this issue well, as it obtained improved results. For a similarity threshold of 50% we had no false alarms and the Recall was higher than the one obtained by Moss (this means that we actually found more plagiarism cases). Also, a fair number of the missed cases were not far below the threshold, and they were arguably cases where more than just functions and variable names differed.

To better understand how well can $WCNsim$ perform, we have plotted the ROC curve by varying the plagiarism threshold in Figure 4.8. We can see that if we are satisfied with 30% false positive rate (70% precision), we can detect more than 95% of the plagiarism cases. A lower precision means that the user should carefully verify the reported cases, as some false alarms might occur.

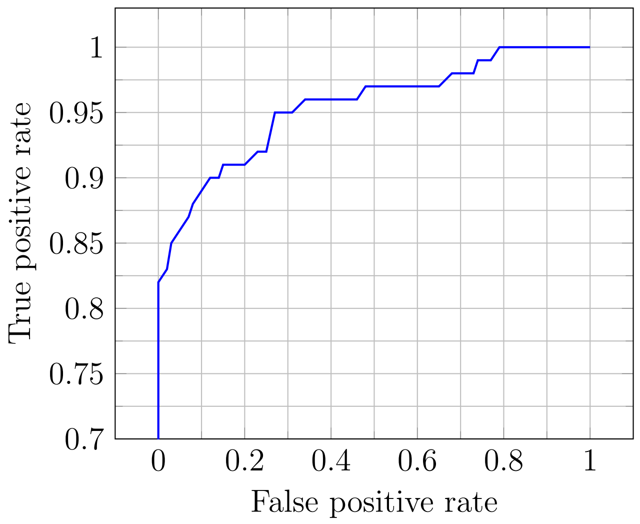


Figure 4.8: ROC curve for plagiarism detection using weighted common n-grams

4.5.1.1 Similarity assessment on a malware collection

The four proposed similarity metrics were also tested on a collection of 1742 malware samples that were already clustered manually into malware families.

We have taken each pair of samples and checked if their similarity is above the threshold for malware belonging to the same family and below otherwise. Based on this observation, each pair was labeled as a True Positive, False Positive, True Negative or False Negative.

The quality indices above were also computed, with a small difference. For the malware clustering, Precision is more important than Recall. It is not desirable to group together samples that are different. For this reason, we will use the value $\beta = 0.5$ in the F-measure index, to emphasize the importance of Precision.

The results obtained on this task were similar with the ones obtained in plagiarism detection. In Table 4.4 we have collected the measurements, while the quality indices were computed in Table 4.5.

Table 4.4: Malware similarity - simple measurements

Similarity metric	**TP**	**FP**	**FN**
$DEsim$	352	1134	1238
$NCDsim$	910	269	680
$CNsim$	958	263	632
$WCNsim$	1115	187	474

Table 4.5: Malware similarity - quality indices

Similarity metric	**Precision**	**Recall**	**F-measure**
$DEsim$	23.69%	22.14%	23.36%
$NCDsim$	77.18%	57.23%	72.15%
$CNsim$	78.46%	60.25%	73.99%
$WCNsim$	85.64%	70.17%	82.02%

As expected, Descriptional Entropy performed poorly, while the Weighted Common n-grams method got good results. The interesting fact observed in Figure 4.9 is that the differences between $NCDsim$, $CNsim$ and $WCNsim$ are not that big. Since the malware samples are form a more heterogeneous collection that the student's assignments, even some common library code might indicate that there is some similarity between the two samples.

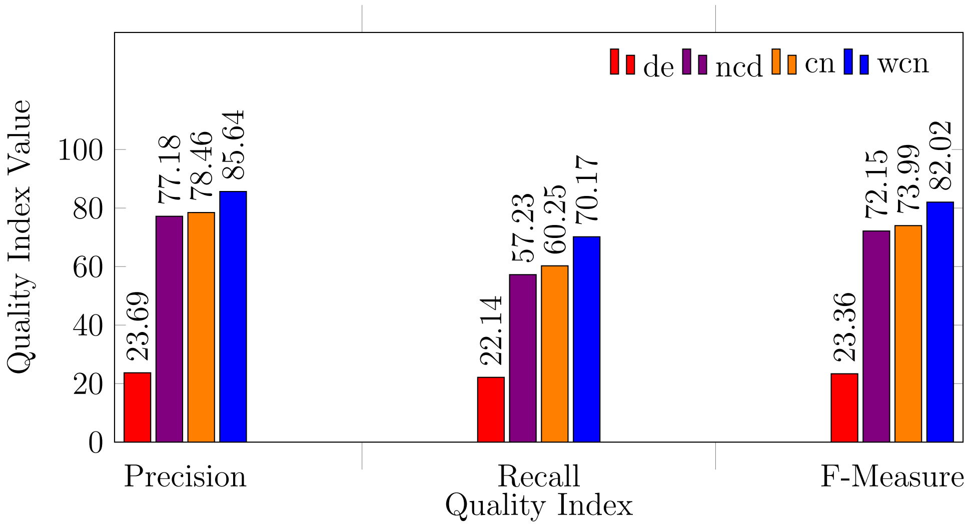


Figure 4.9: Malware similarity results

The running times of the system using the 4 metrics were also measured in Table 4.6. The $DEsim$ metric was particularly fast in this case and the whole algorithm can be improved even more, but due to the poor quality showed, it doesn't worth using. Although the running time for $WCNsim$ was almost double the running time for $CNsim$, we prefer using it because it got the best results.

Table 4.6: Malware clustering - execution times

Similarity metric	**Clustering time (s)**
$DEsim$	2.01
$NCDsim$	302.07
$CNsim$	24.58
$WCNsim$	45.18

4.5.2 Comparison between deq and longest subsequence distance

4.5.2.1 Clusters quality

The first experiment is meant to prove that the *deq distance* provides clusters of similar quality as the *longest subsequence distance*. A group of $N = 2011$ malware samples was used in a single linkage hierarchical clustering algorithm [Sib73]. A cluster id was assigned to each sample. We will denote by $c_{deq}(s)$ the cluster id assigned using the *deq distance* and $c_{lsd}(s)$ the cluster id assigned using the *longest subsequence distance*. For each pair of samples, we are interested if they were classified in the same cluster or not by the two algorithms. The confusion matrix in Table 4.7 shows the four categories a pair of samples can fall into, and is similar to the confusion matrix in Table **??**

- True Positive (TP): The pair was placed into the same cluster by both distances.

$$\begin{aligned} TP = |\{(i,j) \mid 0 \le i < j \le N \wedge \\ c_{deq}(S_1) = c_{deq}(S_2) \wedge c_{lsd}(S_1) = c_{lsd}(S_2)\}| \end{aligned}$$

- False Positive (FP): The pair was placed into the same cluster by *deq distance* but not by *longest subsequence distance.*

$$\begin{aligned} FP = |\{(i,j) \mid 0 \le i < j \le N \wedge \\ c_{deq}(S_1) = c_{deq}(S_2) \wedge c_{lsd}(S_1) \neq c_{lsd}(S_2)\}| \end{aligned}$$

- True Negative (TN): The pair wasn't placed into the same cluster by either distances.

$$\begin{aligned} TN = |\{(i,j) \mid 0 \le i < j \le N \wedge \\ c_{deq}(S_1) \neq c_{deq}(S_2) \wedge c_{lsd}(S_1) \neq c_{lsd}(S_2)\}| \end{aligned}$$

- False Negative (FN): The pair was placed into the same cluster only by *longest subsequence distance* and was missed by *deq distance.*

$$\begin{aligned} FN = |\{(i,j) \mid 0 \le i < j \le N \wedge \\ c_{deq}(S_1) \neq c_{deq}(S_2) \wedge c_{lsd}(S_1) = c_{lsd}(S_2)\}| \end{aligned}$$

Table 4.7: Confusion matrix for clusters quality evaluation

	$c_{deq}(S_1) = c_{deq}(S_2)$	$c_{deq}(S_1) \neq c_{deq}(S_2)$
$c_{lsd}(S_1) = c_{lsd}(S_2)$	TP	FN
$c_{lsd}(S_1) \neq c_{lsd}(S_2)$	FP	TN

The quality indices Precision, Recall and F-Measure are computed using the same formulas.

We have also considered another set of clusters, using the malware detection of several anti-virus products for our collection of samples. Samples detected with the same family name will belong to the same cluster. The same quality indices will be computed using these clusters as a benchmark. All the measure results are present in Table 4.8.

The results show that the clusters obtained with the *deq distance* are very similar with the ones obtained by the *longest subsequence distance* (Benchmark 1), as 98.65% Precision and 99.98% Recall were obtained.

The result against Benchmark 2 confirm that our distance can be successfully used in clustering and classifying real-world malware.

4.5.2.2 Performance evaluation

To evaluate the performance of the algorithms that compute the two distances, we have considered strings extracted from malicious samples of various lengths. For each length from 50 to 1000, counting from 10 to 10, a group of strings with that length was selected and both distances were computed for each pair.

Table 4.8: Quality measurements for the clusters obtained with the *deq distance*

Quality index	Benchmark 1	Benchmark 2
TP	374320	372821
FP	5118	6617
TN	1641532	1559097
FN	85	82520
P	98.65%	98.26%
R	99.98%	81.88%
F_1	99.31%	89.32%

Table 4.9 shows the average running time in milliseconds for the two algorithms to compute the distance between to samples, by varying the string length m. For space reasons, the table shows only 10 measurements.

Table 4.9: Average running time for *deq distance* and *longest subsequence distance*

m	*deq distance* (ms)	*longest subsequence distance* (ms)
100	0.066	0.024
200	0.139	0.102
300	0.208	0.212
400	0.274	0.415
500	0.336	0.612
600	0.422	0.979
700	0.539	1.362
800	0.556	1.838
900	0.697	2.552
1000	0.719	2.972

Figure 4.10 shows the charts with the running time for the full set of measurements. The charts confirm the theoretical result that the *deq distance* can be computed in linear time, while the *longest subsequence distance* requires quadratic time.

Although the *longest subsequence distance* is faster to compute for small strings ($m < 300$), for $m = 1000$ the *deq* distance becomes 4 times faster. The charts overlap in the region $250 \leq m \leq 300$, where the two distances obtained the same performance.

4.5.3 Experimental results on structured code

In this subsection we will evaluate both the quality and the performance for the techniques used to deal with structured code. We used a dataset consisting of 53 Android applications that were already clustered into 18 groups. The applications belonging to the same group are either different versions of the same application or real plagiarism cases. An ideal similarity measure will output high similarity values for all the pairs of applications that belong to the same group and low values for the applications pairs belonging to different groups.

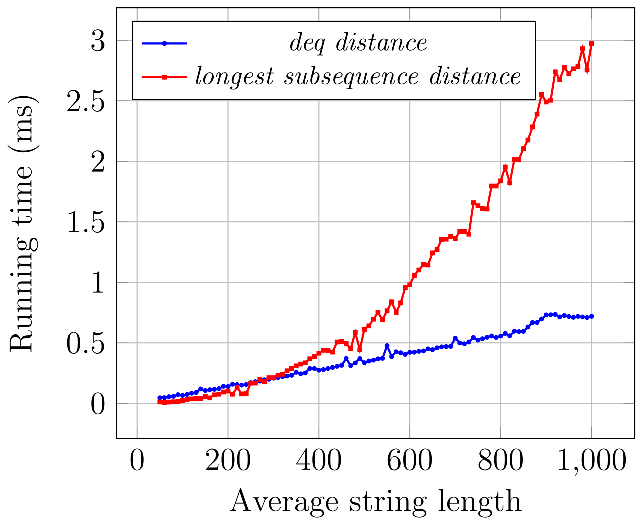


Figure 4.10: Performance evaluation for *deq distance* and *longest subsequence distance*

To emphasize the quality and the performance of the hierarchical similarity technique, we will also compare it with the flat model. The flat model assumes that no hierarchical information is known about applications and all the methods are situated in a single class.

4.5.3.1 Classification quality

For each of the $\frac{53(53-1)}{2} = 1378$ pairs of applications we computed the hierarchical similarity using both the thresholded and contiguous match strategy. By comparing the similarity result with a threshold, we can state if the two applications are similar or not. For each matching strategy, we will divide the pairs into the four categories of the confusion matrix (TP, FP, FN and TN) and compute the quality indices Precision, Recall and F-Measure.

If we selected the similarity threshold at 50%, we obtained the following values for the aforementioned quality indices:

- thresholded: $P = 100\%$; contiguous: $P = 100\%$;
- thresholded: $R = 78.33\%$; contiguous: $R = 68.33\%$;
- thresholded: $F_1 = 87.85\%$; contiguous: $F_1 = 81.18\%$;

The precision is at 100%, meaning that no pair of dissimilar items was classified as similar. The thresholded match strategy provided a higher recall than the contiguous match with about 10%, leading also to a higher F-Measure.

By decreasing the similarity threshold, we can increase the recall by decreasing the precision. At 25% threshold, the values are the following:

- thresholded: $P = 89.06\%$; contiguous: $P = 95\%$;
- thresholded: $R = 95\%$; contiguous: $R = 95\%$;
- thresholded: $F_1 = 91.93\%$; contiguous: $F_1 = 95\%$;

In this case, the precision is no longer 100%, meaning that some pairs of applications may be flagged as similar, even if they are not. However, the recall reached 95%, and the overall F-Measure is also higher.

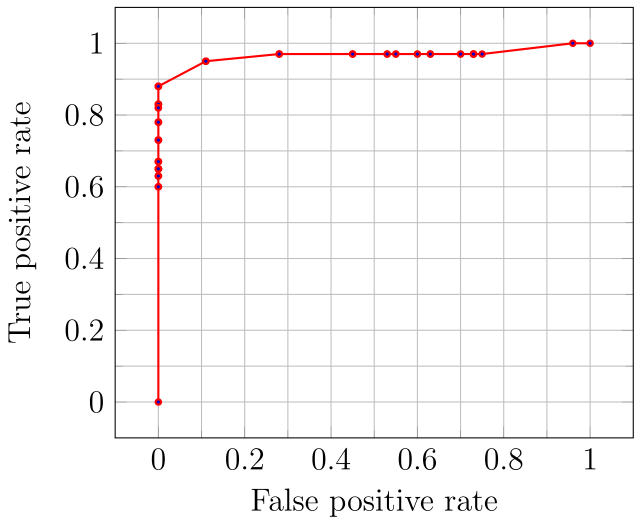


Figure 4.11: ROC curve for thresholded match

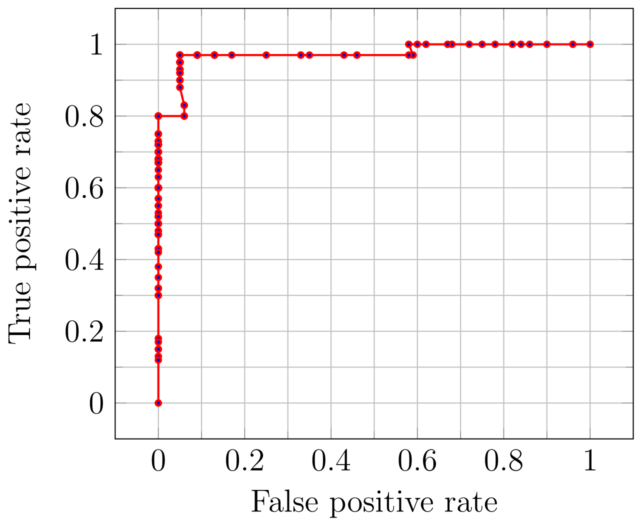


Figure 4.12: ROC curve for contiguous match

Figure 4.11 shows the ROC curve for the classifier that uses the thresholded match strategy, while Figure 4.12 shows the ROC curve for the classifier that uses the thresholded match strategy. We can observe that both curves are closer to the ideal $(0, 1)$ point than to the *line of no discrimination*, meaning that we managed to build good classifiers.

Table 4.10 contains the quality indices for the two hierarchical similarity strategies and for the flat similarity, for the closest to the ideal $(0, 1)$.

In Figure 4.11, this point is obtained for the similarity threshold 30%. The best point on the ROC curve in Figure 4.12 is obtained for the threshold 17%. This results show that using the contiguous match strategy in the similarity function leads to a slightly better classifier. Also, both strategies are better than the flat strategy, where the best point is obtained at threshold 40%.

Table 4.10: Quality indices for various similarity strategies

	P	R	F_1
thresholded	100%	88.33%	93.8%
contiguous	95.08%	96.66%	95.86%
flat	94.64%	88.33%	91.37%

4.5.3.2 Algorithm performance

Theoretical computations showed that Algorithm 3 has $O(n^2)$ complexity, where n is the average number of methods for the two applications. Figure 4.13 plots the running time of the algorithm against the number of methods.

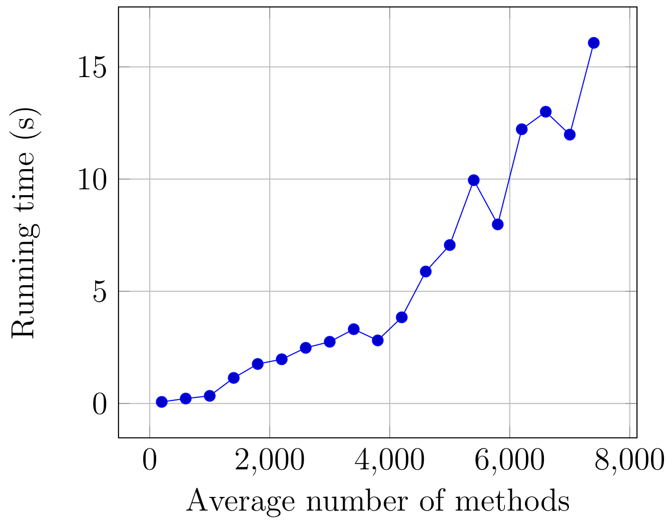


Figure 4.13: Performance measurement for Algorithm 3

The algorithm was implemented in Java and ran on an Intel i7 vPro processor at 2GHz with 8GB of RAM. For large applications of more than 7000 methods, the amount of time required to compute the similarity is a little more than 15 seconds, which make the method good for comparing individual applications but too slow to make every pairwise comparison on large collections.

For comparison, we have plotted in Figure 4.14 the performance of the algorithm in the case where all the methods are situated in the single class (flat similarity). In this case, for large applications with more than 7000 methods, the running time exceeds 5 minutes, which is 20 times worse than the original algorithm.

4.5.4 Experimental results on automated features selection

We have tested both the Genetic Algorithm and the PSO method from section 4.4, in order to find the best subset Λ.

We only obtained non-trivial solutions for large enough cleansets. If a small cleanset is used, we won't be able to proper identify the n-grams present in the real world clean samples. For instance, if we find some n-grams extracted from library code, that are common to a malicious cluster, we could falsely claim that we are able to detected that cluster. In reality, that library code is also common to clean samples and our detection

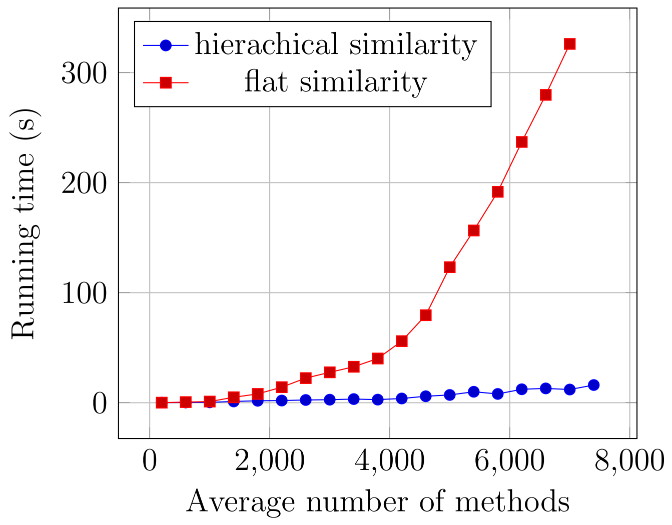


Figure 4.14: Performance measurement for Algorithm 3 assuming a flat model

will give false positives on them. With a small cleanset, those library-code n-grams could not be filtered out.

The experiments confirmed the statements above, as we couldn't find any better solution than $\Lambda = \Sigma$ for cleansets smaller than 10000 samples. For scenarios closer to real-life however, non-trivial solutions were found. The searches performed with the cleanset containing 558695 different methods from 55230 samples found better solutions that those found manually, based just on observation.

For the Genetic Algorithm, we have chosen a population size of 200, and it ran for 168 generations, searching for the subset Λ with the greatest fitness value. The evolution of the best fitness of the population for each generation can be observed in Figure 4.15. We can see that for the first generation, we have a lower score, that increases as the population evolves. After a while, the fitness score stagnates, meaning that it got stuck in a local optimum. After the best fitness score maintains the same value for too many generations, the population gets randomly re-initialized. At those points, we can see sudden drops on the chart, because random populations will usually perform worse than local optima.

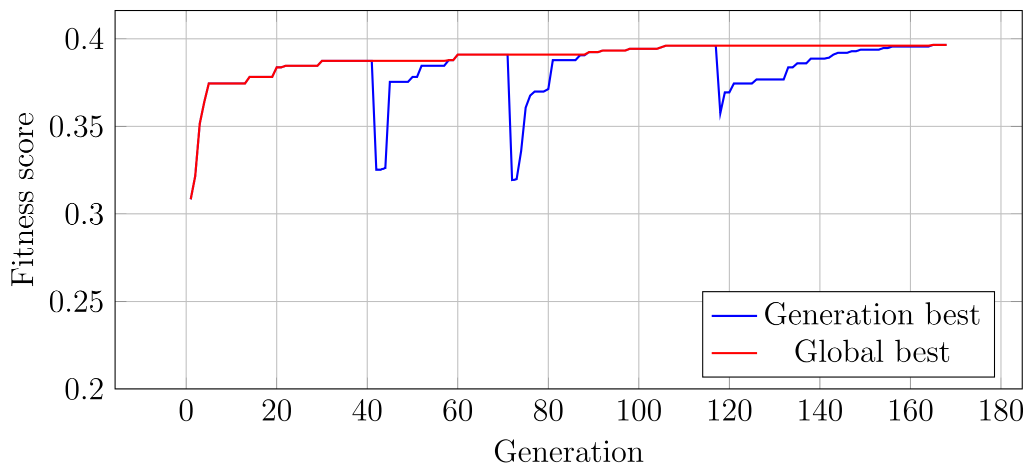


Figure 4.15: Evolution of the fitness function for the Genetic Algorithm

The maximum value reached during 168 generations was 0.3965. This fitness score means that best found solution was able to detect 39.65% of the clusters. Although this score seems low, we must remember that we had clusters of methods, not clusters of samples. Among the methods of a malware sample we can expect to find many similarities with methods from clean samples (not everything in a malware sample is malicious).

Particle Swarm Optimization got promising results with a smaller swarm, of only 25 particles. Since the fitness function was the performance bottleneck, we were able to run it for 665 generation, as we can see in Figure 4.16.

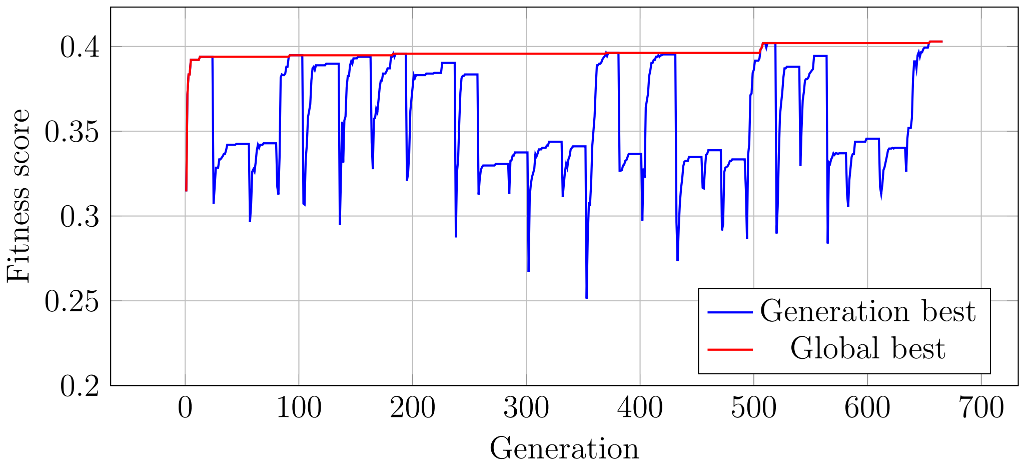


Figure 4.16: Evolution of the fitness function for the Particle Swarm Optimization

By looking at the figure, we can also see regions where the swarm evolves, followed by stagnation to a local optimum, followed by a random re-initialization. Having the chance to run for more generations, PSO found a slightly better solution than the Genetic Algorithm, with a fitness value of 0.4029 (40.29% of the clusters could be detected). The value falls subject to the same interpretations as the one obtained for the Genetic Algorithm.

After running these algorithms, we wanted to see how well they performed. The best solutions found by them were cross-validated on some new clusters. For training, we had two kinds of clusters: manually selected malware samples from "the wild" and obfuscated ones, using obfuscation tools. For cross-validation, these types will be separated, for a better understanding of the results. The fitness values for these are shown in Table 4.11.

Table 4.11: Cross-validation results

	GA best	PSO best
Similar malware samples	0.1819	0.1833
Obfuscated samples	0.8859	0.8859

The cross-validation experiments show that it is easy to learn how to bypass commercial obfuscators. Although the score for the malware methods from "the wild" was lower, it shows the probability for a group of methods to be detected, not for a group of entire samples. If we consider a sample to have m methods and the probability for a method to be detected is p, then the chance for each method to evade detection is $1 - p$. Since the probability to detect each method is independent from the others, the probability that all m methods evade detection is $(1 - p)^m$, so the detection probability for the sample is $P(detection) = 1 - (1 - p)^m$.

Considering $m = 20$ (it is reasonable to consider that a non-trivial .NET application has at least 20 methods) and $p = 0.1833 = 18.33\%$ (the cross-validation score obtained by the best solution found by PSO), we obtain $P(\mathit{detection}) = 1 - (1-p)^m = 0.9825 = 98.25\%$.

The calculated detection probability exceeds 98%, a score similar to the ones obtained by [SMF+12b] and [Bil07]. The papers mentioned above also studied malware detection using n-grams, but their research was focused on x86 OpCodes.

The Genetic Algorithm and the Particle Swarm Optimization ran for a couple of days in order to find reasonable solution for the subset Λ.

4.6 Chapter Conclusions

This chapter presented techniques for computing software similarity based on the features extracted from the binary code. By parsing the specific file format for different types of executables we can locate the code buffers and disassemble them in order to extract OpCode sequences. The OpCode sequences can be considered an abstract representation of a program's code and can be used to compute the distance between two programs.Several distance metrics were proposed. They can be categorized according to the program's abstract representation:

- string semantics: a program is abstractly represented as string of OpCodes
- set semantics: a program is represented as a set of OpCode n-grams (which can be considered abstract composite operations)
- hierarchical structures: the binary code can be split into packages, classes and methods

If a file is abstracted as a string of OpCodes, the fastest distance to compute is deq distance, because it is based on the longest common substring concept, rather than the longest common subsequence used for the edit distance. The improved performance didn't affect the quality of the distance, as it obtained similar results with the edit distance.

When we switch to set semantics, the weighted common n-grams distance outperformed other distance functions in terms of Precision and Recall and even obtained better results in identifying plagiarized student homeworks than Moss, a stat of the art plagiarism checker.

When extra information about the structure of a binary program is available, like the program partition into packages, classes and methods, hierarchical similarity can be used. The idea for computing the hierarchical similarity is to compare two packages by computing pairwise similarity between their components, then finding the best bipartite match using the Hungarian algorithm.

The last part of the research in this chapter was to select a subset of the OpCodes to be used for the n-grams extraction. Since the search space for such a subset is too big, we needed to optimize it so we can find a good solution in a reasonable amount of time. The Genetic Algorithm and the Particle Swarm Optimization are bio-inspired algorithms that are appropriate for such an optimization. They both found good solutions that were cross-validated to see how well we can detect new clusters.

