A Feature-Based Call Graph Distance Measure for Program Similarity Analysis

Simo Linkola

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, May 13, 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Simo Linkola

Työn nimi — Arbetets titel — Title

A Feature-Based Call Graph Distance Measure for Program Similarity Analysis

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	May 13, 2016	54

Tiivistelmä — Referat — Abstract

A measurement for how similar (or distant) two computer programs are has a wide range of possible applications. For example, they can be applied to malware analysis or analysis of university students' programming exercises. However, as programs may be arbitrarily structured, capturing the similarity of two non-trivial programs is a complex task. By extracting call graphs (graphs of caller-callee relationships of the program's functions, where nodes denote functions and directed edges denote function calls) from the programs, the similarity measurement can be changed into a graph problem.

Previously, static call graph distance measures have been largely based on graph matching techniques, e.g. graph edit distance or maximum common subgraph, which are known to be costly. We propose a call graph distance measure based on features that preserve some structural information from the call graph without explicitly matching user defined functions together. We define basic properties of the features, several ways to compute the feature values, and give a basic algorithm for generating the features.

We evaluate our features using two small datasets: a dataset of malware variants, and a dataset of university students' programming exercises, focusing especially on the former. For our evaluation we use experiments in information retrieval and clustering. We compare our results for both datasets to a baseline, and additionally for the malware dataset to the results obtained with a graph edit distance approximation.

In our preliminary results we show that even though the feature generation approach is simpler than the graph edit distance approximation, the generated features can perform on a similar level as the graph edit distance approximation. However, experiments on larger datasets are still required to verify the results.

ACM Computing Classification System (CCS):

- **Information systems~Similarity measure**
- Information systems~Clustering and classification
- Security and privacy~Intrusion/anomaly detection and malware mitigation
- **Theory of computation~Program analysis**
- Software and its engineering~Software evolution

Avainsanat — Nyckelord — Keywords
executable analysis, call graph, feature generation, distance measure

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Trying to determine the function of a piece of software from its executable by other means than running the program is a hard task. Even deciding whether the program will come to a stop, or halt, is undecidable. Analysing the behavior of programs is still needed in, for example, malware[1] detection.

The history of large scale malware is approximately 30 years old; although, the Cambrian explosion of malware triggered only after the use of internet spread. The term, computer virus, was introduced by Frederick Cohen in 1984 (see [7]), and the first MS-DOS virus, Brain, was detected in 1986. However, the definition of the computer viruses is not entirely agreed upon; hence, the identity of the first program to carry the name has been exposed to debate in the history of computer security [10].

The first viruses until the mid-80's were largely self-replicating exploits, which did no further harm – excluding the consumption of disc space – to the compromised systems. Afterwards, the evolution of the viruses and other malware has taken a path from damaging boot sector viruses, such as Michelangelo, through e-mail worms and Trojan horses, that bloomed in the late 90's and early 00's, to high profile software – e.g., botnets – that are nearly entirely spread for commercial purposes. In the few recent years, malware tailored to the spesific purposes by the military or national agencies has been exposed.

Malware evolution can be generally seen as a two-step race where both steps are carried simultaneously. In the first step, the malware publishing groups and individuals modify the existing malware, and test them against various cutting edge anti-virus programs until the programs don't recognize the malware anymore. In the second step, anti-virus software groups and companies collect new malware samples and adjust their software to detect them. As the time goes by, also entirely new malware families are created. The new exploits may target previously unknown zero-day vulnerabilities, or existing holes in the security. Thus, the face of the malware detection is ever changing; for example, mobile malware families were said to increase 58% from 2011 to 2012 [34].

The continuously evolving field makes similarity analysis of different malware instances a key point in recognizing the variations of already known malware families, and in detecting the new and unseen exploits. The enhanced detection helps private parties and companies by providing better anti-virus software,

[1] Malware is a general term for various kinds of hostile or intrusive software such as viruses, worms, Trojan horses, ransomware, rootkits, keyloggers, spyware and other malicious programs.

and efficient classification and clustering aids the analysts in anti-virus software companies to perform justified decisions of the relations between different malware families and variants.

As malware publishers can use tools, such as Mistfall, Win32/Smile and RPME [35], to automatise code obfuscation, it is important to handle similarity analysis in a way that is as resistant as possible to such methods. Therefore, the first proposed methods that compute similarities or edit distances between operational code (opcode) sequences are too unstable as even substantial changes in the code may not change the actual functionality of the program.

A plenty of different approaches have been proposed for more robust malware detection and classification. They can be coarcely divided into two groups: dynamic (behavioral) and static analysis. Dynamic analysis of a malware binary is performed by running the malware in a so called sandbox environment and data from the run is collected in some form; typically as a feature vector or as snapshots of the system states before and after the run. In static analysis, malware binary is disassembled with a dissambler, e.g. IDA Pro [9]. Disassembling yields a dissection of the binary's inner structures, such as function caller-callee relationships. The drawback of the dynamic analysis is that the data is only captured from one run and may not represent all of the possible behavioral traits. On the other hand, the static analysis may not be able to interpret all of the code's inner structure, thus yielding only partial or even false information.

The caller-callee relationships acquired by the static analysis can be used to create a call graph of the malware. A call graph of a binary is a directed graph where vertices represent local and non-local functions and edges represent calls between functions.

The call graphs serve as a useful higher abstraction of the program binaries. They are not so prone to simple opcode switching techniques, although they can be relatively easily distorted, e.g. by refactoring a local function into two different