The following contributions were presented in this chapter:

- the deq distance, a distance for binary programs based on string semantics;

- the weighted n-grams distance, a distance for binary programs based on set semantics;
- a hierarchical similarity algorithm, for dealing with structured code;
- a method for automatically selecting the most relevant OpCodes for constructing abstract program representations;
- a plagiarism detection system for students programming assignments;
- a detailed analysis of the .NET executables;

This chapter is based on the following published work:

- **Ciprian Oprișa**, George Cabău, and Adrian Coleșa. From plagiarism to malware detection. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 227–234, Timisoara, Romania, 2013. IEEE. ([OCC13])
- **Ciprian Oprișa**, George Cabău, and Gheorghe Sebestyen Pal. A new string distance for computing binary code similarities. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 91–96, Cluj-Napoca, Romania, 2014. IEEE. ([OCSP14b])
- **Ciprian Oprișa** and Nicolae Ignat. A measure of similarity for binary programs with a hierarchical structure. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, Cluj-Napoca, Romania, 2015. IEEE. ([OI15])
- **Ciprian Oprișa**, George Cabău, and Adrian Coleșa. Automatic code features extraction using bio-inspired algorithms. *Journal of Computer Virology and Hacking Techniques*, 10(3):165–176, 2014. ([OCC14])

Chapter 5

Scalable Clustering Algorithms

Large collections of binary programs can contain duplicates and near-duplicates. Depending on the nature of the collection we can identify plagiarism cases (if the collection is a market of applications) or different variations of a malware family (if we are dealing with malware collections). In either case, we want to group together the binary files that resemble each other. The problem can be solved by employing cluster analysis techniques but the classical algorithms run in quadratic or even cubic time, being unpractical for large collections. The bad news is that such algorithms like SLINK have also been proved to be optimal [Sib73], which means that we can't perform single linkage clustering in sub-quadratic time.

In order to improve the running time, we need to relax the requirements of the algorithm in one of the following ways:

- find particular distance metrics where computing pairwise distances is not necessary
- improve the running time by giving a very good (but inexact) approximation of the clusters

In this chapter, we will propose two approximate clustering algorithm that gives good results, while maintaining reasonable running times.

5.1 Clustering Using Suffix Trees

In the previous chapter we have described the *deq distance*, a distance that can be computed on sequences of OpCodes extracted from binary programs, based on the concept of Longest Common Substring. The distance formula is presented in Equation 4.9. Based on this distance, we also devised a similarity function in Equation 4.11, that outputs values in the $[0, 1]$ interval. This function computes the ratio between twice the longest common substring length and the sum of the lengths of the two given strings.

An even simpler similarity function can be designed if we consider all the strings to have the same length (if this is not the case we can get there by padding each string with a unique symbol). If we consider the length of all strings to be $\sigma \in \mathbb{N}$, the new similarity function will be:

$$\begin{gathered} s' : \Sigma^\star \times \Sigma^\star \to [0,1] \\ s'(S_1, S_2) = \frac{2|lcs(S_1, S_2)|}{|S_1| + |S_2|} = \frac{2|lcs(S_1, S_2)|}{2\sigma} = \frac{|lcs(S_1, S_2)|}{\sigma} \end{gathered} \tag{5.1}$$

We prefer to use the later similarity function s' (from Equation 5.1) because it only depends on the longest common substring. In other words, we will say that two samples are 60% similar if the longest common substring of their representation is 60% of the maximum string length.

After testing the deq distance, we have noticed some practical issues. When the strings correspond to code extracted from programs, some substrings might correspond to library code. For practical considerations, we will slightly modify the definition of the longest common substring, so it will only consider substrings that start with non-library code. We will see that the complexity of the following algorithms is not affected by this modification, while the practical results are dramatically improved.

To check if a string starts with library code or not, we can use a precomputed look-up table, where a hash on the first symbols from each *library code* string is marked.

5.1.1 The algorithm for computing the deq distance

The key for computing the deq distance, as defined in Equation 4.9 is the computation of the longest common substring. Several algorithms exists for solving this problem. Dynamic programming can be used for finding the longest common substring in $O(m_1 \cdot m_2)$ [ZCM07], where m_1 and m_2 are the lengths of the given strings. A faster approach for this problem uses the Suffix Tree data structure that was introduced by Weiner in 1973 [Wei73]. The construction of the Suffix Tree was improved by Ukkonen in 1995 [Ukk95]. Ukkonen's algorithm is linear in the size of the given string, so the cost for computing the longest common substring is $O(m_1 + m_2)$. Our work is based on this algorithm that is presented with more details in [Gus97].

Algorithm 9 HIGH-LEVEL-UKKONEN(S)

Require: A string S from $\Sigma^\star$ of length m
Ensure: A Suffix Tree T

```
1: $T \leftarrow$ INITIALIZE-SUFFIX-TREE()
2: for $i = 1 \to m - 1$ do
3:     for $j = 1 \to i + 1$ do
4:         $p \leftarrow$ FIND-PATH($T, S[j..i]$)
5:         EXTEND-PATH($T, p, S[i+1]$)
6:     end for
7: end for
8: return $T$
```

Algorithm 9 from [Gus97] builds the suffix tree given the string S. Although from the high level description, the complexity seems to be cubic (FIND-PATH is also $O(m)$), it is proven that by using some implementation tricks, the whole algorithm is linear in the size of the string ($O(m)$). The main loop (line 2) is still used, but the second loop (line 3) and the path search (the FIND-PATH function at line 4) can be avoided by using suffix links. These suffix links connect paths in the suffix tree, so there is no need to iterate through all the suffixes of the current substring in order to find the paths that need to be extended.

The function EXTEND-PATH called in line 5 is used to extend the path p from the tree T with the $(i+1)$-th character from the string S.

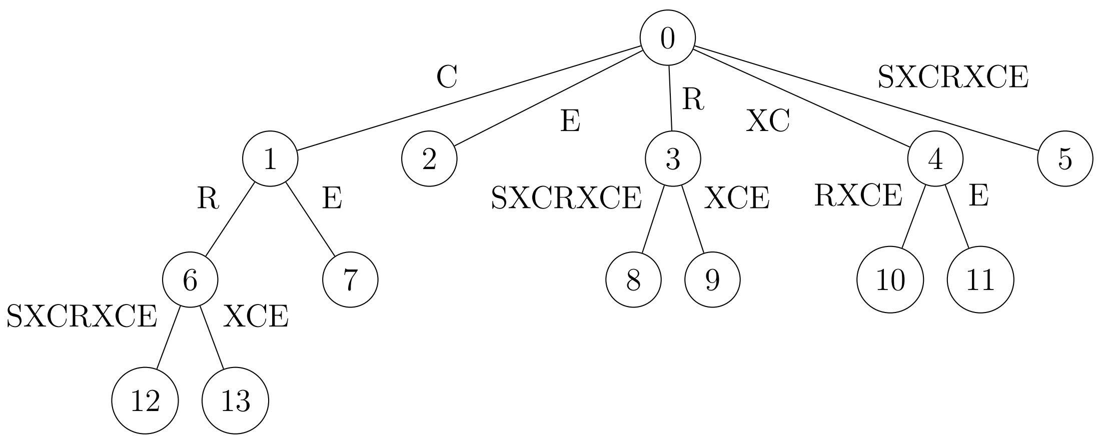


Figure 5.1: Suffix tree representation for string ”CRSXCRXCE”

The suffix tree constructed by Algorithm 9 for the input string ”CRSXCRXCE” is represented in Figure 5.1. Except for the root, each node in the suffix tree has an incoming edge labeled with a substring. If we called the function FIND-PATH on the example tree, for the substring ”CRXCE”, the nodes 0, 1, 6 and 13 would be returned. Since 13 is a terminal node, EXTEND-PATH does not involve creation of a new node. Instead, the next symbol can be appended to the substring on the edge 6-13. If we had the substring ”CRXC” instead, the edge 6-13 would be split, like in Figure 5.2. We can see that two new nodes were added: 14, to split the edge 6-13 and 15, to mark the suffix ”..XCS”.

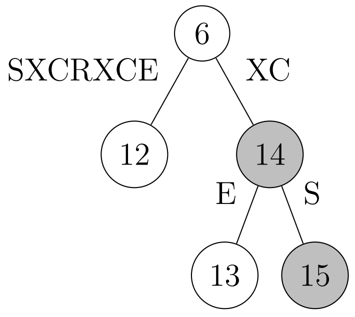


Figure 5.2: Splitting an edge in a suffix tree

We will adapt Algorithm 9 in order to compute the suffix tree for an arbitrary number of strings. The nodes in this tree will also contain a list with all the strings that contain the associated path.

The function MODIFIED-UKKONEN in Algorithm 10 receives an existing suffix tree T and extends it with all the suffixes from the new string S.

Two new symbols are added to the alphabet Σ: ’\$’ and ’#’. They are both used as terminal characters for the inserted strings in order to force all suffixes to end in a node. In Figure 5.3 we can see a portion of the built suffix tree for two strings. The first strings ends with ”ABCDE” and the second one with ”ABC”. The suffixes appear in both strings but in Figure 5.3a, the second suffix (”ABC”) is only implicit, as it is included in the edge 0-3. In Figure 5.3b, the terminating character ’\$’ is added to both strings. The edge 0-3 gets split by node 1. In this case, the substring ”ABC” also have an explicit representation. It is important to have such explicit representations for each substring, in order to add string references for all of them.

Gusfield recommends in [Gus97] “to append a different end of string marker to each

Algorithm 10 MODIFIED-UKKONEN(T, S, idx)

Require: An existing suffix tree T
Require: A string S from $\Sigma^\star$ of length m
Require: The index of the new string idx
Ensure: An updated suffix tree T

```
1: $S \leftarrow S +' \$'$
2: $endings \leftarrow \{\}$
3: for $i = 1 \rightarrow m - 1$ do
4:     for $j = 1 \rightarrow i + 1$ do
5:         $p \leftarrow$ FIND-PATH$(T, S[j..i])$
6:         for all $node \in p$ do
7:             ADD-STRING-REFERENCE$(T, node, idx)$
8:         end for
9:         EXTEND-PATH$(T, p, S[i + 1])$
10:        if $S[i + 1] =' \$'$ then
11:            $endings \leftarrow endings \cup \{$LAST-NODE$(p)\}$
12:        end if
13:    end for
14: end for
15: for all $node \in endings$ do
16:     REPLACE-ENDING$(T, node, ' \$', ' \#')$
17: end for
```

string in the set" in order to build a generalized suffix tree (a tree that contains suffixes for several strings). However, if we used this approach, a new symbol had to be added to the alphabet Σ for every string. Since we want the alphabet to have a constant size, we have used a slightly different technique.

Before insertion in the tree, we concatenate the symbol '\$' to the string to be inserted (line 1). The last nodes of each path that ends with '\$' are kept in the set *endings* (line 11). After the insertion, the symbol '\$' is replaced in all these nodes by '#' (line 16), ensuring that the following string to be inserted does not end with a symbol that already exists in the tree. For each node contained in a path associated with a substring from S, a reference to the current string index is added (line 7).

Using suffix links like in [Ukk95] and [Gus97], the implementation of the Algorithm 10 will also have $O(m)$ complexity.

Finally, the generalized suffix tree for a set of strings can be built by initializing an empty substring T, then calling MODIFIED-UKKONEN for each string S and increasing the current index.

Now, to compute the longest common substring, we will use the function LCS presented in Algorithm 11. The tree traversal is performed in Algorithm 12 by the function LCS-AUX.

The function LCS starts by building the suffix tree for the given strings. At line 5, the LCS-AUX function is called, in order to perform the tree traversal, starting with the root node that is situated at depth 0.

The LCS-AUX function considers only the nodes that have references to all the input strings (line 1). Also at this point, the node validation is performed. This validation is performed for practical issues, in order to avoid library code, as discussed at the end of

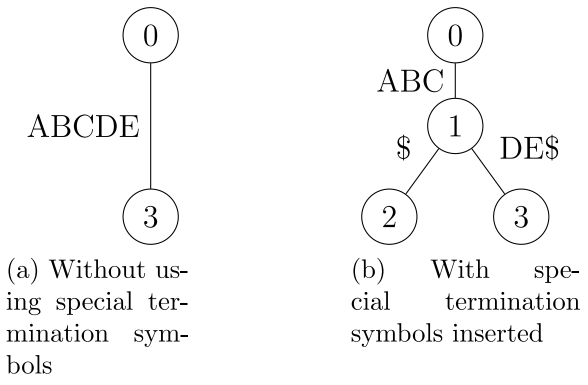

0
ABCDE
3
(a) Without using special termination symbols
0
ABC
1
$
DE$
2
3
(b) With special termination symbols inserted

Figure 5.3: Example of suffix trees with and without using the special terminating character

Algorithm 11 LCS(*strings*)

Require: *strings*, the set of strings to compute the longest common substring for
Ensure: The length of the longest common substring of all the strings from SS

```
1: $T \leftarrow$ INITIALIZE-SUFFIX-TREE()
2: for $i = 1 \rightarrow |strings|$ do
3:     MODIFIED-UKKONEN($T, strings[i], i$)
4: end for
5: return LCS-AUX($T.root, 0, |strings|$)
```

Algorithm 12 LCS-AUX(*crtNode*, *parentDepth*, *nrStr*)

Require: *crtNode*, the current node in a suffix tree
Require: *parentDepth*, the depth of the current node's parent
Require: The number of strings *nrStr*
Ensure: The length of the longest common substring

```
1: if $crtNode.nrRefs \neq nrStr$ or not IS-NODE-VALID($crtNode$) then
2:     return $-1$
3: end if
4: $crtDepth \leftarrow parentDepth +$ EDGE-LENGTH($crtNode$)
5: $maxLen \leftarrow crtDepth$
6: for all $node \in crtNode.children$ do
7:     $len =$ LCS-AUX($node, crtDepth, nrStr$)
8:     if $len > maxChild$ then
9:         $maxLen \leftarrow len$
10:     end if
11: end for
12: return $maxLen$
```

the previous subsection. If library code issues do not arise, this validation can be skipped.

For each valid node, the current depth is computed by adding the length of the current edge to the parent's depth (line 4). The function EDGE-LENGTH is used to compute the length of this edge. The length of an edge is the number of symbols on that edge. For example, in Figure 5.1, the edge 0-1 has the length 1, while the edge 6-13 has the length

3.

For each child of the current node, the function LCS-AUX is called recursively, in order to find the depth of deepest node, that corresponds to the longest common substring.

Since the number of nodes in the suffix tree is $O(n \times m)$ and the tree traversal in LCS-AUX reaches each node at most once, the algorithm complexity is $O(n \times m)$. For computing the longest common substring for only two strings, we have $n = 2$, so the algorithm complexity is $O(m)$ in this case.

5.1.2 Clustering Algorithms

In this subsection, we will show how the *deq distance* d (Equation 4.9) and the associated similarity function s' (Equation 5.1) can be used with classical clustering algorithms in order to compute the malware clusters in quadratic time and how these clusters can also be computed in linear time using the same Suffix Tree data structure presented in the previous section.

5.1.2.1 Single-link clustering in quadratic time

Given a measure of dissimilarity (like our *deq distance*), we can perform various clustering algorithms on a collection of items. The *single-link* approach, also called *nearest-neighbour* is a hierarchical clustering method, where two clusters are joined if the smallest dissimilarity between an item from the first cluster and an item from the second cluster doesn't exceed a given threshold. An algorithm called *SLINK* with time complexity $O(n^2)$ is given in [Sib73] and is proved to be optimal. This algorithm constructs a dendrogram that can be further used to select the clusters at any given threshold. However, if the threshold is given, we can compute the clusters more directly, although the complexity of the algorithm is also $O(n^2)$.

Our algorithm considers all the items to be nodes in a graph. Two nodes are connected if the distance between them doesn't exceed a threshold. The connected components will be the resulted clusters. Clustering algorithms based on graph theory were attempted before, mostly using the Minimum Spanning Tree data structure [GR69][XOX02][JN09]. We won't use this data structure, instead we will compute the connected components of the graph using the disjoint-set forest data structure [Tar75][CLR+01].

The disjoint-set forest is a data structure used to keep a collection of disjoint sets. Each set can be identified by a representative, that is one of the set members. The following operations are permitted:

- MAKE-SET(x): creation of a new set with a single element
- UNION(x, y): unification of the two sets containing x and y
- FIND-REPR(x): finding the representative member of the set containing x

The amortized analysis in [CLR+01] proves that using two heuristics, *union by rank* and *path compression* the cost for running k operations on n elements is $O(k\ \alpha(n))$, where α is the inverse of the Ackermann function [Ack28], which is at most 4, for all practical purposes.

A quadratic method for performing cluster analysis on a set of samples is presented in Algorithm 13. The algorithm receives the samples as an array of strings. Two samples will receive the same cluster id if their similarity is greater than a given threshold θ.

Algorithm 13 QUADRATIC-CLUSTERING$(samples, \theta)$

Require: A list of strings *samples*
Require: A similarity threshold θ
Ensure: An array *clusters* of the same size with *samples* that associates each string with a cluster id

```
 1: for $i = 1 \to |samples|$ do
 2:     MAKE-SET$(i)$
 3: end for
 4: for $i = 1 \to |samples| - 1$ do
 5:     for $j = (i + 1) \to |samples|$ do
 6:         $rep_i \leftarrow$ FIND-REPR$(i)$
 7:         $rep_j \leftarrow$ FIND-REPR$(j)$
 8:         if $rep_i \neq rep_j$ then
 9:             $sim \leftarrow s'(samples[i], samples[j])$
10:             if $sim \geq \theta$ then
11:                 UNION$(rep_i, rep_j)$
12:             end if
13:         end if
14:     end for
15: end for
16: for $i = 1 \to |samples|$ do
17:     $clusters[i] \leftarrow$ FIND-REPR$(i)$
18: end for
19: return $clusters$
```

The algorithm starts by placing each element in a distinct cluster (a new disjoint set). For each pair of samples, we first verify if they belong to the same set or not. If they don't belong to the same set, the similarity between them is computed and if it is at least θ, their sets are joined by the UNION operation. In the end, each sample receives as a cluster number the index of the representative.

If we consider $n = |samples|$, the operations performed on the disjoint-set forest are:

- n operations for initialization (line 2).
- 2 FIND-REPR operations for each pair of samples (lines 6-7). Since we have $\frac{n(n-1)}{2}$ pairs, the total number of such operations is $n(n-1)$.
- At most $n-1$ UNION operations (line 11). We only perform UNION on items that do not belong to the same cluster, so it's like building a spanning tree.
- n FIND-REPR operations for assigning the items to the clusters (line 17).

The total number of operations performed on the disjoint-set forest is $n + n(n-1) + n - 1 + n = n^2 + 2n - 1$. Given the amortized analysis in [CLR+01], the complexity for these operations is $O(n^2\alpha(n))$.

The number of similarity computations is also $O(n^2)$ in the worst case (line 9). If the average string length for a sample is m, the complexity here will be $O(n^2\ m)$, since the deq distance is computed in $O(m)$.

Although our goal is to compute the clusters on a very large number of samples and m can be considered constant, for all practical purposes $m \gg \alpha(n)$. The final complexity of the algorithm will be then $O(n^2 \times m)$ instead of $O(n^2\alpha(n))$.

5.1.2.2 Single-link clustering in linear time

The previous method is slow because it has to compute the longest common substring for each pair of strings. In this subsection, we will build a single suffix tree for all the given strings and compute the clusters in linear time for the average case.

Algorithm 14 traverses all the nodes in the generalized suffix tree constructed from all the input strings. Nodes deeper than $\theta \cdot \sigma$ are associated with common substrings longer than $\theta \cdot \sigma$, so the similarity between each pair of string references is at least θ. For each node situated under this depth and with more than one string reference, all the references are joined in the same cluster.

Algorithm 14 LINEAR-CLUSTERING($samples, \theta$)

```
Require: A list of strings $samples$
Require: A distance threshold $\theta$
Ensure: An array $clusters$ of the same size with $samples$ that associates each string
    with a cluster number
 1: for $i = 1 \to |samples|$ do
 2:     MAKE-SET($i$)
 3: end for
 4: $T \leftarrow$ INITIALIZE-SUFFIX-TREE()
 5: for $i = 1 \to |samples|$ do
 6:     MODIFIED-UKKONEN($T, samples[i], i$)
 7: end for
 8: for all $node \in T$ do
 9:     if $node.depth \geq \theta \cdot \sigma$ and IS-NODE-VALID($node$) then
10:         for $i = 2 \to node.nrRefs$ do
11:             $rep_1 \leftarrow$ FIND-REPR($node.refs[1]$)
12:             $rep_2 \leftarrow$ FIND-REPR($node.refs[i]$)
13:             if $rep_1 \neq rep_2$ then
14:                 UNION($rep_1, rep_2$)
15:             end if
16:         end for
17:     end if
18: end for
19: for $i = 1 \to |samples|$ do
20:     $clusters[i] \leftarrow$ FIND-REPR($i$)
21: end for
22: return $clusters$
```

The algorithm starts by initializing the disjoint-set forest (lines 1-3) and the suffix tree (line 4). The suffix tree is then added all the samples in lines 5-7 in the same way as in Algorithm 11. The tree traversal is then initiated at line 8. Although the actual implementation uses a recursive traversal like the LCS-AUX function, we will consider it implicit and just iterate through all the tree nodes.

The substring validation also appears, like in Algorithm 12 and is only necessary if we want to avoid library code.

For each valid node with a depth greater than $\theta \cdot \sigma$ and with at least two string references, all the string references must be joined in the same cluster. For this, we compute the representative of the disjoint set of the first string (line 11) and of the every other string in the list of references (line 12). If the two representatives are not in the same disjoint set, the UNION operation is performed (line 14).

Finally, each sample is assigned a cluster number based on the set representative, like in Algorithm 13 (line 20).

The complexity of this algorithm is given by the suffix tree building ($O(n \times m)$) and the tree traversal for finding the clusters. The suffix tree has at most $O(n \times m)$ nodes and for each node, the list of string references must be iterated. In the worst-case scenario, this list can contain all the strings, being $O(n)$ long. For the average case, we can consider that the length of this list is γ, a constant that also expresses the average cluster size. For each item in the list of string references except for the first one, the FIND-REPR function is called two times, for an amortized cost of $O(\alpha(n))$. The UNION function is called at most $n - 1$ times the whole algorithm.

The final complexity of this algorithm is then $O(n \times m \times \gamma \times \alpha(n))$. Although for the worst-case scenario the algorithm is still quadratic (γ can be $O(n)$) like Algorithm 13, for the average case γ and $\alpha(n)$ are constant so the complexity is only $O(n \times m)$. The linearity of this algorithm is also confirmed by the experimental evaluation.

5.2 Clustering Using Locality-Sensitive Hashing

Locality-sensitive hashing is a technique introduced in 1998 by Indyk and Motwani [IM98] in order to find similar items in a multi-dimensional space. The basic idea is to use a family of hash functions where the collision probability for two hashes is equal to the similarity between their preimages. The ability of this technique to find similar items in a large collection was used in [KIW07] for performing agglomerative hierarchical clustering.

In this section, we will also apply the locality-sensitive hashing technique for cluster analysis, but we will focus on clustering binary programs.

An executable program can be represented as a set of n-grams, as described in the previous chapter. The Jaccard similarity computes the ration between the number of common n-grams and the total number of n-grams for two samples. Our clustering algorithm will group together in the same cluster the samples whose similarity exceeds a given threshold. We will use the single linkage approach [Sib73] but we will show that a very good approximation of the clusters can be obtained much faster with locality sensitive hashing.

Previous work exists both in the field of LSH optimizations and improving the speed of malware clustering. In [SLH12], Slaney proposes a method for choosing the optimal parameters for LSH, in order to improve the speed of searching for similar items. As in our approach, the data distribution proves to be very important in choosing these parameters.

Our efforts mainly consisted in applying the locality-sensitive hashing technique for clustering and proposing a method for selecting the best parameters for it.

5.2.1 Locality-sensitive hashing theory

This subsection will present the theory behind locality-sensitive hashing as explained in [IM98], [RU12] and [SC08]. The description was adapted in order to fit our particular problem, clustering malware samples.

The last part of the subsection adds some original work to the ideas in [RU12], by showing how minhash functions can be computed efficiently.

5.2.1.1 Locality-sensitive hashing definition

Generically speaking, a *hash* is a function that maps arbitrary-length data to fixed-length values. Common areas where hash functions are used are data transmissions, where hashes assure data integrity, databases where hashes are used for indexing and cryptography where hashes are used to secure the data. In all these areas, the hash functions are designed in such a way to minimize the probability of a collision (situation where the result of the hash function is the same for different inputs).

Locality-sensitive hash functions have a slightly different goal: to have a high probability of collision on similar items. To specify these functions in more formal terms, we'll use the definition from [RU12]. Let's define a set $\mathcal{S}$ and a distance function d that computes the distance between two items of the set:

$$d : \mathcal{S} \times \mathcal{S} \to [0, \infty) \tag{5.2}$$

For $X, Y \in \mathcal{S}$, $d(X, Y)$ will be a real number that represents the distance between X and Y. The distance is 0 only if the two items are identical. Given this distance d, we can formally define a locality-sensitive hash function on $\mathcal{S}$.

Definition 1. *A function $h : \mathcal{S} \to \mathcal{H}$ is called a (d_1, d_2, p_1, p_2)-sensitive hash function if $\mathcal{H}$ is a finite set, $d_1 < d_2$ and the following properties hold for $X, Y \in \mathcal{S}$:*

- *If $d(X, Y) \leq d_1$, then $P(h(X) = h(Y)) \geq p_1$.*
- *If $d(X, Y) \geq d_2$, then $P(h(X) = h(Y)) \leq p_2$.*

What Definition 1 states is that for two items whose distance is smaller than d_1, the probability of hash collision is at least p_1, while for two items whose distance is greater than d_2, the probability of hash collision is smaller than p_2.

5.2.1.2 Jaccard distance and the minhashes

In the previous section, we have stated that we are working with executable programs that will be represented as sets of features. A sample X will be represented as the set $X = \{x_1, x_2, \ldots x_k\}$, where $x_1, x_2, \ldots x_k$ are natural numbers representing the identifiers for the features that are set. If all the possible features are given identifiers from 0 to $m - 1$, then each sample will be represented as a subset of the set $M = \{0, 1, \ldots m - 1\}$. The set $\mathcal{S}$ will be the set containing all the subsets of M, $\mathcal{S} = \mathcal{P}(M)$.

The distance between two samples $X, Y \in \mathcal{S}$ will be computed using the Jaccard similarity, as in Equation 5.3.

$$d_J(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|} \tag{5.3}$$

This distance is a real number between 0 and 1. If two sets are identical, $X \cap Y = X \cup Y$, so the distance will be 0. If the sets have no common element, $|X \cap Y| = 0$, so the distance will be 1. It can be proven that d_J is a proper metric, meaning that all the metric properties hold for d_J.

Given a permutation $\sigma : M \to M$, we can define a hash function as in Equation 5.4. Such a hash function will be called a *minhash*.

$$\begin{gathered} h : \mathcal{S} \to M \\ h(X) = \min_{x \in X} \sigma(x) \end{gathered} \tag{5.4}$$

The following result is proved in [RU12]:

Lemma 1. *If $h : \mathcal{S} \to M$ is a minhash function, and $d_J : \mathcal{S} \times \mathcal{S} \to [0,1]$ is the Jaccard distance, then the probability that $h(X) = h(Y)$ is equal with $1 - d_J(X,Y)$, for $X, Y \in \mathcal{S}$.*

Using Lemma 1, the following theorem can be proved:

Theorem 2. *If $h : \mathcal{S} \to M$ is a minhash function, then h is $(d_1, d_2, 1-d_1, 1-d_2)$-sensitive.*

5.2.1.3 The banding technique

Minhash functions are used for finding similar items because the probability of two items having the same minhash is equal to their similarity. However, we cannot use a single minhash function because many pairs might be missed. The banding techniques considers two parameters b and r, both natural numbers greater than 0. Instead of one minhash function, $b \times r$ functions are used. For a given sample, the results of these hash functions are grouped in b bands, each having r rows, as illustrated in Figure 5.4.

The $b \times r$ minhash functions will be grouped in a matrix, $h_{ij} : \mathcal{S} \to M$ being the minhash situated in the band i, on the j-th row.

The first r hash results will be $h_{11}(X), h_{12}(X), \ldots, h_{1r}(X)$, the next r will be $h_{21}(X), h_{22}(X), \ldots, h_{2r}(X)$ and the last r hash results will be $h_{b1}(X), h_{b2}(X), \ldots, h_{br}(x)$. For each band i, we will also compute another hash function on the hash results situated in that band. H_i will be a classical hash function, applied to the group of r hash results on the band i, concatenated: $H_i(X) = H(h_{i1}(X)h_{i2}(X)\ldots h_{ir}(X))$. The function H can be MD5, SHA-1 or any other hash function with a negligible chance of collision.

For every band i, we will store a group of buckets, each containing all the samples $Y \in \mathcal{S}$ that have the same hash value ($H_i(Y) = H_i(X)$) on that band.

For a given sample X, in order to find similar samples without comparing it with all the samples in the collection, all we have to do is to compute the b hashes $H_1(X), H_2(X), \ldots, H_b(X)$ and compare our sample only with the other samples situated in the b buckets that it falls.

We will now compute the probability that a pair of samples X and Y whose Jaccard distance is d have at least one common bucket.

We start with Lemma 1. If the $d_J(X,Y) = d$, then $\forall(i,j)$ with $1 \le i \le b, 1 \le j \le r$, we have that $P(h_{ij}(X) = h_{ij}(Y)) = d$. Since the probability of a hash collision for the H_i functions is negligible, in order to have $H_i(X) = H_i(Y)$, we need to have equality between

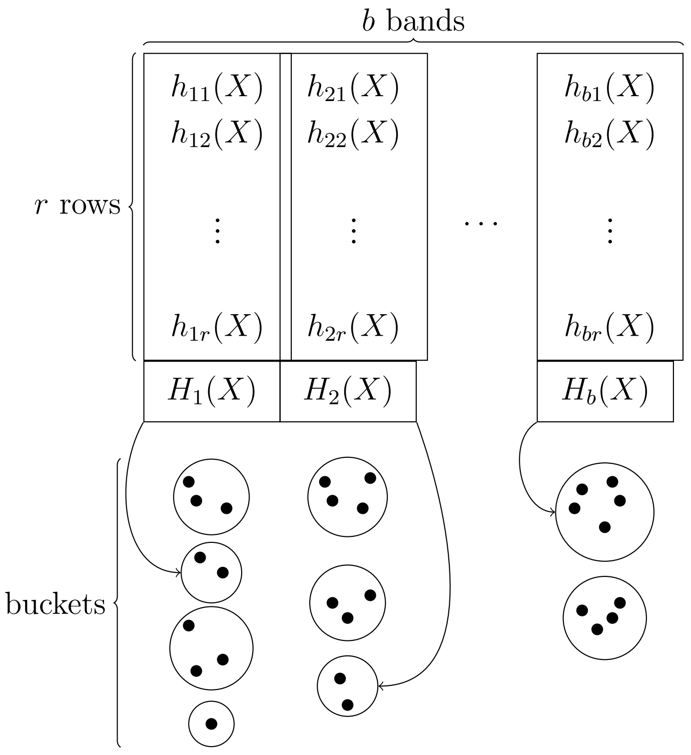


Figure 5.4: Banding technique illustration

all the hashes on that band. In other words:

$$\begin{aligned}
&P(H_i(X) = H_i(Y)) \\
&\quad = P(h_{i1}(X) = h_{i1}(Y) \wedge \ldots \wedge h_{ir}(X) = h_{ir}(Y)) \\
&\quad = P(h_{i1}(X) = h_{i1}(Y)) \cdot \ldots \cdot P(h_{ir}(X) = h_{ir}(Y)) \\
&\quad = (1-d) \cdot (1-d) \cdot \ldots \cdot (1-d) \\
&\quad = (1-d)^r
\end{aligned}$$

We were able to write the probability that all events occurred as a product because they are independent.

Now, we want the probability to have a hash collision on at least a band:

$$\begin{aligned}
P_c(X,Y) &= P(H_1(X) = H_1(Y) \vee \ldots \vee H_b(X) = H_b(Y)) \\
&= 1 - P(\overline{H_1(X) = H_1(Y) \vee \ldots \vee H_b(X) = H_b(Y)}) \\
&= 1 - P(\overline{H_1(X) = H_1(Y)} \wedge \ldots \wedge \overline{H_b(X) = H_b(Y)}) \\
&= 1 - P(\overline{H_1(X) = H_1(Y)}) \cdot \ldots \cdot P(\overline{H_b(X) = H_b(Y)}) \\
&= 1 - (1-(1-d)^r) \cdot \ldots \cdot (1-(1-d)^r) \\
&= 1 - (1-(1-d)^r)^b
\end{aligned}$$

By varying d from 0 to 1, we obtained the chart in Figure 5.5. On the X axis the samples distance d was varied from 0 to 1 and we represented $P_c(X,Y)$ (the probability to have a collision on at least one band) on the Y axis. The parameters b and r were set to 20 and 4.

We can see that the collision probability is very high if the samples have a small distance (99.58% if the distance is 0.3) and very small if the distance is big (15.01% if the distance is 0.7).

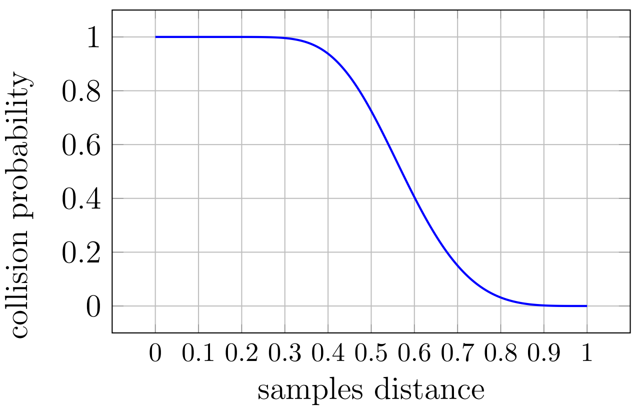


Figure 5.5: Probability to have at least a collision depending on the samples distance

5.2.1.4 Computing the minhashes efficiently

A minhash, as defined in Equation 5.4 can be computed by using a permutation σ. In order to generate quality minhash functions, we will need to be able to generate random permutations on the set M. Unfortunately, working with random permutations on large sets in computationally unfeasible. However, we can use a different approach.

In [RU12] it is claimed that a true permutation is not really necessary and functions of the form $(a \cdot x + b) \mod m$ are used instead. Although these functions will "map some pairs of integers to the same bucket and leave other buckets unfilled", the only conditions are that m is large enough and there are not too many collisions. However, we will show that such functions are true permutations if m is prime.

Because M contains all the elements from 0 to $m-1$, it is isomorphic with the set of all congruence classes of the integers for the modulus m, $\mathbb{Z}_m$. If m is a prime number, we can define a function $\sigma : \mathbb{Z}_m \to \mathbb{Z}_m$ as in Equation 5.5, with $1 \leq a \leq m-1$ and $0 \leq b \leq m-1$.

$$\sigma(x) = a \cdot x + b \tag{5.5}$$

Lemma 2. *If m is prime, then σ is a permutation.*

Proof. We will first prove that σ is injective:

If $\sigma(x) = \sigma(y)$, then $a \cdot x + b = a \cdot y + b$. By subtracting b we obtain $a \cdot x = a \cdot y$. Since m is prime, $\mathbb{Z}_m$ is a finite field, so there exists an inverse for each element. We will multiply both parts with a^{-1} and will obtained that $x = y$.

Since $\sigma(x) = \sigma(y) \to x = y$ it follows that σ is injective. σ is also surjective since its domains and codomain are finite and have the same size.

Being injective and surjective, σ is a bijective function and since $\mathbb{Z}_m$ is finite, σ is a permutation. □

From Lemma 2, we obtained that instead of generating and storing a random permutation, it suffices to generate two numbers a and b if m is prime. If the total number of features is not a prime number, we can set m to be the smallest prime larger than that number and add a couple of extra features that will never be set.

5.2.2 The clustering algorithm

Like in the previous section, based on Suffix Trees, our algorithm considers all the items to be nodes in a graph and computes the connected components using the disjoint-set forest data structure [Tar75][CLR+01].

To use the disjoint-set forest to compute the connected components of a graph, we need to traverse the list of edges and perform the UNION operation on every pair of vertices connected by an edge. This UNION will be performed only if the two vertices don't already belong to the same component. Since we need to find the set for every two vertices and perhaps perform a UNION, the total number of operations will linearly depend on the number of edges. The problem is that in order to discover all the edges in the graph of malware samples, it is necessary to compute the Jaccard distance between every pair of samples.

By using the locality-sensitive hashing technique, we will try to reduce the number of distance computations, while giving a good approximation of the clusters. The first part of the algorithm will use the banding technique from subsection 5.2.1.3 to place all the samples in a group of buckets. Then, we will only compute the distance between samples situated in the same bucket.

5.2.2.1 Algorithm presentation

Algorithm 15 performs the first part of the clustering task, the construction of the buckets. The buckets are stored as an array of b dictionaries, one for each band. The dictionary (a data structure that associates keys to values) maps the hash on a given band to the list of files that have the same hash on that band.

Algorithm 15 BUILD-BUCKETS($samples, b, r$)

```
Require: An array of samples $samples$
Require: The number of bands $b$ and the number of rows $r$
Ensure: $buckets$, an array of $b$ dictionaries
 1: for $i = 1 \to b$ do
 2:     $buckets[i] \leftarrow$ EMPTY-BUCKETS()
 3: end for
 4: for $k = 1 \to |samples|$ do
 5:     for $i = 1 \to b$ do
 6:         $bandHash \leftarrow$ INIT_VALUE
 7:         for $j = 1 \to r$ do
 8:             $hVal \leftarrow h_{ij}(samples[k])$
 9:             HASH-UPDATE($bandHash, hVal$)
10:         end for
11:         ADD-TO-BUCKET($buckets[i], bandHash, k$)
12:     end for
13: end for
14: return $buckets$
```

The algorithm starts by initializing the array of dictionaries (lines 1-3). Then, for each sample, the $b \times r$ minhash values are computed, and for each band, the hash value $H_i(samples[k])$ is computed. This hash value, stored in the variable *bandHash* is first initialized for each band (line 6), then updated with the computed minhash value, for each row, using the function HASH-UPDATE (line 9). Having the *bandHash* computed, a reference to the current sample is added to the bucket containing all the files that have the same hash value on the current band (line 11). The called function (ADD-TO-BUCKET) takes as parameters the dictionary for the current band, the computed hash value that will be used as a key in the dictionary and the reference to the current sample, which will be added in the bucket.

Algorithm 16 is used in the second part of the clustering. It takes as input one bucket, which is an array of sample references that had a hash collision on at least one band. Each pair of samples is considered and if two samples have a Jaccard distance smaller than the threshold θ, their disjoint sets are joined.

Algorithm 16 PARSE-BUCKET($bucket, samples, \theta$)

```
Require: An array of sample references, $bucket$
Require: An array of samples $samples$
Require: A distance threshold $\theta$
Ensure: The disjoint sets for similar samples in the bucket will be grouped together
 1: for $i = 1 \to |bucket| - 1$ do
 2:     for $j = (i+1) \to |bucket|$ do
 3:         $i_1 \leftarrow bucket[i], j_1 \leftarrow bucket[j]$
 4:         $rep_i \leftarrow$ FIND-REPR($i_1$)
 5:         $rep_j \leftarrow$ FIND-REPR($j_1$)
 6:         if $rep_i \neq rep_j$ then
 7:             if not CACHE-SEARCH($i_1, j_1$) then
 8:                 $d \leftarrow d_J(samples[i_1], samples[j_1])$
 9:                 if $d \leq \theta$ then
10:                     UNION($rep_i, rep_j$)
11:                 else
12:                     CACHE-INSERT($i_1, j_1$)
13:                 end if
14:             end if
15:         end if
16:     end for
17: end for
```

The most costly operation here is computing the distance between two samples. In theory, the FIND-REPR and the UNION operations should be more costly since they have an amortized complexity of $O(\alpha(n))$, while computing a distance does not depend on n, the total number of samples. However, $\alpha(n)$ can take only small values (at most 4, in any reasonable case), while computing the Jaccard distance means finding the intersection and the union of two sets, which takes more time.

Because of this cost, we will try to avoid computing this distance when unnecessary. First of all, if two samples are already in the same cluster (or set), we don't have to compute their distance anymore (we check this in line 6). Also, since the same pair of samples may be encountered in different buckets, we will keep a global cache with all the pairs for which the distance was already computed and proved to be greater than the threshold θ. Each pair will be first searched in the cache (line 7) and only if it's not found, the distance will be computed (line 8). If the distance that was just computed is greater than θ, the pair is cached so we won't have to compute this distance anymore.

Using the BUILD-BUCKETS function in Algorithm 15 and the PARSE-BUCKET function in Algorithm 16, we can finally build the locality-sensitive hashing clustering in Algorithm 17.

```
Algorithm 17 LSH-CLUSTERING($samples, \theta, b, r$)
Require: An array of sets $samples$
Require: A similarity threshold $\theta$
Require: The number of bands $b$ and the number of rows $r$
Ensure: An array $clusters$ of the same size with $samples$ that associates each string
    with a cluster id
 1: for $i = 1 \to |samples|$ do
 2:     MAKE-SET($i$)
 3: end for
 4: $buckets \leftarrow$ BUILD-BUCKETS($samples, b, r$)
 5: for all $bucket \in buckets$ do
 6:     PARSE-BUCKET($bucket, samples, \theta$)
 7: end for
 8: for $i = 1 \to |samples|$ do
 9:     $clusters[i] \leftarrow$ FIND-REPR($i$)
10: end for
11: return $clusters$
```

The algorithm starts by creating a separate set for each sample (line 2). The sets corresponding to similar samples will be joined in the PARSE-BUCKET function, in order to compute the clusters. In the first phase, the BUILD-BUCKETS function is called, in order to place the samples into buckets (line 4). In the second phase, all the buckets are parsed (line 6) in order to find similar samples and join their sets. Finally, a cluster id is assigned to each sample (line 9). The cluster id will be the set representative, found through the function FIND-REPR.

5.2.2.2 Algorithm performance

Algorithm 15, which performs the first phase of the clustering, computes $b \times r$ minhashes for each sample. The time to compute a minhash depends on the number of features set (because we need to iterate them and find the minimum value based on the permutation discussed above. We will consider this time a constant t_1, because it does not depend on the number of samples n. The dictionary insertion performed by the function ADD-TO-BUCKET (line 11) will also be considered to take constant time, since the dictionary can be implemented as a hash table. The approximative execution time of this phase of the algorithm will then be $t_1 \cdot b \cdot r \cdot n$.

The performance of the second phase of the algorithm is harder to estimate, since the appartenance to the same set for a pair of samples largely depends on the samples distribution. To simplify the analysis, we will consider that the representatives verification in line 6 of Algorithm 16 will mostly fail, so we can give an upper bound of the algorithm running time. Still, we will consider the caching mechanism, so we won't have to compute the distance between the same pair of samples more than once. In this case, the performance of the second phase of the algorithm is reduced to computing the Jaccard distance for a fraction of the samples pairs. The time required to compute such a distance will also be a constant t_2, since it only depends on the number of set features for each sample. Since the number of pairs for n samples is $\frac{n(n-1)}{2}$, the running time of the second phase of the algorithm will be $t_2 \cdot f(b,r) \cdot \frac{n(n-1)}{2}$.

All that is left to do is to estimate $f(b,r)$, the fraction of the sample pairs that we are required to compute the distance for.

Let $p : [0,1] \to [0,1]$ be the distribution of the distances between the pairs of samples on the given collection. $p(d)$ will be the estimated percentage of the pairs whose distances are d. In the banding technique description we have shown that for such a pair whose distance is d, the probability that the two samples share at least a common bucket (has a hash collision on at least a band) is $1-(1-(1-d)^r)^b$. This means that for a given distance d, the number of pairs that we will need to compute the distance for is $p(d) \cdot (1-(1-(1-d)^r)^b)$.

The total fraction of the samples required will then be:

$$f(b,r) = \int_0^1 p(x) \cdot (1-(1-(1-x)^r)^b)dx \tag{5.6}$$

Finally, having a formula for f, we can estimate the upper bound for the running time of the Algorithm 17, in Equation 5.7:

$$T(n,b,r) = t_1 \cdot b \cdot r \cdot n + t_2 \cdot f(b,r) \cdot \frac{n(n-1)}{2} \tag{5.7}$$

The bad news is that the complexity is still $O(n^2)$, since the dominant term of the expression is quadratic. However, we will show further how to make this coefficient as small as possible.

5.2.2.3 Clusters quality

In order to asses the clusters quality, we will use an external evaluation strategy. For the same group of samples, we can compute the clusters using the SLINK algorithm [Sib73]. Then, for each pair of samples we can see if they were placed in the same cluster or not by the two algorithms.

- If the two samples are placed in the same cluster by both algorithms we will say we have a True Positive. The number of True Positives will be denoted by TP.
- If the two samples were not placed in the same cluster by either algorithm, we will have a True Negative.
- If the samples were placed in the same cluster by LSH-CLUSTERING but not by SLINK, we will have a False Positive. Their number will be denoted by FP.
- If the samples were placed in the same cluster by SLINK but not by LSH-CLUSTERING, we will have a False Negative. Their number will be denoted by FN.

From these numbers, we can compute the Precision (P) and Recall (R) indices.

Since our algorithm will join sets only after computing the Jaccard distance, there will be no False Positive. If $FP = 0$, then from the Precision formula ($P = \frac{TP}{TP+FP}$) $P = 1$, so we have maximum precision. Still, we might have some False Negatives, since there might be similar samples with no hash collisions.

By using the banding technique, we have that the probability for two samples with distance $d = d_J(X, Y)$, $d \leq \theta$ to have at least a hash collision is $P_c(X, Y) = 1 - (1 - (1 - d)^r)^b$. Even if such a hash collision does not exist, there might exist a sample Z, such that $d_J(X, Z) \leq \theta$, $d_J(Y, Z) \leq \theta$ and both pairs (X, Z) and (Y, Z) have hash collisions. In this case, the single linkage approach will ensure that X and Y fall into the same cluster.

The Recall, as the fraction of relevant instances retrieved will then be approximated with this probability that a collision exists, considering the distance between samples to be θ, so $R \approx 1 - (1 - (1 - \theta)^r)^b$. If we want the Recall to be above a certain threshold, we can set a maximal admissible error ϵ and add the constraint $R \geq 1 - \epsilon$.

In this case, we have:

$$\begin{aligned} 1 - (1 - (1 - \theta)^r)^b &\geq 1 - \epsilon \\ \iff (1 - (1 - \theta)^r)^b &\leq \epsilon \\ \iff b \cdot \log(1 - (1 - \theta)^r) &\leq \log(\epsilon) \\ \iff b &\geq \frac{\log(\epsilon)}{\log(1 - (1 - \theta)^r)} \end{aligned}$$

From Equation 5.7, we can see that for a fixed r and n, if b increases, the whole running time increases (formally, we can prove that $\frac{\partial T}{\partial b} > 0$). This means that we want b to take the smallest possible value, in order to enforce the maximum admissible error ϵ, so given r, θ and ϵ, b can be computed using Equation 5.8.

$$b = \left\lceil \frac{\log(\epsilon)}{\log(1 - (1 - \theta)^r)} \right\rceil \tag{5.8}$$

Table 5.1 contains some common values that we can use for the algorithm's parameters r and b.

Table 5.1: Values of the parameter b for different settings

r		1	2	3	4	5	6	7	8
θ	ϵ	b							
0.1	0.1	1	2	2	3	3	4	4	5
	0.01	2	3	4	5	6	7	8	9
	0.001	3	5	6	7	8	10	11	13
	0.0001	4	6	8	9	11	13	15	17
0.25	0.1	2	3	5	7	9	12	17	22
	0.01	4	6	9	13	17	24	33	44
	0.001	5	9	13	19	26	36	49	66
	0.0001	7	12	17	25	34	47	65	88
0.5	0.1	4	9	18	36	73	147	294	589
	0.01	7	17	35	72	146	293	588	1177
	0.001	10	25	52	108	218	439	881	1765
	0.0001	14	33	69	143	291	585	1175	2354

The distance threshold θ dramatically influences the growth of b, which in turn increases the running time of the algorithm. The good news, from Equation 5.8 and Table 5.1 is that b grows only logarithmically when ϵ decreases. This means that we can set the error as small as we want and still get a reasonable running time.

5.2.3 Parameters optimization

The distances distribution p from Equation 5.6 cannot be computed directly, but we can estimate it by sampling a reasonable number of random pairs. If we split the interval $[0, 1]$ into k equal intervals, we can approximate p by a constant, on each small interval. We will denote by p_i, the probability that the distance between two samples falls into the interval $\left[\frac{i-1}{k}, \frac{i}{k}\right)$, for $1 \le i \le k$, divided by the length of the interval (or multiplied by k). Note that we will not count the pairs with distance 1 in the last interval, since the probability of such a pair to appear in the same bucket is 0. Now, we can rewrite the fraction f from Equation 5.6:

$$\begin{aligned} f(b,r) &= \int_0^1 p(x) \cdot (1 - (1 - (1 - x)^r)^b) dx \\ &= \sum_{i=1}^{k} \int_{\frac{i-1}{k}}^{\frac{i}{k}} p(x) \cdot (1 - (1 - (1 - x)^r)^b) dx \\ &= \sum_{i=1}^{k} p_i \int_{\frac{i-1}{k}}^{\frac{i}{k}} (1 - (1 - (1 - x)^r)^b) dx \end{aligned}$$

For any given b and r, the fraction f is now easy to compute. Some implementation issues might appear while computing the polynomial under the integral, since the coefficients can reach very large values. The recommendation is to use arbitrary-precision integers and store any rational number as a fraction (nominator and denominator) in order to get the correct results. Some symbolic calculus software package like Maxima [Sch16] can also be used here.

The time required to compute a minhash on a sample and the time required to compute the similarity between two samples are the same order of magnitude. Experimentally, we have determined that $t_2 = 2t_1$, so we can use $t_1 = 1$ and $t_2 = 2$. At this point, if we set the distance threshold θ and the desired error ϵ, the value for b can be computed using Equation 5.8 and the running time of the algorithm only depends on the variable r. In theory, to find the best r we can solve the equation $\frac{\partial T}{\partial r}(r) = 0$. Since r can take only small integer values (for practical purposes we have established that $r \le 8$) it is much simpler to try all the values in range and find the minimum.

Based on our sampled distances' distribution, we have computed the fraction f for the most common case, $\theta = 0.5$, $\epsilon = 0.001$ in Table 5.2. The same table contains the running time T, for $n = 10^5$, $n = 10^6$ and $n = 10^7$.

As we expected, as r and b increase, the fraction of samples that we need to compute the distances for decreases. Even for the smallest values, we have $f(10, 1) = 16.63 \cdot 10^{-5} \approx 0.02\%$. This means that despite the quadratic complexity of the algorithm, we have managed to reduce the coefficient of the quadratic term by several orders of magnitude. Unfortunately, as we increase b and r, the number of minhash functions that we need to compute also increases, so the running time of the first phase of the algorithm

Table 5.2: The fraction of samples to compute distance for and the total running time

r	b	$f \cdot 10^5$	$T(10^5) \cdot 10^{-7}$	$T(10^6) \cdot 10^{-7}$	$T(10^7) \cdot 10^{-7}$
1	10	16.63	**0.27**	17.63	1673
2	25	8.96	0.59	**13.96**	946
3	52	6.12	1.62	21.72	**768**
4	108	4.84	4.37	48.04	916
5	218	4.16	10.94	113.16	1506
6	439	3.76	26.38	267.16	3010
7	881	3.50	61.70	620.20	6517
8	1765	3.31	141.23	1415.31	14451

is greater. This reflects on the total running time, so for $n = 10^5$ samples, the best decision is to set $r = 1$ and $b = 10$. As we increase the number of samples to a million, the quadratic coefficient becomes more important, so the best time is obtained for $r = 2$ and $b = 25$. If the number of samples increases to 10 million, then we must also increase r to 3, to get the best performance.

If we increased the number of samples, we can observe that the trend maintains. For example, if $n = 10^9$, the best performance is obtained for $r = 6$. However, when we are dealing with such a large collection, other problems may arise, like the fact that the data might not fit in the main memory.

5.2.4 An improved clustering algorithm

The previous results can be further improved, by proposing several heuristics. Algorithm 17 computes b bands of hashes and for each band, two samples will end up in the same bucket only if they have the same MinHash value on all r rows. The constant b can be understood as the number of iterations of the algorithm. If we add more bands, it means that we have more buckets and more chances for two similar samples to end up in the same cluster. The number of rows r tells how many times we should split the collection of samples before starting to compute pairwise similarity.

The b and r parameters can be precomputed using the methodology above and be fixed for the entire duration of the algorithm, or they can be adapted as the algorithm progresses. Algorithm 18 is a different way of writing Algorithm 17, emphasizing the ideas of splitting the samples into groups and deciding if we need to take more iterations or not. It starts by creating a set for each sample from the input data (lines 1-3). In the *repeat-until* loop that follows, a sequence of operations is performed on the samples, until a termination condition is met (lines 4-18).

Inside the loop, the entire set of samples is split into smaller partitions by the function SPLIT-SAMPLES() that is detailed in Algorithm 19 (line 5). For each partition, every pair of samples is considered for similarity (line 7). First, the representatives of the two samples are found (lines 8 and 9). If they have the same representative, it means they already belong to the same cluster so no further step is required for that pair. If not, we need to compute the distance between the two samples (line 11). If the distance is smaller than the threshold θ, we just found two similar samples that belong to different clusters. In this case, the two clusters must be joined together, by performing the UNION

Algorithm 18 PERFORM-CLUSTERING($samples, \theta$)

```
Require: A set of features sets $samples$
Require: A distance threshold $\theta$
Ensure: An array $clusters$ of the same size with $samples$ that associates each item with
    a cluster id
 1: for all $s \in samples$ do
 2:     MAKE-SET($s$)
 3: end for
 4: repeat
 5:     $partitions \leftarrow$ SPLIT-SAMPLES($samples, 1$)
 6:     for all $part \in partitions$ do
 7:         for all $(s_1, s_2) \in part \times part$ do
 8:             $rep_1 \leftarrow$ FIND-REPR($s_1$)
 9:             $rep_2 \leftarrow$ FIND-REPR($s_2$)
10:             if $rep_1 \neq rep_2$ then
11:                 $d \leftarrow d_J(s_1, s_2)$
12:                 if $d \leq \theta$ then
13:                     UNION($rep_1, rep_2$)
14:                 end if
15:             end if
16:         end for
17:     end for
18: until TERMINATION-CONDITION()
19: for all $s \in samples$ do
20:     $clusters[s] \leftarrow$ FIND-REPR($s$)
21: end for
22: return $clusters$
```

operation on their sets (line 13).

Finally, all the samples in the same disjoint set are assigned to the same cluster, by labeling them with the same label as the set representative (line 20).

The split must be performed in order to reduce the number of distances to compute. Without the split step, the total number of pairs for n samples is $\frac{n(n-1)}{2} = O(n^2)$. If the set is split into k partitions, of sizes $n_1, n_2, \ldots n_k$, with $n_1 + n_2 + \ldots + n_k = n$, the number of distance computations will be:

$$\begin{aligned} NrPairs &= \frac{n_1(n_1-1)}{2} + \frac{n_2(n_2-1)}{2} + \ldots \frac{n_k(n_k-1)}{2} \\ &= O(n_1^2 + n_2^2 \ldots + n_k^2) \end{aligned}$$

Assuming the k partitions have roughly the same size $n_1 = n_2 = \ldots = n_k = \frac{n}{k}$, then

$$NrPairs = O\left(k \cdot \left(\frac{n}{k}\right)^2\right) = O\left(\frac{n^2}{k}\right) \tag{5.9}$$

Equation 5.9 shows that by splitting the sample into k partitions of roughly the same size, the number of pairs decreases k times.

Algorithm 19 SPLIT-SAMPLES($samplesSet, depth$)

Require: A set of features sets $samplesSet$
Require: An integer $depth$ that represents the number of splits done so far
Ensure: A partition of the input set $samplesSet$ into smaller subsets

```
 1: $h \leftarrow$ RANDOM-MINHASH-FUNCTION()
 2: $partitions \leftarrow \emptyset$
 3: $map \leftarrow$ NEW-ASSOCIATIVE-ARRAY()
 4: for all $s \in samplesSet$ do
 5:     $mh \leftarrow h(s)$
 6:     $map[mh] \leftarrow map[mh] \cup \{s\}$
 7: end for
 8: for all $mh, part \in map$ do
 9:     if SPLIT-CONDITION($part, depth$) then
10:         $newParts \leftarrow$ SPLIT-SAMPLES($part, depth + 1$)
11:         $partitions \leftarrow partitions \cup newParts$
12:     else
13:         $partitions \leftarrow partitions \cup \{part\}$
14:     end if
15: end for
16: return $partitions$
```

The actual split is performed in Algorithm 19 by the function SPLIT-SAMPLES. This function tries to split the given set of samples into several subsets such that the samples situated in the same subset are more likely to be similar than the ones situated in different subsets.

The algorithm starts by generating a random MinHash function (line 1). As explained in the previous section, all we need to do is select two random integers a and b, in order to have a random permutation. The MinHash function will just compute the minimum value of the features from the given set, after applying the random permutation, as in Equation 5.10.

$$h(s) = \min_{x \in s}((a \cdot x + b) \mod |M|) \tag{5.10}$$

The algorithm will also need an associative array (line 3) that represents the mapping between MinHash values and the sets of samples that have the same value. In lines 4-7 the MinHash value for each sample is computed and the sample is inserted in the corresponding set.

In other words, the set of samples received as the function argument is split into several parts by the MinHash value computed on it. These partitions are processed in lines 8-15, where the function SPLIT-CONDITION() decides whether a partition will be further split or left as it is. The *depth* parameter of the function contains the number of splits performed so far. At each recursive call (line 10), the *depth* is bigger. The value for this parameter will be used in the SPLIT-CONDITION() function in order to decide if we must performed a new split.

5.2.4.1 Heuristics for improving the algorithm performance

There are two elements in the clustering algorithm that were left out so far. These elements are:

- the TERMINATION-CONDITION() function, that decides whether the main loop of the algorithm should stop or perform another iteration.
- the SPLIT-CONDITION() function, that decides for a subset of samples if a new split is needed or the algorithm should leave it as it is and proceed with the pairwise distances computation.

The previous clustering technique from Algorithm 17 that uses similar principles for clustering terminates after a finite number of iterations, that are known at the beginning of the algorithm. Our number of iterations corresponds to b, the number of bands used, each containing r rows of MinHashes.

According to this technique, the two functions would be:

- TERMINATION-CONDITION = $(nr.iterations \geq b)$
- SPLIT-CONDITION = $(depth \leq r)$

First of all, we will try to come up with a better termination condition. Our approach is based on the idea that after a couple of iteration, we have more information about the behavior of the algorithm on the current data set, so the decision to terminate or not is more informed than a decision taken at the beginning of the algorithm.

The best indicator of the algorithm's progress is the number of UNION operations performed in Algorithm 18, line 13. This operation is only performed if we found two similar samples that belong to different clusters. In this case, the two clusters will be joined and the total number of clusters will decrease by one. Since the number of clusters at the beginning of the algorithm is n (the number of samples) and we will have at least one cluster in the end, this operation is performed at most $n - 1$ times (this number is much smaller in practice).

In Figure 5.6 we have plotted the number of UNION operations performed by the algorithm in each iteration. This plot was obtained by clustering a collection of 5 million samples for $\theta = 0.25$ with SPLIT-CONDITION = $(depth \geq 3)$ with the above settings but in every other case we experimented the results were similar.

Since the decrease in the number of unions is exponential, we have plotted its logarithm in Figure 5.7. More precisely, if the number of unions at iteration i is $u(i)$, Figure 5.7 plots $U(i) = \log(u(i) + 1)$. when $u(i)$ drops to 0, $U(i)$ also drops to 0.

The new plot shows that the number of unions performed in each iteration is decreasing almost monotonically (with just some variations at the end). Moreover, if we plot the average number of unions in the last 5 iterations, the function becomes truly monotonic. This means that after a specific number of iterations very few unions are left to be performed so the results of the algorithm won't improve too much. Indeed, by computing the percentage of unions completed after a number of iterations, we obtained:

- 74.07% after the first iteration
- 90.13% after two iterations
- 98.52% after 5 iterations

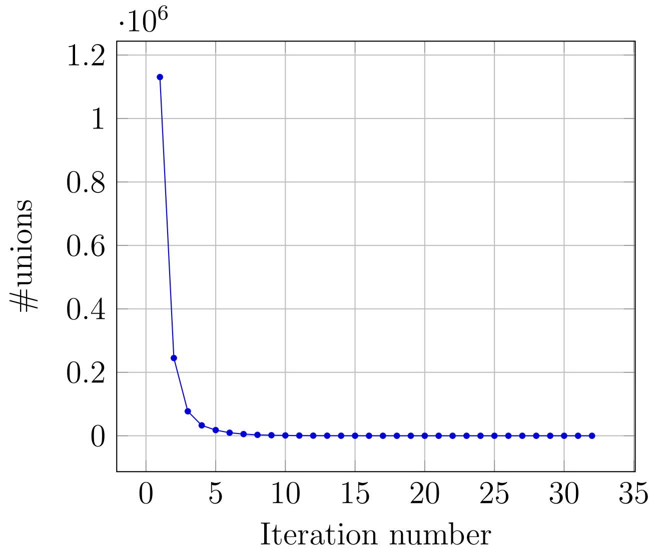


Figure 5.6: Number of unions performed in each iteration

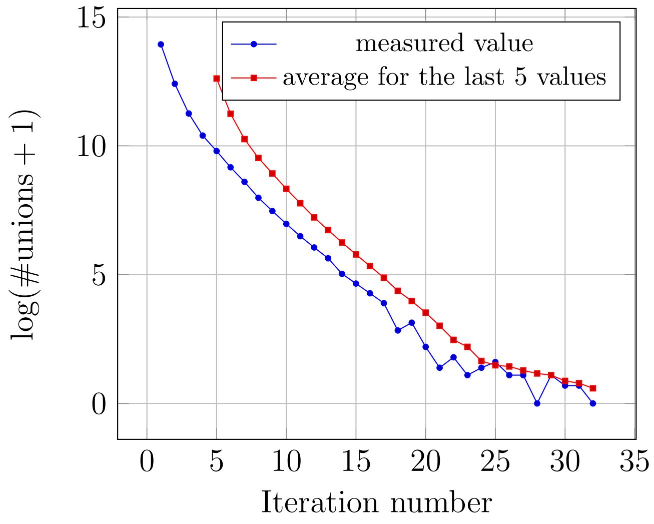


Figure 5.7: Number of unions performed in each iteration

- 99.88% after 10 iterations
- 99.98% after 15 iterations

The above experiments lead to the following heuristic for the termination condition: *The algorithm should finish when the number of unions performed in the last iterations is too small for any significant improvements.* A safe TERMINATION-CONDITION() function, based on Figure 5.7 is to stop when the average number of unions performed in the last 5 iterations is smaller than 1.

The SPLIT-CONDITION() function can also improve if we don't impose the split depth to be the same every time. For small subsets of the samples set the split is not really necessary, as the number of pairs to be evaluated is also small. Splitting subsets will separate pairs of potentially similar samples that will delay the clusters joining and will increase the number of necessary iterations of the algorithm. On the other hand, large subsets must be split, otherwise a big number of distance computations will slow down the algorithm. In the following section, we will try three heuristics for the split condition and determine experimentally which one achieves better results:

- *depth heuristic*: SPLIT-CONDITION = $(depth \leq r)$
- *size heuristic*: SPLIT-CONDITION = $(|part| \geq size_1)$
- *size and depth heuristic*: SPLIT-CONDITION = $(depth \leq maxDepth \wedge \frac{|part|}{depth} \geq size_2)$

The *depth heuristic* will follow the principle in the previous subsection, where a fixed number of splits were performed in each band (r, the number of rows was constant).

The *size heuristic* is based on the reasoning that large subsets must be split in order to reduce the number of distance computations while small sets are better left untouched. In this case, the split is performed only when the size of the subset is larger than a threshold.

The *size and depth heuristic* is an extension of the previous one and also takes into account the depth of the split. This depth should not exceed a maximum value (this margin should be larger than the one from the *depth heuristic*) and also the ratio between the size of the subset and the depth should be larger than a threshold. By dividing the size of the subset by the current depth, we are discouraging splits at higher depths and allow them only if the subset is really large.

5.3 Experimental Evaluation

5.3.1 Experimental results on clustering using Suffix Trees

Both Algorithm 13 and Algorithm 14 have been implemented in C++ and tested on OpCode strings extracted from real-world malware that were prevalent in the last quarter of 2013 and the first half of 2014. Several tests were performed:

- We have tested to see if both algorithms produced the same output.
- Some clusters (about 5%) were validated manually to see if the samples in the same cluster belong indeed to the same malware family.
- The performance of Algorithm 14 was measured in order to prove the linearity on a system with a 2.40 Ghz processor and 24 GB of RAM memory.
- On a machine with more RAM memory (96 GB), we have tested to see that the algorithm is scalable and can be used to cluster a large number of samples.

Because Algorithm 13 performs clustering in quadratic time, we were not able to test it on a large number of samples. For 5000 samples, the average running time was a little more than an hour and a half. In all the performed tests, QUADRATIC-CLUSTERING and
LINEAR-CLUSTERING produced exactly the same clusters. The clusters quality was good, especially after filtering out the library code.

The function QUADRATIC-CLUSTERING in Algorithm 14 produced the running times from the second column of Table 5.3, also illustrated in Figure 5.8.

For space reasons, in Table 5.3 are shown only a couple of measurements. In order to create the plot in Figure 5.8, 5 test were run for each value of n, varying from 200 to

Table 5.3: Running time for QUADRATIC-CLUSTERING (Algorithm 13) and LINEAR-CLUSTERING (Algorithm 14)

n	Running time (s) QUADRATIC-CLUSTERING	Running time (s) LINEAR-CLUSTERING
200	16.82	0.53
600	130.49	0.72
1000	316.42	0.89
2000	1095.34	1.35
3000	2417.19	1.90
4000	3861.28	2.43
5000	5645.97	3.00
10000		5.66
20000		10.75
30000		15.98
40000		21.77
50000		26.51
60000		31.56
70000		36.42
80000		41.65
90000		46.86
100000		52.01

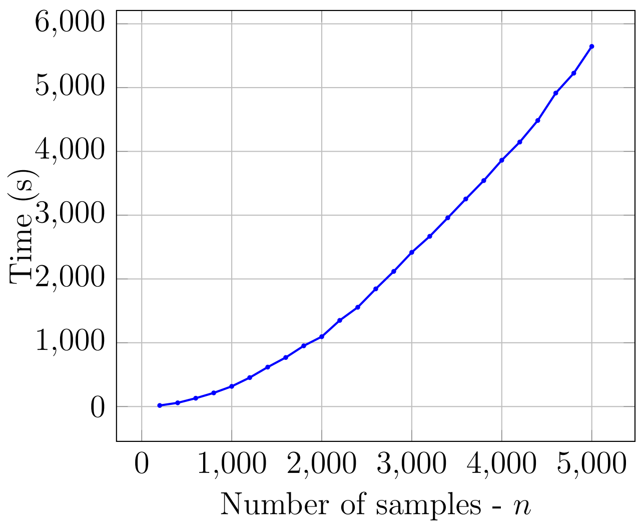


Figure 5.8: Running time for Algorithm 13 varying the number of samples n

5000, using a step of 200. From the table and the plot we observe that the complexity of the algorithm is quadratic in n, as shown in subsection 5.1.2.1. Also, since the running time is more than an hour and a half for $n = 5000$, the algorithm is impractical for a large number of samples (for 20000 samples, the estimated running time is more than 24 hours).

We have also tested the function LINEAR-CLUSTERING from Algorithm 14 for the same samples, also varying from 200, using a step of 200. Since the algorithm proved to be much faster, we didn't stop at 5000, but continued the tests to 100000, as shown in Table 5.3. The plot for the entire range of values is shown in Figure 5.9. The plot confirms that the algorithm is linear for the average case.

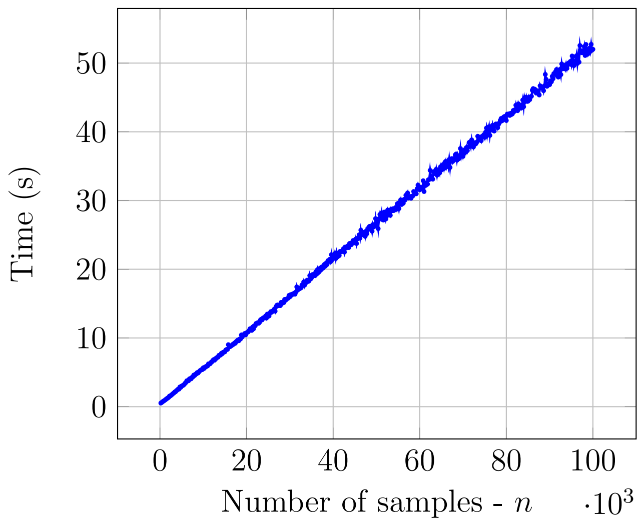


Figure 5.9: Running time for Algorithm 14 varying the number of samples n

One disadvantage of the linear clustering algorithm is the memory consumption. On a 64-bit machine, the suffix tree takes about 12 GB of RAM.

We have also tested the algorithm on another system with more memory and managed to cluster 250000 samples in 137 seconds. This test shows that Algorithm 14 is a viable solution for clustering large malware collections because the running time for any quadratic algorithm is impractical for collections larger than 100000 samples.

5.3.2 Clustering results with LSH optimization

We have experimented the clustering algorithm on a collection of over one million malware samples, from which we extracted 130000 unique sets of features. In this section, we will consider the number of samples $n = 130000$, as the identical samples can be easily eliminated.

The first thing that we tested was the quality of the obtained clusters. We have used the methods described in the theoretical section in order to compute the recall and see if the maximum admissible error ϵ matches the theoretical results. For $\theta = 0.5$ and $r = 2$, we have tried various values for ϵ and computed b based on them. Then, we ran the clustering algorithm and computed the real error $\epsilon_{exp} = 1 - R$. The results are presented in Table 5.4.

Table 5.4: Comparison between the theoretical estimation and the practical measurement for the error ϵ

ϵ	0.1	0.05	0.01	0.001	0.0001
b	9	11	17	25	33
ϵ_{exp}	0.056	0.027	0.0044	0.0017	0

Next, the performance of the entire clustering algorithm was measured. We have maintained the common settings $\theta = 0.5$, $\epsilon = 0.001$ and used for r and b the pairs $(1, 10)$, $(2, 15)$, $(3, 52)$, $(4, 108)$ and $(5, 218)$. The collection size n was varied from 5000 to 130000, by 5000.

For space reasons, Table 5.5 only shows some of the values. The five cases ($r \leftarrow 1..5$) are also plotted in Figure 5.10.

Table 5.5: The running time of the clustering algorithm

n	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$
5000	0.10	0.21	0.45	1.04	2.41
30000	0.74	1.54	3.36	7.63	17.43
55000	1.48	3.07	6.55	14.91	34.46
80000	2.34	4.71	9.98	22.77	52.33
105000	3.21	6.39	13.45	31.14	71.58
130000	4.15	8.06	17.06	39.67	91.75

Figure 5.10 shows that for $n \leq 130000$, the complexity of the algorithm is perceived to be linear, because the coefficient of the quadratic term is very small.

We wanted to see if we can improve the algorithm even further by parallelizing it. Indeed, the first phase can be run in parallel because the minhashes can be computed individually for each sample and we only need a synchronization mechanism when inserting samples into the buckets. In the second phase, the buckets can be treated in parallel, the only synchronization being required on the disjoint set forest operations.

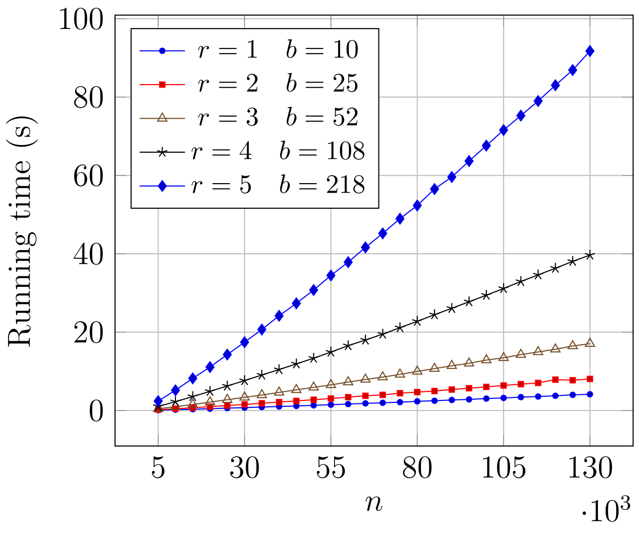


Figure 5.10: Running time of the clustering algorithm

In Figure 5.11 we will show the running time of the algorithm for $r = 5, b = 218$ on 1, 2, 4 and 8 threads.

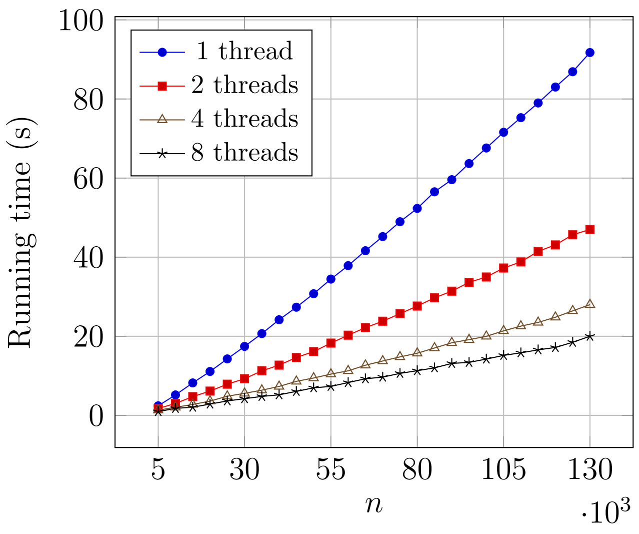


Figure 5.11: Running time of the clustering algorithm on different numbers of threads

The experimental results show that even more performance can be squeezed out of the LSH-CLUSTERING function if parallelism is used.

During the performance measurement we have also counted how many times we had to actually compute the distance between two samples. If we had to compute c such distances for n samples, given the b and r parameters, then $f_{exp}(b, r) = \dfrac{2c}{n(n+1)}$ is the experimental measurement for the fraction in Equation 5.6, that was theoretically estimated in Table 5.2. Table 5.6 will compare the theoretical estimations with the measured values.

The measurements show that we have to compute the distance for even fewer pairs that was estimated theoretically ($f_{exp} < f$). The result was expected, since in the theoretical estimation we didn't account for the fact that the pairs of samples already in the same cluster will be skipped.

Table 5.6: Comparison between the theoretical estimation and the practical measurement for the fraction f

r	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$
$f \cdot 10^5$	16.63	8.96	6.12	4.84	4.16
$f_{exp} \cdot 10^5$	14.99	6.09	3.37	2.36	1.92

5.3.3 Clustering collections of multiple millions

The experimental evaluation for the technique described in subsection 5.2.4 was performed on a collection of over 10 million distinct sets of features extracted from binary programs (both malware and clean samples). For improved results, we have also extracted a larger number of n-grams, by analyzing the entire code of each application.

For each heuristic, we wanted to chose a good threshold before comparing it with the other heuristics. The threshold was chosen by experimenting different values on the first 1, 2, 3, 4 and 5 million samples of the collection.

For the *depth heuristic*, the clustering algorithm was run for $depth \leq 3$, $depth \leq 4$ and $depth \leq 5$. The results are presented in Table 5.7 and Figure 5.12.

Table 5.7: The running time of the clustering algorithm for *depth heuristic*

n	$depth \leq 3$	$depth \leq 4$	$depth \leq 5$
$1 \cdot 10^6$	425.2	441.4	543.6
$2 \cdot 10^6$	1202.1	1158.8	1394.2
$3 \cdot 10^6$	2497.8	2084.1	2128.8
$4 \cdot 10^6$	4075.7	3203.4	3447.1
$5 \cdot 10^6$	6425.9	4651.6	4740.6

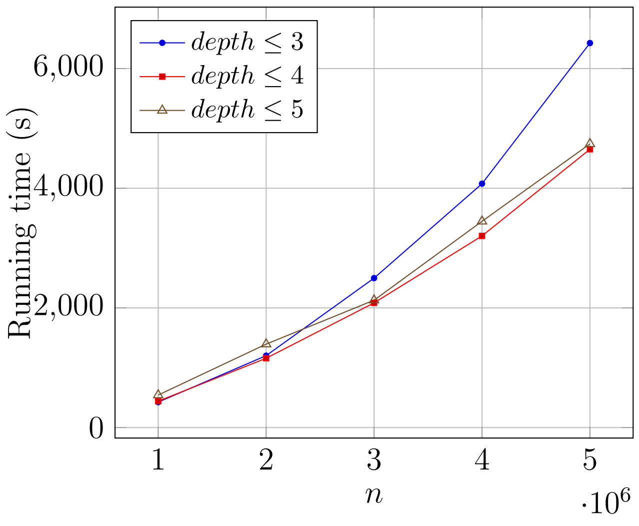


Figure 5.12: Running time of the clustering algorithm for *depth heuristic*

The three plots in Figure 5.12 show that the running time of the algorithm is slightly better for $depth \leq 4$ than for $depth \leq 5$ and significantly better than for $depth \leq 3$. Indeed, for $depth \leq 3$ there were large partitions not split, that caused a significant number of pairwise distance computations. Note that the shape of this plot resembles more a quadratic than a linear function. For *depth* ≤ 4 and *depth* ≤ 5 the partitions are sufficiently split but in the *depth* ≤ 5 the increased number of splits led to a higher number of iterations (48 for 5 million samples, as opposed to 37 in the *depth* ≤ 4 case) and overall a greater running time.

The *size heuristic*, that choses to split a subset only when its size is greater than a threshold obtained better running times than the *depth heuristic*, according to Table 5.8 and Figure 5.13. The plot shows that the performance is affected by the choice of the threshold but not greatly. The best performance was obtained for *size* ≥ 50, by a small margin.

Table 5.8: The running time of the clustering algorithm for *size heuristic*

n	*size* ≥ 25	*size* ≥ 50	*size* ≥ 100
$1 \cdot 10^6$	298.5	310.8	342.4
$2 \cdot 10^6$	1023.8	946.2	927.7
$3 \cdot 10^6$	1888.9	1834.7	1843.1
$4 \cdot 10^6$	2744.5	2572.4	2592.8
$5 \cdot 10^6$	3625.4	3527.9	4536.5

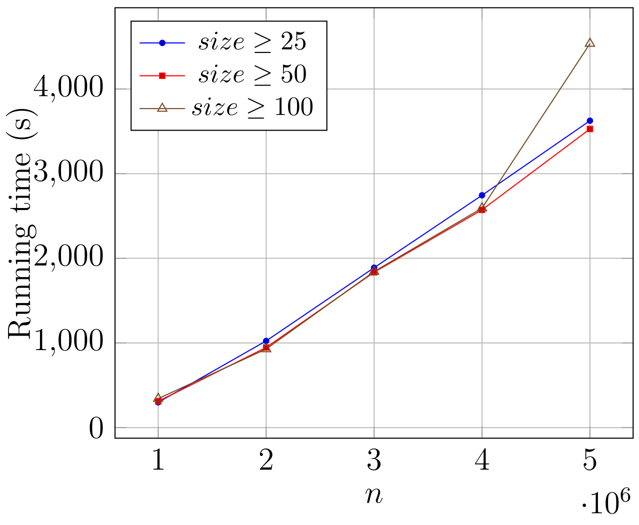


Figure 5.13: Running time of the clustering algorithm for *size heuristic*

The *size and depth heuristic* provided the best results, as shown in Table 5.9 and Figure 5.14. The best results were obtained if we decided to split a subset only if the ration between its size and the number of splits performed so far is at least 50. The thresholds 25 and 100 also provided good performance but the experimental evaluation showed that for 25 there are too many splits and for 100 too few.

After determining the optimal setup for each heuristic, we ran them on the entire collection of 10 million samples and compare the running times among them.

The running times for the three heuristics are presented in Table 5.10 and Figure 5.15. The times are detailed, considering the time required to compute MinHash values in order to split the samples (t_1) and the time required to perform pairwise computations (t_2).

Table 5.9: The running time of the clustering algorithm for *size and depth heuristic*

n	$\frac{size}{depth} \geq 25$	$\frac{size}{depth} \geq 50$	$\frac{size}{depth} \geq 100$
$1 \cdot 10^6$	211.3	165.8	227.3
$2 \cdot 10^6$	738.6	552.7	745.9
$3 \cdot 10^6$	1502.1	1119.2	1688.5
$4 \cdot 10^6$	1976.2	1784.0	2522.2
$5 \cdot 10^6$	2556.7	2492.8	3834.1

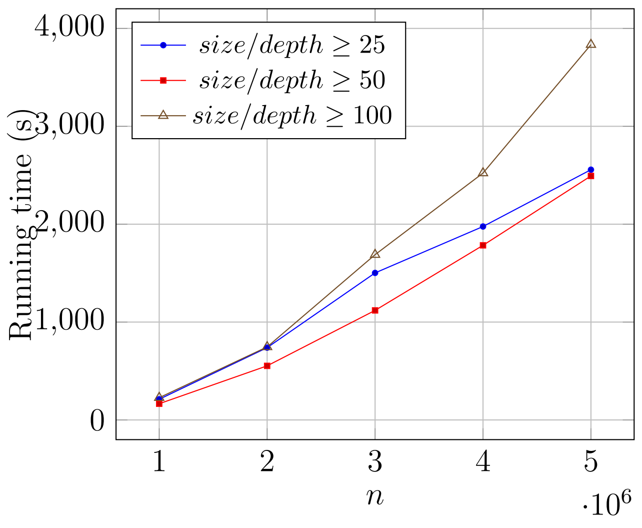


Figure 5.14: Running time of the clustering algorithm for *size and depth heuristic*

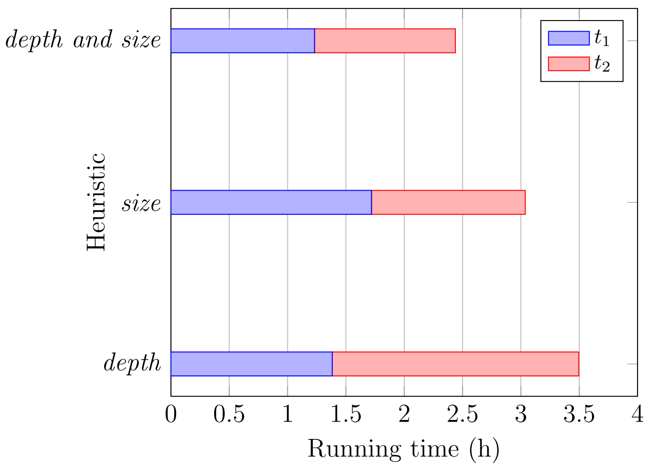


Figure 5.15: Running times of the clustering algorithm for 10 million samples

Figure 5.15 shows that the *depth and size heuristic* have the best performance, spending the least time on both computing MinHashes and computing distances. The *size heuristic* spent more time on computing MinHashes but still has a better overall performance than the *depth heuristic*.

The *depth and size heuristic* found 3,603,251 cluster in the 10 millions samples collection. The number of clusters found by the *depth heuristic* and the *size heuristic* are 3,603,268 and 3,603,277. Since the difference between the smallest and the biggest

Table 5.10: The running times of the clustering algorithm for 10 million samples

	depth	*size*	*depth and size*
$t1$	4978.5	6189.8	4431.5
$t2$	7607.7	4745.6	4346.9
total time	12586.2	10935.4	8778.4

number of clusters is less than 0.0007%, we can conclude that all 3 heuristics give results of the same quality and should only be compared in terms of performance.

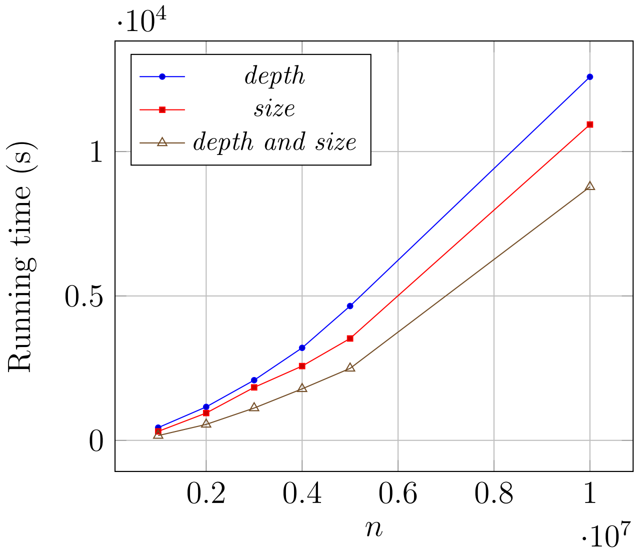


Figure 5.16: Performance comparison between the three heuristics

Finally, the overall performance for the best setting of the three heuristics is plotted in Figure 5.16 where we see that the best approach is the *depth and size heuristic*.

5.4 Chapter conclusions

In order to analyze large programs collections we need to split them into clusters, each cluster containing similar samples so only some representatives need to be further analyzed.

Classical clustering algorithms involve computing the distance between each pair of samples in the collection. For large collections of millions or even billions samples, these distance computations alone would take too much time for the algorithm to be of any practical value.

A clustering algorithm based on Ukkonen's Generalized Suffix Tree was given and we proved both theoretically and experimentally that it is linear in the string size and in the number of samples on the assumption that the average cluster size is constant.

The other proposed clustering technique was based on the concept of Locality-Sensitive Hashing. The thesis contribution is a formal technique for selecting the optimal parameters for LSH in order to reduce the number of distance computations to a minimum. This technique allowed clustering 10 million samples in a few hours.

The following contributions were presented in this chapter:

- an almost-linear clustering algorithm for OpCode strings based on Suffix Trees;
- a technique for selecting the optimal parameters for Locality-Sensitive Hashing in order to improve clustering performance;
- two clustering systems based on Suffix Trees and Locality-Sensitive Hashing, able to cluster several million unique samples in a few hours;

This chapter is based on the following published work:

- **Ciprian Oprișa**, George Cabău, and Gheorghe Sebestyen Pal. Malware clustering using suffix trees. *Journal of Computer Virology and Hacking Techniques*, pages 1–10, 2014. ([OCSP14a])
- **Ciprian Oprișa**, Marius Checicheș, and Adrian Năndrean. Locality-sensitive hashing optimizations for fast malware clustering. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 97–104, Cluj-Napoca, Romania, 2014. IEEE. ([OCN14])
- **Ciprian Oprișa**. A minhash approach for clustering large collections of binary programs. In *Control Systems and Computer Science (CSCS), 2015 20th International Conference on*, pages 157–163, Bucharest, Romania, 2015. IEEE. ([Opr15])

Chapter 6

Clustering-based Malware Analysis

6.1 Multi-centroid Cluster Analysis

The previous chapters showed how to extract code features from binary programs and how to perform cluster analysis based on these features. Clusters are very helpful in malware analysis tasks, as the similar samples are already grouped together. A human analyst can simply analyze a few representatives from each cluster and infer a verdict for an entire cluster. This section will provide an automated methodology for selecting the representatives for a cluster in order to minimize the human work.

6.1.1 Clustering and Verdicts

We will denote by $\mathcal{S}$, the set of all feature sets extracted from the sample programs. The recurring issue in malware analysis is the verdicts assignments. Each sample should be given a verdict, that can be $clean$ or $infected$, as in Equation 6.1.

$$verdict : \mathcal{S} \to \{clean, infected\} \tag{6.1}$$

A distance function d, as in Equation 6.2 will compute the dissimilarity between two samples.

$$d : \mathcal{S} \times \mathcal{S} \to [0, 1] \tag{6.2}$$

For instance, the extracted features can be a set of n-grams, as detailed in the previous chapters. In this case the Jaccard distance can be used. However, the reasoning will be similar for any metric distance defined on $\mathcal{S}$ that takes values between 0 and 1.

For such a metric d, we will make the following assumption: for two samples $A, B \in \mathcal{S}$ the probability they will share the same verdict is:

$$P(verdict(A) = verdict(B)) = 1 - \frac{d(A, B)}{2} \tag{6.3}$$

Basically, Equation 6.3 states that two samples that are very similar (the distance between them is small) are likely to share the same verdict. However, if the samples are completely dissimilar ($d(A, B) = 1$), no assumption can be made (the probability will be 0.5).

The first step for minimizing the manual work of assigning verdicts to each sample is to cluster them. We will use the single linkage approach [Sib73], in order to ensure that each pair of samples that are similar enough will end up in the same cluster. More formally, for a given distance threshold θ, $\forall A, B \in \mathcal{S}$ such that $d(A,B) \leq \theta$, we will have $cluster(A) = cluster(B)$. For dealing with a large collection of samples, we will use the locality-sensitive hashing approach from the previous chapter.

Each cluster can be interpreted as a connected graph, $G = (V, E)$, where $V \subset \mathcal{S}$ is the set of all features sets associated to the samples in that cluster. Two nodes are connected by an edge if their distance is smaller than the threshold θ: $E = \{(u,v) \in V \times V \mid d(u,v) \leq \theta\}$. Also, each edge will be weighted. The weight function is the distance between the two nodes, as in Equation 6.4.

$$\begin{aligned} w &: E \to [0,1] \\ w((u,v)) &= d(u,v) \end{aligned} \tag{6.4}$$

The biggest issue with the single linkage approach is the *chaining effect*. Sample A can be similar with B and B can be similar with C, while A and C are not necessarily similar. The issue is illustrated in Figure 6.1, where we have used simple lines to represent distances smaller than the threshold θ and dashed lines for larger distances.

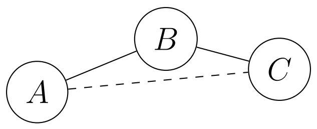


Figure 6.1: Chaining effect illustrated

The single linkage approach would place the nodes A, B and C from Figure 6.1 in the same cluster. To minimize the amount of work required to assign verdicts for all samples, a researcher can be asked to assign verdicts only for some of the elements of a cluster. Then, by association, the verdict can be extended to the neighboring nodes in the graph. Because of the chaining effect, we will only allow verdict extension to the neighboring nodes. For instance, if we have a verdict given by a human for the node A, we can expand the verdict to the node B, but we won't expand it further from B to C, so we will have to request further investigations for sample C. If we asked the human to analyze the sample B, the verdict can be directly extended to both nodes A and C, reducing the human work to a single analysis.

Another issue may arise for the cluster in Figure 6.1 if A and C will be given different verdicts (e.g. A is declared clean and B infected). In this case, further human investigation is necessary in order to establish the correct verdict for B. However, if A and C were given the same verdict, the verdict for B could be automatically inferred with a high degree of confidence.

By generalizing the above issues to any graph, we can define the two main problems that the next section will try to solve:

- Given a cluster of possibly malicious programs, select a subset of samples to be manually analyzed so that the rest of verdicts can be inferred by neighborhood associations.

- For a cluster where some of the samples already have verdicts (given by humans or automated systems), determine if the rest of the verdicts can be automatically inferred or there are conflicts that need to be addressed manually.

6.1.2 Multi-centroid Selection Algorithm

Several clustering algorithms use the concept of *centroid*. For instance, the k-means algorithm [M+67] determine a set of centroids, then groups the remaining points around them, by proximity. Such centroids can be used as representative points for the cluster. However, hierarchical clustering methods, such as single linkage do not use centroids.

In order to manually analyze the most relevant samples, we would like to determine one or more centroids for every cluster. The rule that we have established in the first section, to infer verdicts only for the direct neighbors of the analyzed nodes will require a subset of samples, such that each sample in a cluster is either a centroid or is adjacent to one.

This problem of finding such a set of centroids is called in literature the *minimum dominating set* and has been proved to be NP-hard [HL90]. It can be linearly reduced to the *set cover problem* [Kan92] which already has a $(1 + \log |V|)$-approximation [Joh73]. The problem was also proved not be $(1 - \epsilon) \log |V|$-approximable for any $\epsilon > 0$ [Fei98]. An exact algorithm has been published by van Rooij and have a complexity of $O(1.5048^n)$ [vRNvD09].

The $(1 + \log |V|)$-approximation algorithm was adapted from [Joh73] to find a set of centroids for a given graph in Algorithm 20.

Algorithm 20 FIND-MULTIPLE-CENTROIDS(G)

Require: A cluster of samples, represented as a graph $G = (V, E)$
Ensure: The set of centroids, $C \subset V$

```
1: $C \leftarrow \emptyset$
2: $NotReached \leftarrow V$
3: for all $v \in V$ do
4:     $Neighb[v] \leftarrow \{v\} \cup \{u \in V \mid (v, u) \in E\}$
5: end for
6: while $NotReached \neq \emptyset$ do
7:     $c \leftarrow \arg\max_{v \in NotReached} |Neighb[v]|$
8:     $C \leftarrow C \cup \{c\}$
9:     $crtReach \leftarrow Neighb[c]$
10:    $NotReached \leftarrow NotReached \setminus crtReach$
11:    for all $v \in crtReach$ do
12:        $Neighb[v] \leftarrow Neighb[v] \setminus crtReach$
13:    end for
14: end while
15: return $C$
```

The algorithm starts by initializing the set of centroids with the empty set (line 1) and the set of nodes that have not been reached yet with V (line 2). For each node, we will build a neighborhood set, comprised of itself and all the other nodes directly reachable from it (line 4).

While the set of nodes that weren't reached yet is not empty, the node with the highest neighborhood is selected as a new centroid (lines 7-8). Its entire neighborhood is then eliminated from the list of nodes not reached yet (line 10) and from the neighborhoods of its neighbors (line 12).

The greedy part of the algorithm is the selection of the next centroid (line 7) as the node with the highest neighborhood, considering only the nodes not yet reached. This heuristic tries to eliminate as many nodes as possible from the *NotReached* set at each step. The smaller number of steps, the smaller the number of centroids for that cluster will be.

For each iteration of the algorithm, at least one vertex is eliminated from the *NotReached* set, so we have at most $|V|$ iterations. In each iteration, the vertex with the largest neighborhood is selected (line 7), also in $O(|V|)$ steps, so the complexity so far is $O(|V|^2)$. The subtraction of the currently reachable set from the neighborhoods (line 12) has a running time proportional with size of *crtReach*, assuming the sets are implemented as look-up tables. However, each node will only be eliminated once from the neighborhoods, so this operation also takes $O(|V|^2)$. We conclude that the running time of Algorithm 20 is quadratic in the cluster size.

After a set of centroids is selected and some verdicts are assigned to each of them, the verdicts can be extended to all the nodes in the graph. When all the verdicts for the centroids agree (all of the centroid samples are clean or all of them are infected), the problem is trivial, as every node in the graph will share the same verdict. If some centroids are given different verdicts, there will be at least one contradictory edge (the two ends of the edge will have different verdicts). Since the graph is connected, there will either be two centroids that are directly connected and have different verdicts, or there will be a non-centroid node adjacent to both a clean and an infected centroid.

The aforementioned contradictions may appear for two reasons:

- some verdicts were incorrectly assigned
- some nodes that are connected by an edge are in fact not similar

To discern between the two cases and to fix the graph verdicts, further human intervention is required. An analyst can identify incorrectly assigned verdicts and fix them or he can remove some edges of the graph in order to separate it into smaller connected components with uniform verdicts.

6.2 Semi-automated Verdicts Assignment

The model described in the previous section labeled each node with a single verdict: *clean* or *infected*. The new approach will work with a fuzzy model: each node will be labeled with a real number from the interval $[-1, 1]$. Every node $v \in V$ will have an associated label $l(v)$. Given two thresholds θ_1 and θ_2, such that $-1 < \theta_1 < 0 < \theta_2 < 1$, we can interpret the labels in the following way:

- a node v, such that $l(v) \leq \theta_1$ corresponds to a clean file
- a node v such that $l(v) \geq \theta_2$ corresponds to an infected file
- if $\theta_1 < l(v) < \theta_2$, the verdict is uncertain

For a cluster of samples represented as a connected graph, we will give the following definitions:

Definition 2. *An edge $(u, v) \in E$ is called **balanced** in respect to a given labeling $l : V \to [-1, 1]$ if $|l(u) - l(v)| \leq d(u, v)$ (the absolute difference between the labels of the two nodes should not exceed their distance).*

Definition 3. *A vertex $v \in V$ is called **balanced** in respect to a labeling l if all its adjacent edges are balanced. The set of all balanced vertices for a labeling l of a graph will be denoted as $B(V, l) \subseteq V$. The set of unbalanced vertices will be $\overline{B}(V, l) = V \setminus B(V, l)$.*

Definition 4. *A graph $G = (V, E)$ is called **balanced** in respect to a labeling l if all its vertices are balanced in respect to that labeling ($B(V, l) = V$).*

Labeling verdicts with a real number between -1 and 1 offers the possibility to interpret them as a degree of certainty. For instance, if an analysts is sure a given sample is malicious, it will be labeled as 1. Malicious actions observed in a controlled environment like a virtual machine will also indicate that the sample is likely to be malware, but the label value will be smaller (like 0.9), as some non-malicious action may be falsely interpreted as malicious by heuristics. Detection of the anti-virus engines might also indicate maliciousness with a varying degree of confidence (depending on the individual verdict from each engine and its known false positives ratio). Similarly, a sample declared by a human to be clean or a component from a popular operating system may be labeled with -1, while a sample not detected by any anti-virus engine and prevalent in the market will receive a negative value (it is likely to be clean) but smaller in module. The labeling in the previous examples was performed by empirical rules but a more rigorous model can be given using the probability theory.

Whichever method we choose for assigning initial labels to a cluster of samples, they must be reconsidered after analyzing the similarities between the samples in the cluster. Informally, we can state that similar samples should have close labels. Formally, we can use the definitions above and request that the graph associated with the cluster is balanced. To achieve this goal, the original labels should be modified as little as possible, until the graph becomes balanced. In this section we will give an algorithm to balance a verdicts graph.

The algorithm comprises several iterations and stops only when the graph is balanced. The key idea is that unbalanced nodes send messages to each other through unbalanced edges. At each iteration, each unbalanced node adjusts its label according to the messages it receives from its neighbors.

At the beginning of each iteration, the message queue for each node empties (line 3). Then, for each edge of the graph we check if the absolute difference between the labels of its adjacent nodes is greater than their distance (line 5). If it is, the edge is unbalanced, so the nodes u and v send each other a message containing its current label and their similarity (one minus the distance between them).

In the final part of the iteration, if a node received any message, the function AVERAGE-MSG will compute the weighted average of the labels it received, using the similarities as weights. Then, the label of the current node is updated using the equation in line 12. This equation computes a linear combination between the previous label value and the average of the messages received from the neighbors. The constant λ is called the transfer factor and decides how much the new label is influenced by the neighbors. A small value for λ assures that the labels don't change too much between iterations.

Theorem 3. *The* BALANCE-GRAPH *function in Algorithm 21 terminates in a finite number of iterations.*

Proof. The algorithm terminates when all the nodes of the graph become balanced.

Lemma 3. *Let l_1 and l_2 be the labeling functions for two consecutive steps of Algorithm 21. Then, $\forall v \in \overline{B}(V, l_1)$, $l_2(v) \leq \max\limits_{u \in \overline{B}(V, l_1)} l_1(u) - c$ for some constant c.*

Algorithm 21 BALANCE-GRAPH(G, d, l_0, λ)

```
Require: A cluster of samples, represented as a graph $G = (V, E)$
Require: A distance $d : V \times V \to [0, 1]$
Require: An initial labeling of the nodes $l_0$
Require: A transfer factor constant $0 < \lambda < 0.5$
Ensure: A new labeling of the nodes $l$, such that the graph $G$ is balanced in respect to
    it
 1: $l \leftarrow l_0$
 2: while $|B(V, l)| \neq |V|$ do
 3:     $Msg[v] \leftarrow \emptyset, \forall v \in V$
 4:     for all $(u, v) \in E$ do
 5:         if $|l(u) - l(v)| > d(u, v)$ then
 6:             $Msg[u] \leftarrow Msg[u] \cup \{(l(v), 1 - d(u, v))\}$
 7:             $Msg[v] \leftarrow Msg[v] \cup \{(l(u), 1 - d(u, v))\}$
 8:         end if
 9:     end for
10:     for all $v \in V$ do
11:         if $|Msg[v]| > 0$ then
12:             $l(v) \leftarrow (1 - \lambda) \cdot l(v) + \lambda \cdot$ AVERAGE-MSG$(Msg[v])$
13:         end if
14:     end for
15: end while
16: return $l$
```

Proof. Let $s = \min\limits_{(u,v)\in E} d(u, v)$ be the smallest distance between two vertices in the graph. Since d is a metric, we know that $d(u, v) = 0 \iff u = v$, so $s > 0$.

Let $l_M = \max\limits_{v\in\overline{B}(V,l_1)} l_1(v)$ be the maximum value for the labels of the unbalanced nodes.

For an unbalanced node v, we will call *active neighbors*, the neighboring nodes that are also unbalanced and send messages to v in the current iteration of the algorithm.

We will assume that $\exists v \in \overline{B}(V, l_1)$ such that $l_2(v) > l_M - \lambda \cdot s$ and show that this assumption leads to a contradiction. There are two cases:

- $l_1(v)$ is higher than all the labels of the active neighbors of v
- $l_1(v)$ has at least an active neighbor with a higher label

Case 1: $l_1(v)$ is higher than all the labels of the active neighbors of v.

In order for a neighbor u to send a message to v, the constraint from line 5 must be enforced, so

$$|l_1(u) - l_1(v)| > d(u, v) \tag{6.5}$$

Since for this case, the label of v is higher than the labels of all the active neighbors, including u, $l_1(v) > l_1(u)$ so $|l_1(u) - l_1(v)| = l_1(v) - l_1(u)$. Also, $d(u, v) \geq s$, so the inequality in Equation 6.5 becomes:

$$l_1(v) - l_1(u) > s \tag{6.6}$$

$$\Rightarrow l_1(u) < l_1(v) - s \tag{6.7}$$

Equation 6.7 states that each node that sends a message to v has the label smaller than $l_1(v) - s$. The weighted average of the messages will also be smaller than $l_1(v) - s$, so the equation that updates the label from line 12 will become:

$$\begin{aligned} l_2(v) &= (1-\lambda) \cdot l_1(v) + \lambda \cdot \text{AVERAGE-MSG}(Msg[v]) \\ &< (1-\lambda) \cdot l_1(v) + \lambda \cdot (l_1(v) - s) \\ &= l_1(v) - \lambda \cdot s \\ &< l_M - \lambda \cdot s \end{aligned}$$

Case 2: $l_1(v)$ has at least one active neighbor with a higher label.

In this case, for a node u with the higher label ($l_1(u) > l_1(v)$) to send the message, the difference between their labels should be higher than s (using the same reasoning that led to Equation 6.7). If $l_1(v) > l_M - s$, then $l_1(u) > l_1(v) + s > l_M$ - contradiction, since l_M is the maximum value for the label of an unbalanced node in the current iteration. It turns out that $l_1(v) \leq l_M - s$. From the update equation in line 12, knowing that the weighted average of the received messages cannot exceed l_M, we have:

$$\begin{aligned} l_2(v) &= (1-\lambda) \cdot l_1(v) + \lambda \cdot \text{AVERAGE-MSG}(Msg[v]) \\ &\leq (1-\lambda) \cdot l_1(v) + \lambda \cdot l_M \\ &\leq (1-\lambda) \cdot (l_M - s) + \lambda \cdot l_M \\ &= l_M - (1-\lambda) \cdot s \end{aligned}$$

Since $\lambda < 0.5$, we have that $2\lambda < 1 \iff \lambda < 1 - \lambda \iff -(1-\lambda) < -\lambda$. Then the inequality above turns into:

$$l_2(v) \leq l_M - (1-\lambda) \cdot s < l_M - \lambda \cdot s$$

In both cases we have reached a contradiction so $\nexists v \in \overline{B}(V, l_1)$ such that $l_2(v) \geq l_1(v_M) - \lambda \cdot s$. This means that $\forall v \in \overline{B}(V, l_1)$, $l_2(v) < l_M - \lambda \cdot s$. If we denote $c = \lambda \cdot s$, we obtain $\forall v \in \overline{B}(V, l_1)$, $l_2(v) \leq \max\limits_{u \in \overline{B}(V, l_1)} l_1(u) - c$, which concludes the proof. $\square$

Lemma 3 states that the maximum label of the currently unbalanced nodes will decrease by at least a constant c, in each iteration. However, a node with a higher label that was previously balanced may become unbalanced, so the maximum unbalanced label for the entire set of vertices might increase. Figure 6.2 depicts such a case.

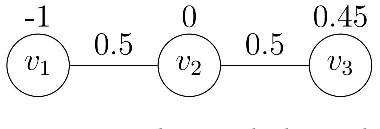


Figure 6.2: Example of a situation where a balanced node becomes unbalanced

The graph has 3 vertices, v_1, v_2, v_3, with $d(v_1, v_2) = 0.5$ and $d(v_2, v_3) = 0.5$. For the constant λ we will use the value 0.1. In the current state, $l_1(v_1) = -1$, $l_1(v_2) = 0$ and $l_1(v_3) = 0.45$. In this state, only the edge (v_1, v_2) is unbalanced, because $|l_1(v_1) - l_1(v_2)| = 1$, while $d(v_1, v_2) = 0.5$. The edge (v_2, v_3) is balanced, since the label difference is only 0.45, while the distance between the nodes is 0.5. In this iteration, $\max\limits_{u \in \overline{B}(V, l_1)} l_1(u) = l_1(v_2) = 0$.

In the current step of the algorithm, a message with the value -1 will be transferred from v_1 to v_2 and a message with value 0 will be transferred from v_2 to v_1. The new labels will be:

$$l_2(v_1) = (1 - \lambda) \cdot l_1(v_1) + \lambda \cdot l_1(v_2) = -0.9$$
$$l_2(v_2) = (1 - \lambda) \cdot l_1(v_2) + \lambda \cdot l_1(v_1) = -0.1$$

The label for v_3 will not change because v_3 was balanced, so $l_2(v_3) = l_1(v_3) = 0.45$. After the label for v_2 changes, we will have $|l_2(v_2) - l_2(v_3)| = |-0.1 - 0.45| = 0.55 > d(v_2, v_3)$, so v_3 becomes unbalanced. For the new labeling, $\max\limits_{u \in \overline{B}(V,l_2)} l_2(u) = l_1(v_3) = 0.45 > 0$.

Despite this shortcoming, Lemma 3 gives the following results:

Corollary 1. *In each iteration of Algorithm 21, either the maximum value for an unbalanced node decreases by at least a constant c, either a previously balanced node becomes unbalanced.*

Lemma 4. *If the graph has only two nodes ($|V| = 2$), Algorithm 21 terminates.*

Proof. A graph cannot have a single unbalanced vertex, since an unbalanced node involves an unbalanced edge and an edge has two vertices. This means that at any given time, either both nodes are unbalanced, either both are balanced. In the second case, the algorithm terminates so there is nothing left to prove.

While both vertices are unbalanced, according to Lemma 3, their maximum label decreases by at least a constant c. This means that after at most $\left\lceil \frac{2}{c} \right\rceil$ steps, the algorithm either finishes or the maximum label decreases by 2 (reaching -1). Let's denote the two vertices of the graph with v_1 and v_2, with $l(v_1) \leq l(v_2)$. Since the minimum value a label can have is -1, we have $-1 \leq l(v_1) \leq l(v_2) \leq -1$, so $l(v_1) = l(v_2)$ and the graph becomes balanced.

Note the importance of the constant c in Lemma 3: if we managed to prove that the maximum value decreases, but not by a constant factor, then the maximum value can decrease in step k by $\frac{\alpha}{2^k}$. Since $\sum\limits_{k=1}^{\infty} \frac{\alpha}{2^k} = \alpha$, the maximum value will not decrease by a value larger than α after any finite number of steps. $\square$

Finally, we can prove the theorem by induction after $n = |V|$, the number of nodes in the graph.

The basis case, for $n = 2$ follows from Lemma 4.

The inductive step: knowing that Algorithm 21 terminates for any graph of size n after a finite number of steps, prove that it also terminates for any graph of size $n + 1$.

Let $G = (V, E)$ be a graph with $|V| = n + 1$. During the execution of Algorithm 21, the graph can be in one of the following two states:

1. a vertex with the maximum label value is balanced

2. any vertex with the maximum label value is unbalanced

After each iteration of the algorithm, the graph can switch or remain in the same state.

If the graph is always in state 1, the balanced vertex will not participate at all in Algorithm 21, so the algorithm will run only on the remaining n nodes. Using the induction hypothesis, the algorithm terminates for any graph of size n, so it will also terminate in this case.

If the graph switches eventually to state 2, according to Lemma 3, the label with the maximum value l_M will decrease by at least a constant c. If any other vertices v remain, with a label $l(v)$, such that $l_M - c < l(v) < l_M$, they will either become unbalanced at some point, or they will never participate in the algorithm, so by induction hypothesis, it terminates. After such a vertex becomes unbalanced, we use again Lemma 4 to prove that its label will get smaller than $l_M - c$. So, after a finite number of iterations (that we will call a step), the algorithm either terminates or the maximum label value in the entire graph decreases by a constant c. After at most $\left\lceil \frac{2}{c} \right\rceil$ steps, the maximum label value decreases to -1, so the algorithm must terminate. □

The fact that the algorithm terminates shows that there always exists a verdicts assignment for a cluster of samples, such that the graph is balanced. Indeed, if we use the same verdict (the same label) for every sample, the graph will definitely be balanced because the absolute difference between the labels of every two nodes will be 0. A trivial solution is then, to assign the same verdict to each sample.

We will design a fitness function that evaluates how well did the verdicts assignment task performed. Informally, we can state that a verdicts assignment is good if the graph becomes balanced and the verdicts are not very different from the ones we started with. Formally, we will compute the root mean square of all these differences, as in Equation 6.8. We will denote by l_0 the initial labels and by l the final labels, such that the graph is balanced.

$$F(l, l_0) = \sqrt{\frac{\sum_{v \in V} (l(v) - l_0(v))^2}{|V|}} \tag{6.8}$$

The smaller $F(l, l_0)$ is, the better is the labeling l.

As we stated, before, a trivial solution is to assign the same value for each label, $l(v) = x, \forall v \in V$. In this case, Equation 6.8 becomes:

$$F(l, l_0) = \sqrt{\frac{\sum_{v \in V} (x - l_0(v))^2}{|V|}}$$

The minimum value for this function is achieved when the numerator of the fraction, $F_1(x) = \sum_{v \in V} (x - l_0(v))^2$ achieves its minimum. x can be found by solving the equation $\frac{\partial F_1}{\partial x}(x) = 0$.

$$\begin{aligned} \frac{\partial F_1}{\partial x}(x) &= 0 \\ \iff \sum_{v \in V} 2(x - l_0(v)) &= 0 \\ \iff |V| \cdot x &= \sum_{v \in V} l_0(v) \end{aligned}$$

$$\Rightarrow x = \frac{\sum_{v \in V} l_0(v)}{|V|} \tag{6.9}$$

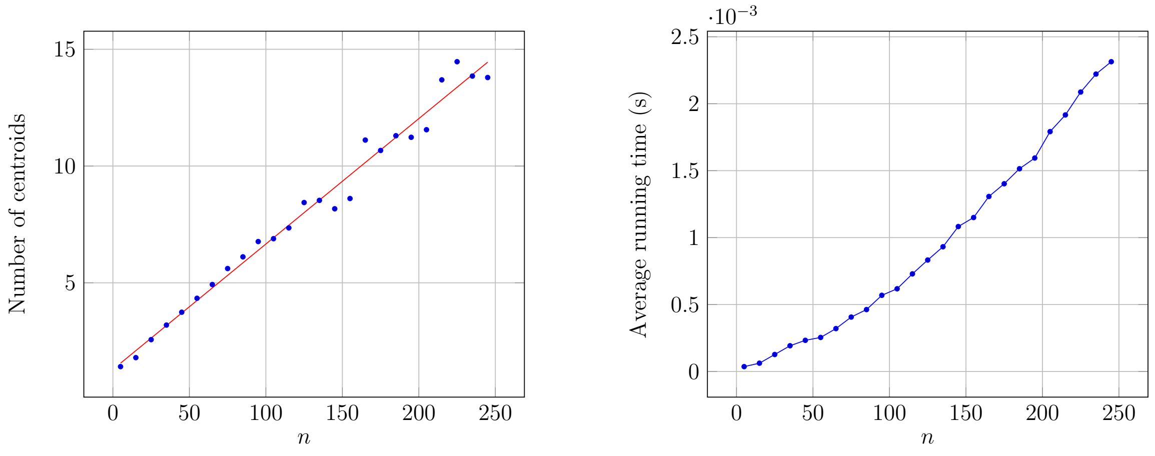
The fitness value when all the labels have the same value x computed above will be called the trivial fitness. In the following section we will show experimentally that the fitness value obtained by Algorithm 21 is better than the trivial fitness.

6.3 Experimental Evaluation

6.3.1 Experimental results on multi-centroid cluster analysis

We have tested the algorithm presented in the first section of this chapter on a collection of more than one million clusters built using the single linkage approach from more than 20 million unique samples. The tests ran on a machine with Intel i7 vPro processor at 2GHz with 8GB of RAM.

For Algorithm 20 we are interested on how much the human effort was reduced, by computing the size of the centroids for different cluster sizes. Since the number of centroids is influenced by the cluster shape, not only by the cluster size, we have split the clusters into groups of similar sizes and computed the average number of centroids for each group. Figure 6.3 shows the results of this algorithm, in terms of cluster sizes and performance.



(a) Number of centroids found (b) Running time of the algorithm

Figure 6.3: Analysis of the results for Algorithm 20

The average number of centroids found by Algorithm 20 in a graph of size n is shown in Figure 6.3a. Although the trend line shows a linear dependency, the slope is only $5.37 \cdot 10^{-2}$ so the amount of samples in cluster that require manual analysis is greatly reduced.

Figure 6.3b confirms the theoretical analysis of the algorithm's complexity and shows its quadratic running time. For most clusters, finding the centroids requires less than two milliseconds.

6.3.2 Experimental results on semi-automated verdicts assignment

We have tested the semi-automated verdicts assignment algorithm on a collection of more than 200000 clusters built using the single linkage approach from more than 20 million unique samples.

The only theoretical result we have is that the algorithm terminates (from Theorem 3) so we are interested at least in an empirical estimation of the performance.

For various transfer factors $\lambda \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$, Figure 6.4 plots the number of iterations of the algorithm against the graph size n.

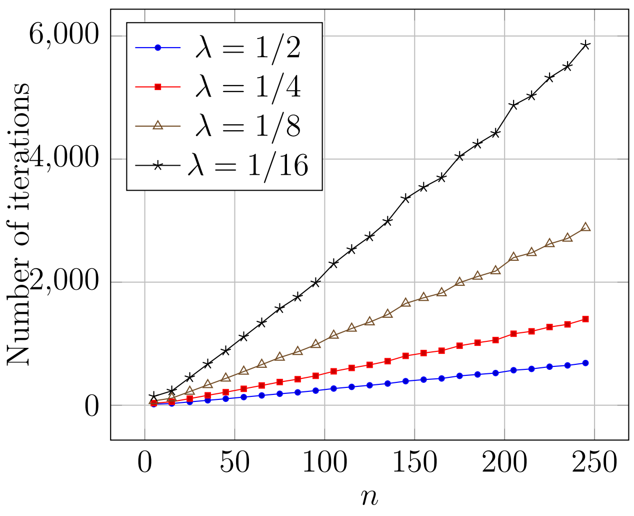


Figure 6.4: Number of iterations for various λ

Two observations can be made for this figure:

- For a fixed λ, the number of iterations is linear in n.
- The number of iterations is higher for smaller values of λ.

In order to observe the correlation between $\frac{n}{\lambda}$ and the number of iterations we have drawn the plot in Figure 6.5. Empirical evidence also suggest linearity.

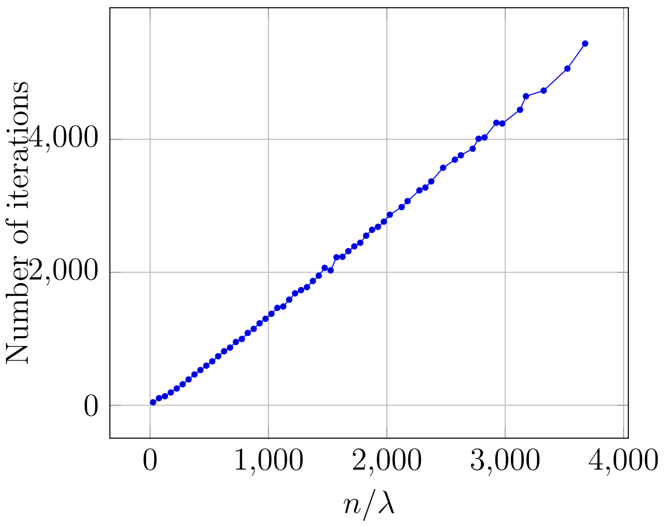


Figure 6.5: Number of iterations against $\frac{n}{\lambda}$

The previous results showed that decreasing λ, the performance of the algorithm drops (the number of iterations increases). Figure 6.6 shows that smaller values for λ lead to better results, as the value of the fitness function decreases for smaller transfer factors. Figure 6.6 also shows the value for the trivial fitness, obtained by assigning each sample the same label. Regardless of the choice of λ, Algorithm 21 always obtained better results.

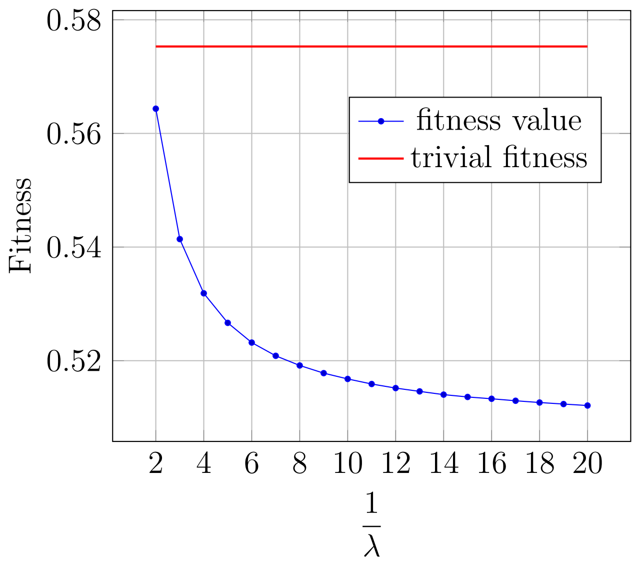


Figure 6.6: Fitness value for $n = 100$, while varying λ

Experimental results show that the choice of the transfer factor should be a trade-off between performance and the quality of the results.

6.4 Chapter Conclusions

This chapter showed some solutions for reducing the amount of manual work performed by malware analysts while classifying large collections of potentially malicious samples.

The first issue was how to select a subset of representative samples from a cluster of similar ones, such that every other sample is similar with at least a sample that was selected. The problem was reduced to the *minimum dominating set*, a known NP-hard problem that allows some approximating algorithms.

The other issue was to find a model where partial information about a set of samples can be used to infer verdicts for each of them. For the chosen model, we presented an algorithm that infers a verdict for the samples where such an inference is possible or requires manual analysis where it is not possible.

Experimental results showed that the number of iterations for the designed algorithm is linear in the graph size, so it has an overall polynomial complexity (this fact was not yet formally proven).

The following contributions were presented in this chapter:

- a system for selecting the most representative samples from a cluster for advanced analysis;
- a technique for inferring the verdicts for new samples based on their similarity with known samples;

The chapter is based on the following published work:

- **Ciprian Oprișa**, George Cabău, and Gheorghe Sebestyen Pal. Multi-centroid cluster analysis in malware research. In *EVOLVE 2015 International Conference*, Iasi, Romania, 2015. ([OCSP15a])
- **Ciprian Oprișa**, George Cabău, and Gheorghe Sebestyen Pal. Semi-automated verdicts assignment for potentially malicious programs. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, Cluj-Napoca, Romania, 2015. IEEE.([OCSP15b])

Chapter 7

A Scalable Approach for Detecting Plagiarism

This chapter describes a complete system able to detect plagiarism cases in Android markets. Although the chapter describing software similarity concepts dealt with the plagiarism issue, the scalability was not a concern. On real-world markets of applications like Google Play, the amount of samples exceed one million [War13]. Computing pairwise similarities for finding plagiarism cases would be equivalent to searching the needle in the haystack. We propose new data structures and algorithms that are able to:

- retrieve applications that are similar to a given one
- retrieve all clusters of applications that are similar to each other and weren't developed by the same entity

Our system is able to output pairs of applications suspected of plagiarism, where further analysis (manual or automatic) can confirm or infirm the verdict.

7.1 Attack Vectors for Plagiarizing Mobile Applications

This section will summarize the attack techniques used for plagiarizing Android applications. We will take into account the entire applications market environment, since a simple comparison between applications is not enough. For instance, we will assume that the goal of the attacker is to make profit. If the cost and effort for plagiarizing and publishing the cloned application is comparable to the cost and effort for developing a new legitimate one, the attack is not likely to happen. Even if this cost is a fraction (like 10%) of the development cost for a new app, it is still more profitable to develop than to plagiarize, since cloned apps are usually short-lived (they are soon reported and taken down from the market).

Another valid reason for reusing the code of an existing application is to spread malware. According to [ZJ12], 86% of the malware samples are "repackaged versions of legitimate applications with malicious payloads". This means that detecting repackaged applications will help discover new malware.

In 2007, Roy and Cordy wrote a comprehensive survey [RC07] on software cloning. Although their scope goes beyond Android and mobile applications, their taxonomy can successfully be applied in our case. The authors of the survey identified four types of cloning, the first three being based on textual similarity, while the last one considers functional similarity:

- Type I: “Identical code fragments except for variations in whitespace and comments”. Since whitespace and comments are not considered when producing bytecode, no plagiarism detection system that relies on already compiled applications will be affected by this type.

- Type II: “Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments”. Some plagiarism detectors take into account the name of methods, classes or packages (like the Symbol-Coverage technique from [PNNRZ12]). Official tools like ProGuard [L$^+$09] allow developers to eliminate these types of literals from their applications in order to make them smaller and more resistant to reverse engineering. Our proposed system will not take into account the literal names, so it is unaffected by this type of cloning.

- Type III: “Copied fragments with further modifications”. Besides cloning, code-level modifications are introduced manually or automatically. PANDORA [PM13] is a proof-of-concept tool designed to apply obfuscation techniques to the Android bytecode in order to make it different from a syntactic point of view. Depending on the obfuscation level, similarity functions based on n-grams or Abstract Syntax Tree are still able to detect common elements between the original and the plagiarized application. The difficulty mainly consists in balancing the true positives and the false positives rate. A small similarity threshold will detect more plagiarism cases but will also flag some legitimate apps as being plagiarized. A high similarity threshold will avoid most of the false positives but will also miss some of the clones.

- Type IV: “Two or more code fragments that perform the same computation but implemented through different syntactic variants”. It is generally undecidable [Tur36] if two programs produce the same output on any given input. Even human experts will not always agree if a pair of samples is a plagiarism case or just a reuse of the same idea.

Since the approach in this section is based on the binary code found in the *classes.dex* file, it is not affected by Type I and Type II cloning. Sometimes the delimitation between Type IV and Type III clones might not be clear, as one would need to decide whether a piece of code is re-implemented or simply obfuscated. In what follows, we will consider Type IV changes only those changes performed by humans, while code-level changes performed by automated system will be classified as Type III, regardless of their sophistication. Type IV clones are difficult to detect using only statical analysis and difficult to prove. Although the general problem of identifying Type IV clones is equivalent to the halting problem [Tur36], there are particular cases where automated analysis can still give valuable insights. If only some parts of the code are re-written, while the others are left intact or simply obfuscated, the remaining parts can still trigger a high similarity score. The quantity of manually re-written code is also correlated with the development cost for the attacker: making few changes would be cheap, but the risk of being detected is high, while making too many manual changes would be impractical. Even if all the methods of an application are re-implemented, the clone can still be detected using structural features, if the classes structure and hierarchy is left unchanged.

In a 2012 paper [PNNRZ12], the authors also described two levels of obfuscation: level 1 only addresses changes to the symbol table (this corresponds to a type II cloning from Roy’s taxonomy), while level 2 considers a number of added methods with no functionality (this partially covers type III). According to the authors, only level 1 obfuscation has been encountered in practice.

Detecting type III clones depends on the nature and the number of changes applied to the original code. The approach in this paper is based on OpCodes n-grams, a method robust to most of these changes. Bellow, we will enumerate the type III cloning approaches from [RC07] and see how the current approach handles them.

- *near-miss clones*: non-identical code fragments that keep the syntactical structure. Since the OpCode includes only the operation to be performed, not the operands, a code fragment will be abstracted as a sequence of operations, that will remain identical, in this case.

- *gapped clones* and *non-contiguous clones*: some code segments are inserted or deleted. The success of the n-grams approach is highly dependent on the amount of changes. If enough segments of at least n consecutive instructions are preserved, the clone will still be detected. If a significant amount of code is altered this way, the clone will evade detection, but the effort for the attacker will also be higher. A clone with a large number of changed code fragments may also be categorized as type IV clone, which is out of our scope.

- *structural clones* and *function clones*: the syntactic structure of the program is modified. In the Android case, some classes can be moved from one package to another, or some methods can be moved from one class to another. Also, package and classes can be split or joined. Since our abstraction considers an application to be a set of methods, such structural changes do not modify the abstract view. The only changes that can affect our approach is to move blocks of code between methods.

- *reordered clones*: the order of some code segments is changed, such that the code semantics is preserved. Two methods will be compared as unordered sets of n-grams. The only n-grams that will differ will be the ones that cross the border between two segments. For instance, if the sequence of operations $ABCDEFGH$ is changed into $EFGHABCD$ (the two halves are swapped), and we extract 2-grams ($n = 2$), the only different 2-gram will be DE, that will be replaced by HA, while the remaining 2-grams (AB, BC, CD, EF, FG and GH) will remain the same.

The applications similarity functions in the next subsection will output a similarity score, based on the amount of common code, between two applications. Since type I and II clones produce identical features, the similarity score will be 100%, so there will be no issue in detecting them. For type III clones, however, the similarity threshold (the lower bound, above which a pair of applications will be suspected as plagiarism) must be chosen such to maximize detection while avoiding false positives. The problem is that many Android applications contain a large amount of library code. It is not uncommon for an Android application to be com prised of more than 90% (sometimes even more than 99%) library code. Two different applications may look very similar because they are using the same library, so even if we set the threshold at 90%, we can't avoid the false positives.

Our approach will try to identify library code and disregard it while computing the similarity score. Some library code was labeled manually, while most of it was identified because it appeared in multiple applications from different developers. A possible attack against this method is to publish multiple plagiarized versions of the same application under different developer IDs. If enough different versions will be found in the market, an automated system will automatically label the entire application code as library and will not identify further plagiarism cases. This attack is viable, since creating a new developer account is relatively inexpensive (25$). A mitigation to this attack is to assign a reputation score to each developer, based on the number of application he published, the user ratings for those applications and the amount of time he has been in the market. A code fragment will only be flagged as library if the sum of the reputation values for the developers that use it exceeds a threshold. This will not make the attack impossible, but will increase the cost for the attacker enough for the plagiarism not to be profitable.

7.2 Applications Similarity Functions

This section deals with the similarity issue and proposes two methods for computing the similarity between applications: shallow similarity and deep similarity. Both methods are based on the applications code and use the concept of n-grams discussed in a previous chapter.

A similarity function is a function that takes as input two applications and outputs a real number between 0 and 1, as in Equation 7.1. If $sim(A_1, A_2) = 1$ it means that the two applications have the same non-library code, while a similarity of 0 means that they are completely different (except for library code).

$$sim : \mathcal{A} \times \mathcal{A} \to [0, 1] \tag{7.1}$$

Definition 5. *We will say that two applications $A_1, A_2 \in \mathcal{A}$ represent a **plagiarism case** iff $sim(A_1, A_2) \geq \theta_p$. $\theta_p \in [0, 1]$ is a constant called plagiarism threshold.*

The set $\mathcal{A}$ is the set of all applications in a collection. An application is represented as a set of methods: $A = \{M_1, M_2, \ldots, M_k\}$ $M_i \in \mathcal{M}, \forall i \in \overline{1, k}$. The set $\mathcal{M}$ is the set of all the unique methods found in the collection's applications. We will also represent each method as a set of n-grams. An n-gram, as discussed in the first section is a sequence of n consecutive OpCodes from a method. The set of all n-grams from the collection's methods will be denoted by $\mathcal{G}$.

7.2.1 Publishers Identification and Library Code

In order to publish an application in the Android market developers need to create an account [And14a]. Currently, there is a one time fee of 25$ for creating a new account, "to encourage higher quality products on Google Play (i.e. less products with SPAM)". A developer may publish as many applications as he wants from the same account. An application is uniquely identified by a package name and is self-signed with a "certificate whose private key is held by the developer" [And14b]. In order to update an existing application, the new version must have the same package name and be signed with the same certificate.

Since the publisher name can be easily changed in the developer's account, we will rely on the following two facts when identifying the publishers:

- If two applications have the same package name, they belong to the same publisher.
- If two applications are signed with the same certificate, it means that the signers had the same private key so they must be the same entity.

The above assumptions don't hold if the private key of a publisher is leaked or if the certificate's algorithm is insecure. On this matter, we have identified several applications that are still signed using the MD4 hash algorithm that was already proved to be insecure [Leu08]. However, most applications use MD5 or SHA for hash in their certificate.

In what follows, we will consider $pub : \mathcal{A} \to \mathbb{N}$, a function that assigns a publisher id to each application. If $pub(A_1) = pub(A_2)$, it means that the applications A_1 and A_2 have the same publisher.

We will now give the definition for *library code*. Informally, we will consider library code a piece of code that is widely used, usually by many publishers. To improve the system's accuracy, some n-grams and methods may be manually marked as library code. The following definitions will formally illustrate the concepts of library n-grams and library methods.

Definition 6. *The set of library n-grams is a set $\mathcal{G}_L \subset \mathcal{G}$ that contains all the n-grams found in at least $\theta_{\mathcal{G}L}$ different methods and the n-grams manually marked as library code, as in Equation 7.2. The set of non-library n-grams will be denoted by $\mathcal{G}^\star = \mathcal{G} \setminus \mathcal{G}_L$.*

$$\mathcal{G}_L = \{g \in \mathcal{G} \mid |\{M \in \mathcal{M} \mid g \in M\}| \geq \theta_{\mathcal{G}L}\} \cup L_{\mathcal{G}} \tag{7.2}$$

$\theta_{\mathcal{G}L}$ is a chosen threshold for library n-grams and $L_{\mathcal{G}}$ is the set of manually chosen library n-grams. For performance reasons, we won't associate the n-grams with publishers and will relay only on the number of methods they appear into.

Definition 7. *The set of library methods is a set $\mathcal{M}_L \subset \mathcal{M}$ that contains all the methods found in at least $\theta_{\mathcal{M}L1}$ different applications or used by at least $\theta_{\mathcal{M}L2}$ publishers and the methods manually marked as library code, as in Equation 7.3. The set of non-library methods will be denoted by $\mathcal{M}^\star = \mathcal{M} \setminus \mathcal{M}_L$.*

$$\begin{aligned}\mathcal{M}_L =& \{M \in \mathcal{M} \mid |\{A \in \mathcal{A} \mid M \in A\}| \geq \theta_{\mathcal{M}L1}\} \\ & \cup \{M \in \mathcal{M} \mid |\{pub(A) \mid A \in \mathcal{A} \wedge M \in A\}| \geq \theta_{\mathcal{M}L2}\} \\ & \cup L_{\mathcal{M}}\end{aligned} \tag{7.3}$$

$\theta_{\mathcal{M}L1}$ and $\theta_{\mathcal{M}L2}$ are a chosen thresholds for library methods and $L_{\mathcal{M}}$ is the set of manually chosen library methods.

In order to mitigate the attack described in the previous section, where a plagiarized application can be re-published using multiple accounts so that the entire code will be flagged as library code, we can alter Equation 7.3 in the following way:

$$\begin{aligned}\mathcal{M}_L =& \{M \in \mathcal{M} \mid |\{A \in \mathcal{A} \mid M \in A\}| \geq \theta_{\mathcal{M}L1}\} \\ & \cup \left\{ M \in \mathcal{M} \;\middle|\; \sum_{P \in \{pub(A) \mid A \in \mathcal{A} \wedge M \in A\}} reputation(P) \geq \theta_{\mathcal{M}L2} \right\} \\ & \cup L_{\mathcal{M}}\end{aligned} \tag{7.4}$$

Basically, in Equation 7.4, instead of counting the number of distinct publishers, we add their reputation values. The reputation function is computed for a given publisher based on the user ratings and the number of downloads for their published apps. The cost for creating a large number of high-rated user accounts is high enough to discourage this attack. The threshold $\theta_{\mathcal{M}L1}$ is larger than the highest number of applications published by a single developer. This way, an attacker cannot force our system to flag all the application's methods as library code by publishing $\theta_{\mathcal{M}L1}$ versions of it using a single account. In theory, it is possible to get beyond this threshold using several accounts and publish a large number of clones with all of them, but this kind of behavior may raise other suspicions.

In the rest of this paper we will consider that each application in our collection has a reasonable number of non-library methods. An application with too few non-library methods (less than a given threshold) will be excluded from the collection $\mathcal{A}$ as we cannot find similar applications based on its extracted code.

7.2.2 Shallow Similarity

After formally defining the concept of library code in the previous subsection, we can now define the shallow similarity between two applications in Equation 7.5.

$$\begin{gathered} ssim : \mathcal{A} \times \mathcal{A} \to [0,1] \\ ssim(A_1, A_2) = \frac{|\{M \in \mathcal{M}^\star \mid M \in A_1 \wedge M \in A_2\}|}{|\{M \in \mathcal{M}^\star \mid M \in A_1 \vee M \in A_2\}|} \end{gathered} \tag{7.5}$$

In other words, the shallow similarity is the Jaccard similarity between the sets where the library methods have been filtered out. It computes the ratio between common non-library methods and total non-library methods.

Let $A \in \mathcal{A}$ be an application. If we denote by $A^\star \subset A$ the subset of non-library methods from A, $A^\star = \{M \in A \mid M \in \mathcal{M}^\star\}$, Equation 7.5 can be rewritten in a simpler way, in Equation 7.6.

$$ssim(A_1, A_2) = \frac{|A_1^\star \cap A_2^\star|}{|A_1^\star \cup A_2^\star|} \tag{7.6}$$

One disadvantage of this similarity function is that it assumes the attacker will leave most of the application's code unaltered. If the attacker uses an obfuscation tool that slightly modifies the code of each method so at least an n-gram will be altered, the shallow similarity will fail to recognize the plagiarism. Internally, a method is stored as a hash on the sorted list of n-grams, so the method hashes won't be the same.

In [PNNRZ12] it is argued that although advanced obfuscation methods exist, "their applicability to mobile applications remains unknown due to the specific byte-code format and the tight resource and energy constraints". Their model allows the attacker to add or remove some methods but not to perform modifications at the method level.

For the cases where method-level modifications are performed, our system is still able to detect the plagiarism cases, by using the deep similarity presented in the next subsection.

If the average number of non-library methods in an application is m (we have already denoted with n the number of consecutive OpCodes in the n-grams) and the applications are represented as sorted lists, the computation complexity is $O(m)$.

7.2.3 Deep Similarity

Deep similarity goes beyond method-level and tries to match the non-identical methods between the applications based on their n-grams.

First of all, we will define the similarity between two methods in the same way the shallow similarity between two applications was defined.

$$msim : \mathcal{M}^\star \times \mathcal{M}^\star \to [0,1]$$
$$msim(M_1, M_2) = \begin{cases} 0, \text{ if } |\{g \in \mathcal{G}^\star \mid g \in M_1 \cap M_2\}| = 0 \\ \dfrac{|\{g \in \mathcal{G}^\star \mid g \in M_1 \cap M_2\}|}{|\{g \in \mathcal{G}^\star \mid g \in M_1 \cup M_2\}|}, \text{ otherwise} \end{cases} \tag{7.7}$$

The *msim* function in Equation 7.7 computes the similarity between two methods using the Jaccard similarity function on the non-library n-grams. Notice that a method may have all the n-grams in $\mathcal{G}_L$ (library n-grams) yet not be a library method. If we need to compute the similarity between two such methods, the denominator of the fraction would be 0. For this reason, the first branch of the equation states that if the two methods have no common n-grams, their similarity is be 0.

For practical purposes, we may consider the similarity between two methods only if it is above a certain threshold (θ_m). In this case, we will use $msim'$ instead of $msim$:

$$msim'(M_1, M_2) = \begin{cases} msim(M_1, M_2), \text{ if } msim(M_1, M_2) \geq \theta_m \\ 0, \text{ otherwise} \end{cases} \tag{7.8}$$

If we take a closer look at Equation 7.5 and Equation 7.6, we observe that the difference between the numerator and the denominator of the fractions consists in the non-common methods from both A_1 and A_2. Equation 7.6 can be further rewritten as:

$$ssim(A_1, A_2) = \frac{|A_1^\star \cap A_2^\star|}{|A_1^\star \cap A_2^\star| + |A_1^\star \setminus A_2^\star| + |A_2^\star \setminus A_1^\star|} \tag{7.9}$$

Because shallow similarity works at method-level, it won't take into account matches between methods from $A_1^\star \setminus A_2^\star$ and methods from $A_2^\star \setminus A_1^\star$ that have common n-grams.

Given two sets of methods $X, Y \in \mathcal{M}^\star$, with $X \cap Y = \emptyset$, we can associate each pair (x, y) with $x \in X$, $y \in Y$ with a weight, $w(x, y) = msim(x, y)$. At this point, we can find a matching between the two sets of methods by solving the **maximum weighted bipartite matching** problem (also called the assignment problem). Several polynomial algorithms exist for this task, the most notable one being the Hungarian algorithm [Kuh55].

The best match value between the sets X and Y will be denoted by $bm(X, Y)$ and is expressed in Equation 7.10.

$$\begin{gathered} bm : \mathcal{M}^\star \times \mathcal{M}^\star \to \mathbb{R}_+ \\ bm(X, Y) = \max \sum_{i=1}^{\min(|X|,|Y|)} msim'(x_i, y_i), \\ x_i \in X, y_i \in Y, \\ x_i \neq x_j, y_i \neq y_j, \forall i \neq j \end{gathered} \tag{7.10}$$

The best match value between the non-common methods of the two applications A_1 and A_2 can be added to the fraction's numerator in Equation 7.6 to compensate for the non-identical function matches. This new similarity function will be called deep similarity and will be expressed in Equation 7.11.

$$\begin{aligned} dsim &: \mathcal{A} \times \mathcal{A} \to [0,1] \\ dsim(A_1, A_2) &= \frac{|A_1^\star \cap A_2^\star| + 2 \cdot bm(A_1^\star \setminus A_2^\star, A_2^\star \setminus A_1^\star)}{|A_1^\star \cup A_2^\star|} \end{aligned} \tag{7.11}$$

The best match value at the numerator is scaled by 2, because each pair of non-common methods $M_1 \in A_1$, $M_2 \in A_2$ contributes with 2 to the denominator while their match can contribute to the numerator with less than 1.

Unlike the shallow similarity, deep similarity is able to identify plagiarism cases even if the attacker performs changes in the code of each method. The drawback is that the deep similarity function is harder to compute. If the average number of non-library methods in an application is m, the complexity of the Hungarian algorithm that finds the best match is $O(m^3)$, which is considerably higher than $O(m)$, the cost for computing the shallow similarity.

7.3 System's Architecture and Algorithms

The previous section presented the two similarity functions for Android applications. Shallow similarity is faster to compute while deep similarity is more robust to obfuscations and both of them work well when they need to check a pair of applications for plagiarism. Unfortunately, the real-world problems are more difficult than this. One common use-case is to add a new application to the system and search for all similar applications belonging to different publishers. Another one would be to output all the plagiarism cases found in the collection. The system also need to scale well and deal with a collection of millions of applications.

7.3.1 Data Model

In order to satisfy the scalability requirement we have used a NoSQL database. Our choice was MongoDB [Cho13]. The data is store into 4 main collections:

- *ApkToMet* maps the relation between applications and methods. Each item is a document that stores the application id and the sorted list of method hashes in the application. The application's publisher id is also stored here.
- *MetToNgr* maps the relation between methods and n-grams. For each method, we will store a sorted list with all the method's n-grams.
- *MetToApk* is the reversed index for the *ApkToMet* collection. If a method is non-library, the collection will hold all the applications that contain it. Otherwise, a boolean field of the document will store the fact that this is a library method.
- *NgrCount* is a simple collection that counts how many times each n-gram appears in methods.

The reversed index (*MetToApk* collection) is used for finding similar applications without checking every item in the *ApkToMet* collection.

Instead of storing a mapping form n-grams to methods, the last collection will store a simple count for each n-gram. Storing an entire list of methods for each n-gram would take up too much space. We will show that we can still find methods similar with a given one without using a reversed index, by employing the locality-sensitive hashing technique.

The ER diagram (Entity-Relationship) is presented in Figure 7.1.

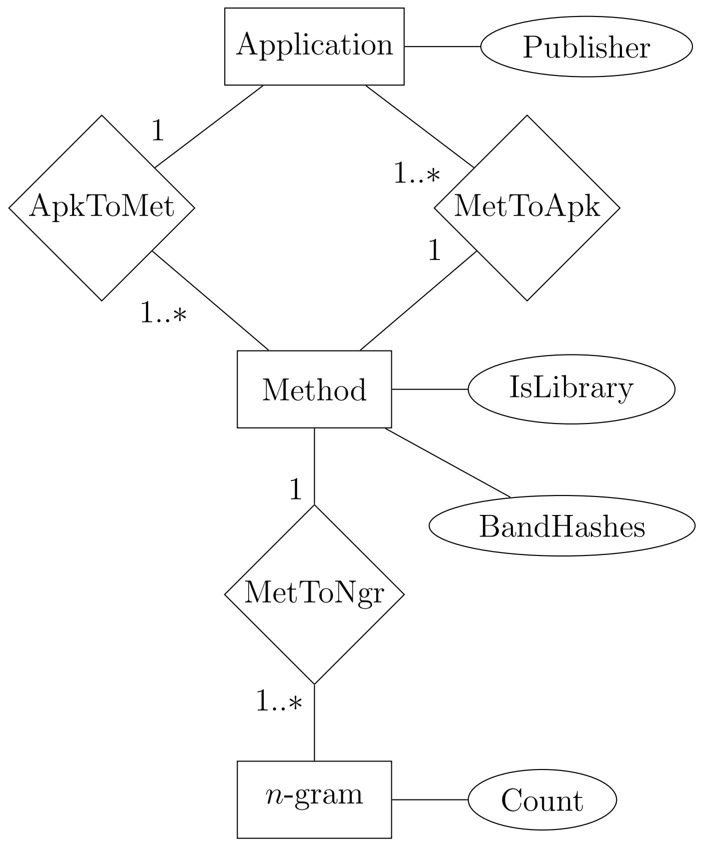


Figure 7.1: Data Model ER diagram

7.3.2 Database Construction

Building the database presented in the previous subsection from a raw collection of applications is no easy task, due to the size of the data. We will use the Map-Reduce model [DG08] for dividing this task into smaller tasks.

This model involves two functions: MAP and REDUCE. MAP takes as input a raw data item and produces key-value pairs, through the function EMIT. The Map-Reduce framework groups all these pairs by the key then passes to each reducer a key and the list of values that were emitted for that key. The REDUCE function should implement the processing of these value lists for each key. After the reduce phase, the algorithm may finish or use the results for another map-reduce stage.

Our implementation uses two map-reduce stages for building the entire database, as in Figure 7.2.

In the first stage, each application is processed by the function MAP-1 described in Algorithm 22.

MAP-1 processes an application and extracts the methods. For each method, a hash on the sorted list of n-grams is computed by the HASH function and added to the *methodHashes* list (line 3). Each method is also emitted along with the id of the current application (line 4). After the list of method hashes has been constructed, it is inserted into the *ApkToMet* collection with the application's id as primary key.

The Map-Reduce framework groups together all the application ids emitted for the same method. The reducers from the first stage, implemented by the function REDUCE-1 in Algorithm 23 will receive a method with all the emitted application ids for it.

Algorithm 23 starts by inserting the method's n-grams in the *MetToNgr* collection,

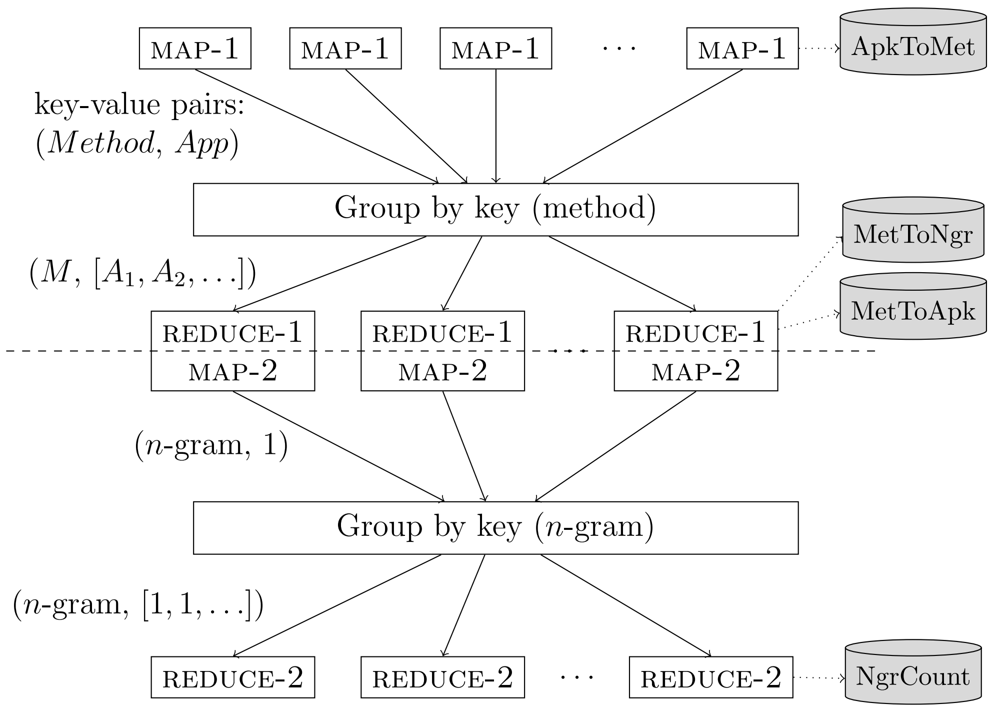


Figure 7.2: Database building

Algorithm 22 MAP-1(A)

Require: An application $A \in \mathcal{A}$
Ensure: A will be stored in the *ApkToMet* collection
Ensure: All methods will be emitted.

```
1: $methodHashes \leftarrow \{\}$
2: for all $M \in A$ do
3:     $methodHashes \leftarrow methodHashes \cup \{\text{HASH}(M)\}$
4:     EMIT($M, A.id$)
5: end for
6: $ApkToMet$.INSERT($A.id, methodHashes$)
```

with the method hash as primary key. The reason we perform the insertion at this point and not in the MAP-1 function is that the same method may belong to several applications. A unique index would ensure it won't be inserted more than once, but the search would still be performed every time. The band hashes computed on the method's n-grams are also inserted in the database at this point, in order to ensure that similar methods can be retrieved given a set of n-grams. The locality-sensitive hashing approach will be detailed in subsection 7.3.5.

Next, the method is inserted in the *MetToApk* collection. If the method is a library method ($M \in \mathcal{M}_L$), according to Definition 7 and Equation 7.3, the list of application ids won't be stored into the collection, just the information that it's a library method (line 3).

According to Figure 7.2, the reduce phase in the first stage (REDUCE-1) is immediately followed by the map phase from the second map-reduce stage (MAP-2). Since both operations work at the method level, they have been joined in Algorithm 23. The MAP-2 operation is performed in the lines 7-9 and for each n-gram $g \in M$, the value 1 is emitted

Algorithm 23 REDUCE-1(M, *Apps*)

```
Require: A method $M \in \mathcal{M}$
Require: A list of application ids $Apps$
Ensure: The method's $n$-grams will be stored in the $MetToNgr$ collection.
Ensure: The method $M$ will be stored in $MetToApk$ collection.
1: $MetToNgr.$INSERT(HASH($M$), $M$, BAND-HASHES($M$))
2: if $|Apps| \geq \theta_{\mathcal{M}L1}$ or $|$UNIQUE-PUBS$(Apps)| \geq \theta_{\mathcal{M}L2}$ then
3:     $MetToApk.$INSERT(HASH($M$), $IsLibrary = 1$)
4: else
5:     $MetToApk.$INSERT(HASH($M$), $Apps = Apps$)
6: end if
7: for all $g \in M$ do
8:     EMIT($g, 1$)
9: end for
```

for the key g. This ensures that the reducers from the second stage will count the number of containing methods for each n-gram.

Algorithm 24 REDUCE-2(g, *Lst*)

```
Require: An $n$-gram $g \in \mathcal{G}$
Require: A list of 1s $Lst$
Ensure: $g$ will be stored in $NgrCount$ collection.
1: $NgrCount.$INSERT($g$, $|Lst|$)
```

The second-stage reducers are described by the function REDUCE-2 from Algorithm 24 and perform a single operation: count how many times an n-gram was emitted and insert this information into the *NgrCount* collection. Since each time an n-gram is emitted the value 1 is used, the parameter *Lst* of REDUCE-2 will be a list containing as many values of 1 as the number of methods the n-gram g was found in.

7.3.3 Searching for Similar Items

We have established in the previous subsection that both shallow and deep similarity are easy to compute when a pair of samples needs to be checked for plagiarism. The problem gets more difficult when we are searching for all the applications in the collection that are similar with a given one or when we are searching for all the pairs of similar applications. We would like to avoid the naive algorithms where we have to compare the searched application with every other application in the first case or check every pair of applications in the second case. This subsection will propose three new algorithms for:

- finding similar items based on shallow similarity
- finding all pairs of similar items based on shallow similarity
- finding similar items based on deep similarity

Each of the following algorithms is based on the Map-Reduce model. Local MAP and REDUCE procedures will be implemented in each one of them. As discussed above, MAP will receive a raw item and emit key-value pairs, while REDUCE will be called for each key and the associate list of values and optionally return an output. The PERFORM-MAP-REDUCE function puts it all together by calling MAP on each item from the input list and return a list with all the outputs returned by the reducers. Unlike classical Map-Reduce frameworks, we have designed our system to perform fast similarity queries so fault tolerance was not an issue. This gives a better performance, since there is no need to store intermediary results on permanent storage and more flexibility, since the MAP and REDUCE functions need not to be stateless.

7.3.3.1 Finding Similar Apps Based on Shallow Similarity

Algorithm 25 FIND-SIMILARS-SHALLOW(A)

Require: An application $A \in \mathcal{A}$
Ensure: A list of applications similar with A

```
1: procedure MAP($M$)
2:     $Rec \leftarrow MetToApk.\text{QUERY}(M)$
3:     if not $Rec.IsLibrary$ then
4:         for all $B \in Rec.Apps$
5:             EMIT($B, 1$)
6:         end for
7:     end if
8: end procedure
9: procedure REDUCE($B, Lst$)
10:    $s \leftarrow \frac{|Lst|}{|A^\star| + |B^\star| - |Lst|}$
11:    if $s \geq \theta_p$ and $pub(A) \neq pub(B)$ then
12:        return $B$
13:    end if
14: end procedure
15: return PERFORM-MAP-REDUCE($A$)
```

Algorithm 25 receives an application $A \in \mathcal{A}$ and produces a list with all the similar applications that have different publishers than A, based on shallow similarity.

The MAP function is called for each method $M \in A$. The method is searched in the *MetToApk* collection (line 2). If M is a non-library method, then each application B in its list of apps is a candidate for similarity so it will be emitted (line 5).

A candidate B will be emitted once for each non-library method that is common with A, so the number of elements in the list Lst, the second parameter of REDUCE will be equal to this number: $|Lst| = |A^\star \cap B^\star|$. To compute the shallow similarity as in Equation 7.6, we will also need the size of the union, for the fraction's denominator. By applying the inclusion-exclusion principle, we have:

$$\begin{aligned} |A^\star \cup B^\star| &= |A^\star| + |B^\star| - |A^\star \cap B^\star| \\ &= |A^\star| + |B^\star| - |Lst| \end{aligned}$$

This means that the equation in line 10 correctly computes the shallow similarity between A and B ($s = ssim(A, B)$). All we have to do now is to check if this similarity is above the plagiarism threshold and that A and B have different publishers. If both conditions hold, B will be added to the list of applications similar with A.

To compute the complexity of Algorithm 25, we will assume that a database query takes constant time (if it is indexed properly) and that an average application has m methods. If $M \in \mathcal{M}^\star$, the maximum number of applications in a *MetToApk* record is $\theta_{\mathcal{M}L1}$, so the MAP function will emit at most $\theta_{\mathcal{M}L1}$ values. Since MAP is called for each method, the number of operations performed by the first phase of Map-Reduce is $O(m \times \theta_{\mathcal{M}L1})$. In the reduce phase, we need to compute $|B^\star|$, or how many non-library methods has the set B ($|A^\star|$ can be computed once then cached), so m database queries will be performed. In the worst-case scenario, the REDUCE function is called for each emitted pair, so it will be called $O(m \times \theta_{\mathcal{M}L1})$ times. Assuming that $\theta_{\mathcal{M}L1}$ is constant, the final complexity will be $O(m^2)$.

7.3.4 Finding All Pairs of Similar Apps Based on Shallow Similarity

Algorithm 26 FIND-ALL-SIMILARS-SHALLOW()

Require: The database described in the previous subsection
Ensure: *AllSims*, a list of plagiarism cases

```
 1: procedure MAP($M$)
 2:     $Rec \leftarrow MetToApk.\text{QUERY}(M)$
 3:     if not $Rec.IsLibrary$ then
 4:         for all $A, B \in Rec.Apps$, $A.id < B.id$
 5:             EMIT($(A, B), 1$)
 6:         end for
 7:     end if
 8: end procedure
 9: procedure REDUCE($(A, B), Lst$)
10:     $s \leftarrow \dfrac{|Lst|}{|A^\star| + |B^\star| - |Lst|}$
11:     if $s \geq \theta_p$ and $pub(A) \neq pub(B)$ then
12:         return $(A, B)$
13:     end if
14: end procedure
15: return PERFORM-MAP-REDUCE($\mathcal{M}$)
```

Algorithm 26 builds a list with all the plagiarism cases found in the database and is very similar with Algorithm 25. It also has a map phase that processes methods, but the inputs won't belong to a single application but will be the entire *MetToApk* collection (or the set $\mathcal{M}$). For a non-library method $M \in \mathcal{M}^\star$, each pair of applications that contain it will be emitted with the value 1, for counting (line 5).

The REDUCE function is identical with the one from Algorithm 25, with the exception that the key input is a pair, not a single application and the output list will also contain pairs.

In order to speed-up, the computations, $|A^\star|$ and $|B^\star|$ won't be computed for every reducer. Instead, the number of non-library methods for each application can be pre-computed (also with a Map-Reduce algorithm) and accessed in $O(1)$. The MAP function now considers all the pairs of apps that contain the non-library method $M \in \mathcal{M}^\star$, which number is at most $\dfrac{\theta_{\mathcal{M}L1}(\theta_{\mathcal{M}L1}-1)}{2}$ or $O(\theta^2_{\mathcal{M}L1})$. The map phase that needs to call MAP $|\mathcal{M}|$ times is the most costly part of the algorithm, since both the reduce phase and the counting of non-library methods for all applications take less operations. The complexity for Algorithm 26 is then $O(|\mathcal{M}| \times \theta^2_{\mathcal{M}L1})$ or $O(|\mathcal{M}|)$ if we consider $\theta_{\mathcal{M}L1}$ to be a constant.

7.3.5 Finding Similar Apps Based on Deep Similarity

This algorithm is split into three parts. Algorithm 27 finds similar methods with a given one, while Algorithm 28 and Algorithm 29 implement the map and reduce phases for finding similar applications based on deep similarity.

The function FIND-SIMILARS-METHODS from Algorithm 27 receives a method M and outputs a list of methods whose similarity with M is above the threshold θ_m (from Equation 7.8). Each item in the list is a pair that contains the similar method and the actual similarity score. The list is sorted by this similarity in descending order (the method with the highest similarity being the first).

The idea for finding similar methods is different from finding similar applications, because we don't have a reverse index for the *MetToNgr* collection. We will use locality-sensitive hashing [IM98] instead. Informally speaking, a locality-sensitive hash is a hash function where the collision probability for similar items is higher than the collision probability for dissimilar ones.

In [RU12] it is proved that the collision probability for a *minhash* is equal with the Jaccard similarity. A *minhash* is a function that retrieves the minimum element from a set, according to a permutation σ: $h(X) = \min_{x \in X} \sigma(x)$. A permutation σ on the set $\{0, 1, 2, \ldots p-1\}$ can be approximated by $\sigma(x) = (a \cdot x + b) \mod p$. The collision probability can be augmented with the banding technique: we will use $b \times r$ different minhash functions, grouped on b bands, each containing r rows. On each band, a band hash is computed on the results of the r minhash functions. If the Jaccard similarity between two applications is s, the probability that at least one band hash has the same value is $1-(1-s^r)^b$. Guidelines for choosing the parameters b and r were given in [OCN14].

Let $\theta_m = 0.8$. If we pick $r = 4$ and $b = 10$, the probability for two methods with similarity θ_m to have at least one common band hash is 99.48%. If the similarity between the two methods is 0.3, the probability drops to 7.81%.

In order to be able to find similar methods with a given one, we will augment the collection *MetToNgr* with the b band hashes computed on the non-library n-grams. These band hashes need to be updated periodically, because an n-gram can move from $\mathcal{G}^\star$ to $\mathcal{G}_L$ after adding new applications to the collection. If we keep indexes on all these b band hashes, we can find similar candidates by querying the collection for items with the same band hash on each band.

Algorithm 27 begins by querying the database for the current method record. A list of candidate similar methods is produced, by querying the collection *MetToNgr* for records with the same band hashes (lines 2-6). Each candidate except for the actual queried method is checked for similarity with M and if the similarity is at least θ_m, the

Algorithm 27 FIND-SIMILARS-METHODS(M, A)

Require: A method $M \in A$
Ensure: A list of methods similar with M that don't belong to A and the associated similarities

```
1: $candidates \leftarrow \emptyset$
2: $MRec \leftarrow MetToNgr.\textsc{query}(M)$
3: for $i = 1 \rightarrow b$
4:     $rs \leftarrow MetToNgr.\textsc{query}(band_i = MRec.band_i)$
5:     $candidates \leftarrow candidates \cup rs$
6: end for
7: $results \leftarrow \emptyset$
8: for all $C \in candidates \setminus A$
9:     $s \leftarrow msim(M, C)$
10:    if $s \geq \theta_m$ then
11:        $results \leftarrow results \cup \{(C, s)\}$
12:    end if
13: end for
14: SORT($results$)
15: return $results$
```

candidate and the similarity scores are added to the results list. Finally, the result list is sorted in descending order on the similarity field (line 14) and returned (line 15).

The function FIND-SIMILARS-METHODS that finds methods similar with the given parameter M is a useful tool for finding similar applications. The function that performs this task FIND-SIMILARS-DEEP is split between Algorithm 28 where we can find the map phase and Algorithm 29 that contains the reduce code.

The MAP function from Algorithm 28 receives one of the application's methods ($M \in A$) as a parameter and emits applications that are likely to be similar with A, along with a similarity weight and the information that the method is common with A or is just similar with one of A's methods.

If the method M is not a library method, each application B that contains it is emitted with the weight $w = 1$ and the information that the method is common to A and B, $common =$ **true** (line 5). For each method similar with M that doesn't belong to A and is not a library method, all applications that haven't been used so far for this method are emitted with the weight $w = 2 \cdot s$ and the information that the method is not common to A, $common =$ **false**.

The reason for choosing the weight to be twice the similarity is because the best match value is also multiplied by two in Equation 7.11. The *common* field will be used by the REDUCE function to treat differently the fact that an emit operation was performed from a common method or a similar method.

Algorithm 29 describes the second phase of the FIND-SIMILARS-DEEP function, the reducers code. The inner REDUCE function is called for each application B that has a chance of being similar with A and also receives all the emitted values for it. At this point we could compute the value $dsim(A, B)$ and decide if the applications are indeed similar or not. The problem is that this computation is costly, so we want to perform it only for applications that are very likely to be similar with A.

The denominator of the fraction in Equation 7.11 is easy to compute using the

Algorithm 28 FIND-SIMILARS-DEEP(A) - part 1

Require: An application $A \in \mathcal{A}$
Ensure: A list of applications similar with A

```
 1: procedure MAP($M$)
 2:     $Rec \leftarrow MetToApk$.QUERY($M$)
 3:     if not $Rec.IsLibrary$ then
 4:         for all $B \in Rec.Apps \setminus \{A\}$
 5:             EMIT($B, (w = 1, common = \textbf{true})$)
 6:         end for
 7:         $usedApps \leftarrow Rec.Apps$
 8:         $sims \leftarrow$ FIND-SIMILARS-METHODS($M, A$)
 9:         for all $(M', s) \in sims$
10:             $Rec \leftarrow MetToApk$.QUERY($M'$)
11:             if not $Rec.IsLibrary$ then
12:                 for all $B \in Rec.Apps$
13:                     if $B \notin usedApps$ then
14:                         EMIT($B, (w = 2 \cdot s, common = \textbf{false})$)
15:                         $usedApps \leftarrow usedApps \cup \{B\}$
16:                     end if
17:                 end for
18:             end if
19:         end for
20:     end if
21: end procedure
```

Algorithm 29 FIND-SIMILARS-DEEP(A) - part 2

```
 1: procedure REDUCE($B, Lst$)
 2:     $num \leftarrow 0, common \leftarrow 0$
 3:     for all $val \in Lst$
 4:         $num \leftarrow num + val.w$
 5:         if $val.common$ then
 6:             $common \leftarrow common + 1$
 7:         end if
 8:     end for
 9:     $s \leftarrow \frac{num}{|A^\star| + |B^\star| - common}$
10:     if $s \geq \theta_p$ and $pub(A) \neq pub(B)$ then
11:         $s' \leftarrow dsim(A, B)$
12:         if $s' \geq \theta_p$ then
13:             return $B$
14:         end if
15:     end if
16: end procedure
17: return PERFORM-MAP-REDUCE($A$)
```

inclusion-exclusion principle like we did in Algorithm 25 because we can count exactly how many common methods were emitted from the *common* filed of each emitted value (lines 5-7). Next, we will prove the following lemma:

Lemma 5. *The sum of all the emitted weights for a key B in Algorithm 28 is greater or equal than the fraction's numerator in Equation 7.11.*

Proof. Each time the application B is emitted as potentially similar with A, we either have a common method (Algorithm 28, line 5), in which case the weight is 1, either the method belongs only to B and is similar with one of A's methods (Algorithm 28, line 14).

$$\begin{aligned} num &= \sum_{key=B} w = \sum_{\substack{key=B\\ common=\mathbf{true}}} w + \sum_{\substack{key=B\\ common=\mathbf{false}}} w \\ &= |A^\star \cap B^\star| + \sum_{M\in A^\star\setminus B^\star} \sum_{M'\in B^\star\setminus A^\star} 2\cdot msim'(M,M') \\ &\geq |A^\star \cap B^\star| + 2 \sum_{M\in A^\star\setminus B^\star} \max_{M'\in B^\star\setminus A^\star} msim'(M,M') \\ &\geq |A^\star \cap B^\star| + 2\cdot bm(A^\star \setminus B^\star, B^\star \setminus A^\star) \end{aligned}$$

□

From Lemma 5 we have that the variable s computed at line 9 in Algorithm 29 is greater or equal than $dsim(A,B)$. This means that if $s < \theta_p$, B cannot be similar with A. We will only compute the real deep similarity only if $s \geq \theta_p$.

7.4 Experimental Evaluation

7.4.1 Algorithms Running Times

We have created the database described in the previous section for a collection of 1,165,942 Android samples from the Google Play market. The collection comprised mostly of free apps, but we also had a small budget (a couple of hundreds US dollars) in order to purchase the most popular paid apps.

The database was created from an initial collection of 1,065,000 samples that grew as new applications were added. The initial construction took 4 days and 15 hours. After that, several thousands new samples were added daily.

Table 7.1 shows the total running times of the three functions used for the database construction. The MAP-1 function took 76.7 hours or 69% of the total time. From this time, 94% or 72.1 hours were spent parsing the binary applications in order to extract the methods and the n-grams. Both REDUCE-1 and REDUCE-2 times also contain the time spent by the framework to build the list of values for each key.

Table 7.1: Running times for database construction

Operation	Running time (h)
MAP-1	76.7
REDUCE-1	27.7
REDUCE-2	6.46

Having the database built, the average running time for finding all pairs of similar applications based on the shallow similarity (Algorithm 26) takes 2 hours and 37 minutes.

Both searches for similar applications with a given one vary with the number of methods of the searched sample. If we use the shallow similarity algorithm, the running time takes from less than a second to a couple of seconds. Deep similarity is more costly, varying from a few seconds for small applications to a few minutes on average and more than 10 minutes for large applications.

7.4.2 Similarity Cases Found by the System

Before discussing the results found by the system, we must point out is that the similarity of two application could indicate different things:

- they both may be using the same framework but with a slightly change configuration (e.g. a framework for on-line radio but with different addresses from where the application uses the radio-stream).
- one could be an re-branded version of the other (e.g. the vendor of the first application is willingly selling it's code to another vendor; the other vendor usually only applies some UI changes to the original app (colors, skin, images, icon, texts, etc).
- finally, one could be a copy of the other one, without any previous agreement between the two vendors (plagiarism).

Although our goal is to find only plagiarism cases (the third case), the system will also output some pairs that belong to the first two cases.

The first case should be ruled out by the library code identification that was previously discussed. In practice, a popular framework that is used by enough different developers or by many different applications will be marked as library code automatically. The methods extracted from other frameworks / libraries can be manually set as library code. However, the system may still have some false alarms for different applications that use the same uncommon framework.

Unfortunately the second case is almost impossible to rule out automatically. The difference between a re-branding and a plagiarism case can consist only in some agreement between the vendors that cannot be inferred by an automatic system and sometimes not even by a human.

For the plagiarism cases we will also be interested in the monetization techniques used by the attacker. We have established in the introductory section that one hour could suffice to create a repackaged app. The 25$ payment is still needed for publication so there must be methods for getting more money out of a repackaged app. The answer lies in the modifications performed to the original application. The most common changes that can easily be applied are:

- if an Adware SDK is used, change the ad sdk unique identifier. This identifier tells the server where to put the virtual money that are earn by clicking an add. In many cases this is merely a string that can be easily replaced with another one (for example the following string represent an add id for Google AdMob SDK: "UA-99999999-9")
- add another Adware SDK – preferable one that can offer more money through advertising. This is a little bit more difficult than the first approach as it requires an

integration with the application. However, several Ad SDK offers push notification – a technique that does not require a special integration with the host app, but only a separate thread that from time to time will alert the user about different promotional offers. Further more, push notifications does not require the app that is hosting the SDK to be active for them to work. This actually makes this technique more expensive (e.g. having push notification usually means more money for the one that is hosting them) as the user can be spammed continuously with promotional messages.

- remove some components that are not required (remove some ad SDK or different payment methods that the app is using).

Table 7.2: Example of application that was plagiarized

Package	Vendor	Installs	Price
com.vectorunit.red	Vector Unit	1,000,000 – 5,000,000	~2.3 Euro
com.vectorunit.red.bdroid.a12b2f30	Jennifer Nelson	100 - 500	Free
com.vectorunit.redbbzf50	Laura Stone	1000 - 5000	Free
com.vectorunit.red.bdroid.a12b2102	Janet Stofkoper	1000 - 5000	Free
com.vectorunit.red.bdroid.a12b2100	Joshua Parker	1000 - 5000	Free

Table 7.2 shows an example of such a case. The first line represents the original application. The next ones are different copies of it. In this case the original app was not free. The copies however were free and bundled with the following Adware SDK: AirPush, StartApp. Even if all of the copies had different vendors they all share the same Ad identifier for the two adware SDK; this is a clear indicator that the same entity was behind this action. Further more, additional permission were added to the repackaged forms:

- `android.permission.VIBRATE`
- `android.permission.ACCESS_COARSE_LOCATION`
- `android.permission.ACCESS_FINE_LOCATION`
- `android.permission.ACCESS_WIFI_STATE`
- `android.permission.READ_PHONE_STATE`
- `com.android.launcher.permission.INSTALL_SHORTCUT`
- `com.android.launcher.permission.UNINSTALL_SHORTCUT`

All of these permission were required by the AdSDK. For example, `INSTALL_SHORTCUT` offers the AdSDK the possibility of adding icons to the main desktop (each added icon gives the vendor some money).

To make it even more convincing, the repackaged app had a similar icon with the original one (actually the mirror image of the original icon).

These results represent the state of the market on May 2013. The fake application were removed (they usually managed to stay up to 10 - 14 days in the market). Also the number of installs and / or price of the original application may have changed during this time. The names used by the repackaged apps for the vendors are obviously fake (most likely were chosen randomly).

It's not clear how much money the ones that created these fake apps gain however, it's likely that is more that 25$ per app. Further more, assuming that the users that installed the fake apps would have bought the original one, than the original vendor would have gain from 10,000 to 50,000 Euros.

Algorithm 26 ran on the entire collection of 1,165,942 Android samples and found 214,818 pairs of applications whose shallow similarity was above $\theta_p = 50\%$. The total number of unique samples involved was 44,675. The involved samples were also grouped into 4047 clusters using the single linkage approach [Sib73].

We have split the similar pairs by the similarity score as in Table 7.3. The first category contains the pairs with a perfect match (100% similarity on non-library methods), while the subsequent categories contain the 10% length intervals starting from 50%. The values are also illustrated in the chart from Figure 7.3.

Table 7.3: Statistics on the similar pairs found in the collection

Interval	Nr. pairs	Nr. samples
100%	18709	5587
[90% - 100%)	38920	14949
[80% - 90%)	38182	13807
[70% - 80%)	32177	15486
[60% - 70%)	40063	17599
[50% - 60%)	46767	20117

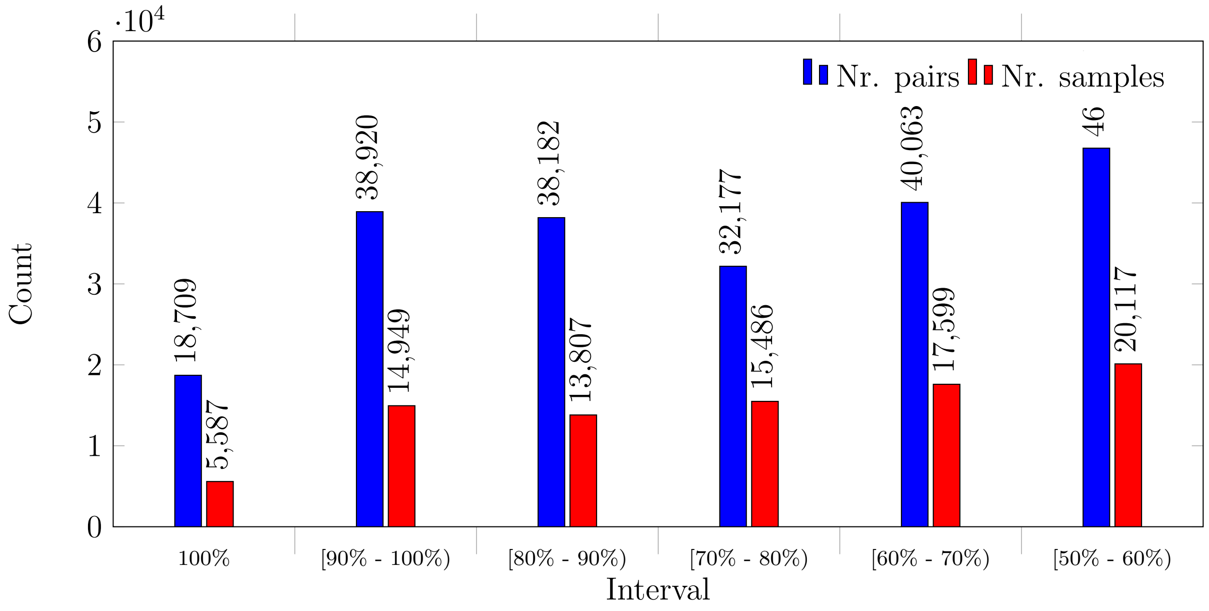


Figure 7.3: Statistics on the similar pairs found in the collection

For each interval, we have counted the number of pairs with the similarity in that interval, along with the number of unique samples involved. Since an application A can be 95% similar with an application B and 75% similar with an application C, the sum on the *Nr. samples* column is greater than the total number of unique samples involved in the pairs.

The involved samples were also analyzed dynamically for behavior that affects the user's privacy. The following undesired actions were found:

- *id*: Sending the device unique identifier on the Internet.
- *e-mail*: Sending the user's e-mail address.
- *pass*: Different user's passwords are sent in plain text.
- *location*: Sending the user's GPS location.
- *phone*: Sending the phone number.
- *contacts*: Sending the contacts from the user's phone agenda.

For each similarity interval, we have counted how many involved applications perform the actions above in Table 7.4.

Table 7.4: Statistics on the similar pairs found in the collection

Interval	*id*	*e-mail*	*pass*	*location*	*phone*	*contacts*
100%	777	24	21	74	18	0
[90% - 100%)	2451	157	88	152	69	1
[80% - 90%)	2291	157	85	133	79	1
[70% - 80%)	2702	186	75	171	92	1
[60% - 70%)	3169	192	97	178	103	1
[50% - 60%)	3643	194	102	240	102	1

Most of the actions from Table 7.4 are attributed to aggressive Adware SDKs that were introduced by the attacker to gain more financial revenue.

7.5 Chapter Conclusions

We have proposed a new approach for finding plagiarism cases in the Android applications market. Since the number of current applications is over one million, the main concern was scalability.

One of the challenges in identifying plagiarized Android applications was that 90% of the code of a typical app is library code, that is likely to be found in other applications as well. Another challenge is the fact that most Android developers reuse the code between their apps, but we don't want to flag two applications belonging to the same developer as plagiarized. To address this challenges, the proposed system is able to identify the applications publishers and to detect and eliminate from analysis the library code.

Two similarity metrics, shallow similarity and deep similarity can be used to compute how similar two applications are. They are both based on feature extracted from the code, but the shallow similarity works with methods hashes while the deep similarity also considers similar but non-identical methods based on their OpCode n-grams.

Although deep similarity is slower to compute, the performance is not an issue when checking a single pair of applications. Our solution solves the problem of searching for applications similar with a given one in the entire collection, or even finding all the pairs of similar items.

A database was designed to store all the collection's data in a manner that allows fast retrieval of similar applications. The search is based on the concepts of reversed index and locality-sensitive hashing and is performed using Map-Reduce algorithms. For shallow similarity, we can find all pairs similar with a given one or all similar pairs. For deep similarity, we can perform an approximate search in order to retrieve even heavily obfuscated clones for a given application.

The designed system was able to handle a large collection of 1,165,942 Android samples, from which 44,675 unique ones were involved in 214,818 similarity cases. A dynamic analysis on the involved applications showed that plagiarism not only affects developers but also the users, by performing actions that affect the user's privacy.

The following contributions were presented in this chapter:

- two similarity functions for Android applications markets:
  - shallow similarity, which is faster to compute;
  - deep similarity, a slower but more robust to obfuscations function;
- a scalable architecture based on map-reduce for identifying plagiarism cases in entire Android markets (millions of applications);
- a description and classification of the attack vectors involved in mobile applications plagiarism;
- a technique for identifying application publishers in order to avoid plagiarism reports for applications developed by the same entity.

This chapter is based on the following published work:

- **Ciprian Oprișa**, Dragoș Gavriluț, and George Cabău. A scalable approach for detecting plagiarized mobile applications. *Knowledge and Information Systems*, pages 1–27, 2015.([OGC15])

Chapter 8

Conclusions and Contributions

8.1 Thesis Conclusions

This thesis proposes new techniques for dealing with the overgrowing threat of malicious software. The large number of new samples that appear each day impose automatic analysis and machine learning techniques.

One important characteristic of the malware ecosystem is that we are dealing with many variations of the same binaries generated in order to avoid anti-virus detection. To handle these variations, we must be able to identify similar samples based on characteristics extracted from their code. Several distance metrics were proposed in order to compute such similarities. They can be categorized according to the underlying model:

- string semantics: a program is abstractly represented as string of OpCodes
- set semantics: a program is represented as a set of OpCode n-grams (which can be considered abstract composite operations)
- hierarchical structures: the binary code can be split into packages, classes and methods

For each model, several distance functions were described, analyzed and tested against real-world programs. For each distance computation, the first step is to parse the file format and disassemble the program code. The experiments were performed on Microsoft Portable Executables, .NET programs and Android applications. Each file format has its own particularities but can be parsed and adapted to work with the proposed distance functions.

The weighted common n-grams distance, based on set semantics outperformed other distance functions in terms of Precision and Recall and even obtained better results in identifying plagiarized student homeworks than a stat of the art plagiarism checker. The deq distance was proved to be a viable alternative to the edit distance because it can be computed in linear time, instead of quadratic time and provides similar results. When extra information like the original code structure (division into classes and methods) is available, a distance that takes into account such information will provide better results.

The OpCode n-grams become more robust to program obfuscations if we carefully select which instructions to take into consideration and which to ignore. Unfortunately, this selection process is not difficult, as the search space for the optimal subset of operations is exponential. This thesis proposes two selection methodologies based on genetic algorithms and Particle Swarm Optimization that are able to *learn* quality selections of instructions subsets (although not guaranteed to be optimal).

Being able to compute the distance between two programs is not enough to analyze large collections. What we need is to split the collection into clusters, each cluster containing similar samples so only the cluster representatives need to be further analyzed.

Classical clustering algorithms involve computing the distance between each pair of samples in the collection. For large collections of millions or even billions samples, these distance computations alone would take too much time for the algorithm to be of any practical value. This thesis proposes two new algorithms for approximating the clusters without requiring pairwise distance computations.

The first proposed clustering algorithm works with the deq distance and is based on the linearity of the Generalized Suffix Tree data structure. Deep nodes in this three will correspond to groups of strings that have long common substrings so they will be joined into the same cluster.

The second clustering algorithm is based on Locality-Sensitive Hashing. Although this clustering model has been proposed before in the literature, the LSH parameters selection have been left to empirical procedures. Our work provides a technique to select the optimal parameters based on data distribution.

The Suffix Tree clustering approach managed to cluster 10^5 samples in less than a minute but it requires too much memory to scale further. The LSH approach does not have this limitation and was able to cluster 10 million samples in a few hours.

Having the clusters built, two practical problems were solved in the following chapter:

- given a cluster, select the most representative samples to be further analyzed
- if some of the sample verdicts (clean or infected) were already determined, can we infer the verdicts for the other similar samples?

The representatives selection problem is NP-Hard and was solved using an approximation algorithm. The experimental results showed that by selecting the centroids to analyze further, the amount of work is reduced almost 20 times.

For the verdicts inference, we have designed a propagation algorithm that threats the cluster as a connected graph and infers the verdicts from the neighboring nodes.

The final part of the thesis presents practical considerations on the scalability issue, by proposing a complete system able to identify plagiarism cases in the Android market. Some practical issues are discussed, starting with the motivation and the techniques employed by the attackers when plagiarizing mobile applications. One of the challenges in identifying plagiarized Android applications was that 90% of the code of a typical app is library code, that is likely to be found in other applications as well. Another challenge is the fact that most Android developers reuse the code between their apps, but we don't want to flag two applications belonging to the same developer as plagiarized. To address this challenges, the proposed system is able to identify the applications publishers and to detect and eliminate from analysis the library code. The proposed architecture is based on the Map-Reduce paradigm, in order to ensure scalability and to deal with the fact that our apps collections is continuously increasing.

The results presented in this thesis show that machine learning and big data make a great addition to the existing security and privacy techniques. The proposed methods and algorithms can be applied in order to reduce the amount of manual work in analyzing security threats and also to deal with the scale of the new emerging threats, for which classical security solutions are obsolete.

8.2 Thesis Contributions

8.2.1 Theoretical contributions

- a literature survey describing recent results in the fields of malware research and machine learning applications (chapters 2 and 3).
- several distance/similarity metrics for comparing binary programs. The proposed metrics are presented, formalized and tested on real-world programs.
  - the *deq distance*, based on OpCodes strings semantics (subsection 4.2.2). This distance is faster to compute then the classic edit distance and obtains similar results. The basic idea is to use the longest common substring concept instead of longest common subsequence.
  - weighted common n-grams, for assigning lower weights to frequent n-grams when comparing n-grams sets (subsection 4.2.3). The assigned weights help building a better similarity function that is less prone to false alarms due to legitimate shared code.
  - a hierarchical similarity algorithm, for dealing with structured code (section 4.3). The algorithm takes into account the hierarchical structure of some programs (like Android applications) and provides more precise and faster results than the ones obtained with raw n-grams.
  - two similarity functions for Android applications markets (section 7.2). The similarity functions will consider the whole applications ecosystem, not only the two compared applications. One of the functions is faster to compute, while the second one is slower but more robust to obfuscation.
- a method for automatically selecting the most relevant OpCodes for constructing abstract program representations (section 4.4). The method employs Genetic Algorithms and Particle Swarm Optimization in order to select a subset of OpCodes, such that the n-grams extracted using only the selected OpCodes provide the best separation between clean and malicious samples.
- an almost-linear clustering algorithm for OpCode strings based on Suffix Trees (section 5.1). A Generalized Suffix Tree can be constructed in linear time and each node at a given depth corresponds to a common substring of the same length. The pairwise samples comparison is avoided by linking only the samples situated in the same nodes, below a threshold depth.
- a technique for selecting the optimal parameters for Locality-Sensitive Hashing in order to improve clustering performance (section 5.2). Locality-Sensitive Hashing also avoids pairwise samples comparison by selecting only pairs of samples with a high probability of being similar. The proposed parameters selection technique allows finding a balance between the number of computed hashes and the number of pairwise distance computations in order to reduce the overall computing time.
- a technique for inferring the verdicts for new samples based on their similarity with known samples (section 6.2). An approximation algorithm for the dominating set problem is used in order to select the most representative samples from a cluster. By using the verdicts computed for a subset of the items in a clusters, the remaining verdicts are automatically inferred using a new propagation algorithm.

8.2.2 Practical contributions

- a plagiarism detection system for students programming assignments (subsection 4.5.1). The system uses a compiler to produce binary code and and extracts OpCode sequences from it. The obtained results are better than state of the art plagiarism detectors.

- a detailed analysis of the .NET executables (subsection 4.1.3). The description includes the file format analysis but focuses on the disassembly process and how to treat various instructions classes.

- two clustering systems based on Suffix Trees and Locality-Sensitive Hashing, able to cluster several million unique samples in a few hours (section 5.3). The systems can perform the cluster analysis faster by relaxing the single linkage condition but the error percentage can be tuned and can be arbitrarily small.

- a system for selecting the most representative samples from a cluster for advanced analysis (section 6.1). By analyzing only the selected representatives, the analysis effort is reduced 20 times and we are still able to determine the verdicts for all samples in a cluster.

- a scalable architecture based on map-reduce for identifying plagiarism cases in entire Android markets (millions of applications) (section 7.3). The proposed system can build a continuously updating database that contains the features extracted from the entire application market. The database can be queried to retrieve items that are similar to a given one or to retrieve all pairs of plagiarized apps.

- quality and performance evaluation for all described algorithms and comparison with state of the art when possible.

The results of this thesis have been disseminated by publishing 4 journal papers (3 as first author), 11 conference papers (9 as first/unique author), 1 short paper and 3 non-academic papers presented at international conferences for the anti-malware industry. One of the journal papers is ISI indexed, with an impact factor of 1.782. One of the conference papers received the Best Paper Award. The publications received 13 independent citations. Some of the technologies developed in this thesis are used in Bitdefender anti-virus, an award winning product that protects more than 500 million users worldwide.

Bibliography

[ABKS99] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod Record*, volume 28, pages 49–60. ACM, 1999.

[Ack28] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[Adl13] Mark Adler. Zlib home site, 2013.

[And14a] Android Developers. Developer registration, 2014.

[And14b] Android Developers. Signing your applications, 2014.

[AT16] AV-Test. Malware statistics, 2016.

[Bäc96] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, 1996.

[BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[Bil07] Daniel Bilar. Opcodes as predictor for malware. *Security Informatics*, 1:156–168, 2007.

[Bis02] Matt Bishop. *Computer security: art and science.* Addison-Wesley, 2002.

[Bra12] Brandon Bray. Announcing the release of the .net framework for windows phone 8. Technical report, Microsoft Corporation, 2012.

[Bro97] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[CBG14] Doina Cosovan, Razvan Benchea, and Dragos Gavrilut. A practical guide for detecting the java script-based malware using hidden markov models and linear classifiers. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 236–243. IEEE, 2014.

[CGC12] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, pages 37–54. Springer, 2012.

[Cho13] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[CLR$^+$01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[Coh89] Fred Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):297–298, 1989.

[Cor13] Microsoft Corporation. Visual c++, 2013.

[Dab13] Gil Dabah. distorm, powerful disassembler for x86/amd64, 2013.

[Def77] Daniel Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Eag11] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler.* No Starch Press, 2011.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[Faw06] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[Fei98] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[Fer07] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[For14] Pau Oliva Fora. Beginners guide to reverse engineering android apps. *RSA Conference*, 2014.

[FOW87] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[FS97] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[GR69] John C Gower and GJS Ross. Minimum spanning trees and single linkage cluster analysis. *Applied statistics*, pages 54–64, 1969.

[Gru15] Ben Gruver. smali/baksmali, 2015.

[Gus97] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[H$^+$96] Nevin Heintze et al. Scalable document fingerprinting. In *1996 USENIX workshop on electronic commerce*, volume 3, 1996.

[HHW$^+$13] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[HL90] Stephen T Hedetniemi and Renu C Laskar. Bibliography on domination in graphs and some basic definitions of domination parameters. *Discrete Mathematics*, 86(1):257–277, 1990.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[Int12] Ecma International, editor. *Common Language Infrastructure (CLI).* Ecma International, 2012.

[Int13] Intel Intel. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1, 2013.

[Jac01] Paul Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura.* Impr. Corbaz, 1901.

[JN09] PK Jana and Azad Naik. An efficient minimum spanning tree based clustering algorithm. In *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pages 1–5. IEEE, 2009.

[Joh73] David S Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49. ACM, 1973.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. Java (tm) language specification. *Addisson-Wesley, June*, 2000.

[Kan92] Viggo Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992.

[KE95] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[KIW07] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1):25–53, 2007.

[Knu98] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[Kol68] Andrei Nikolaevich Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

[KP98] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2-3):271–274, 1998.

[Kuh55] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[L$^+$09] Eric Lafortune et al. Proguard, 2009.

[LD03] 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*, pages 290–299. ACM, 2003.

[Leu08] Gaëtan Leurent. Md4 is not one-way. In *Fast Software Encryption*, pages 412–428. Springer, 2008.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[Llo82] Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[M$^+$67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, number 14 in 1, pages 281–297. California, USA, 1967.

[M$^+$94] Udi Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference*, volume 1. San Fransisco, CA, USA, 1994.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Mur84] Fionn Murtagh. Complexities of hierarchic clustering algorithms: State of the art. *Computational Statistics Quarterly*, 1(2):101–113, 1984.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[NW70] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[OCC13] Ciprian Oprișa, George Cabău, and Adrian Coleșa. From plagiarism to malware detection. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 227–234, Timisoara, Romania, 2013. IEEE.

[OCC14] Ciprian Oprișa, George Cabău, and Adrian Coleșa. Automatic code features extraction using bio-inspired algorithms. *Journal of Computer Virology and Hacking Techniques*, 10(3):165–176, 2014.

[OCN14] Ciprian Oprișa, Marius Checicheș, and Adrian Năndrean. Locality-sensitive hashing optimizations for fast malware clustering. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 97–104, Cluj-Napoca, Romania, 2014. IEEE.

[OCSP14a] Ciprian Oprișa, George Cabău, and Gheorghe Sebestyen Pal. Malware clustering using suffix trees. *Journal of Computer Virology and Hacking Techniques*, pages 1–10, 2014.

[OCSP14b] Ciprian Oprișa, George Cabău, and Gheorghe Sebestyen Pal. A new string distance for computing binary code similarities. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 91–96, Cluj-Napoca, Romania, 2014. IEEE.

[OCSP15a] Ciprian Oprișa, George Cabău, and Gheorghe Sebestyen Pal. Multi-centroid cluster analysis in malware research. In *EVOLVE 2015 International Conference*, Iasi, Romania, 2015.

[OCSP15b] Ciprian Oprișa, George Cabău, and Gheorghe Sebestyen Pal. Semi-automated verdicts assignment for potentially malicious programs. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, Cluj-Napoca, Romania, 2015. IEEE.

[OGC15] Ciprian Oprișa, Dragoș Gavriluț, and George Cabău. A scalable approach for detecting plagiarized mobile applications. *Knowledge and Information Systems*, pages 1–27, 2015.

[OI15] Ciprian Oprișa and Nicolae Ignat. A measure of similarity for binary programs with a hierarchical structure. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, Cluj-Napoca, Romania, 2015. IEEE.

[Opr15] Ciprian Oprișa. A minhash approach for clustering large collections of binary programs. In *Control Systems and Computer Science (CSCS), 2015 20th International Conference on*, pages 157–163, Bucharest, Romania, 2015. IEEE.

[PG12] Pouik and G0rfi3ld. Similarities for fun & profit. *Phrack Magazine*, 2012.

[Pie02] Matt Pietrek. An in-depth look into the win32 portable executable file format. *MSDN Magazine*, 2002.

[Pis08] Daniel Pistelli. The .net file format. `http://www.codeproject.com/Articles/12585/The-NET-File-Format`, 2008.

[PM13] Mykola Protsenko and Tim Muller. Pandora applies non-deterministic obfuscation randomly to android. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 59–67. IEEE, 2013.

[PMRB09] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[PNNRZ12] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing smartphone applications: attack strategies and defense techniques. In *Engineering Secure Software and Systems*, pages 106–120. Springer, 2012.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[RC07] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical report, Technical Report 541, Queen's University at Kingston, 2007.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[Rij79] C.J. Van Rijsbergen. *Information Retrieval (2nd ed.).* Butterworths, 1979.

[RU12] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets.* Cambridge University Press, 2012.

[SC08] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

[Sch16] B. Schelter. Maxima, 2016.

[SDA02] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE, 2002.

[SE98a] Y. Shi and R. Eberhart. A modified particle swarm optimizer. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73, May 1998.

[SE98b] Yuhui Shi and RussellC. Eberhart. Parameter selection in particle swarm optimization. In V.W. Porto, N. Saravanan, D. Waagen, and A.E. Eiben, editors, *Evolutionary Programming VII*, volume 1447 of *Lecture Notes in Computer Science*, pages 591–600. Springer Berlin Heidelberg, 1998.

[SF01] Péter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, 2001.

[SH12] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[Sib73] Robin Sibson. Slink: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[SLH12] Malcolm Slaney, Yury Lifshits, and Junfeng He. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9):2604–2623, 2012.

[SMF$^+$12a] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1–22, 2012.

[SMF$^+$12b] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1, 2012.

[Sok58] Robert R Sokal. A statistical method for evaluating systematic relationships. *Univ Kans Sci Bull*, 38:1409–1438, 1958.

[Sør48] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biol. Skr.*, 5:1–34, 1948.

[STC04] John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis.* Cambridge university press, 2004.

[SW48] Claude Elwood Shannon and Warren Weaver. A mathematical theory of communication, 1948.

[SWA03] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[VCGL15] Cristina Vatamanu, Doina Cosovan, Dragoş Gavriluţ, and Henri Luchian. A comparative study of malware detection techniques using machine learning methods. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 9(5), 2015.

[VGB12] Cristina Vatamanu, Dragoş Gavriluţ, and Răzvan Benchea. A practical approach on clustering malicious pdf documents. *Journal in Computer Virology*, 8(4):151–163, 2012.

[vRNvD09] Johan MM van Rooij, Jesper Nederlof, and Thomas C van Dijk. Inclusion/exclusion meets measure and conquer. In *Algorithms-ESA 2009*, pages 554–565. Springer, 2009.

[Wan06] Annie Wang. Deploying microsoft .net framework version 3.0. Technical report, Microsoft Corporation, 2006.

[War13] Christina Warren. Google play hits 1 million apps, 2013.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[XOX02] Ying Xu, Victor Olman, and Dong Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[ZCM07] Yi-chao ZHANG, Mei CHE, and Jun MA. Analysis of the longest common substring algorithm. *Computer Simulation*, 12:025, 2007.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.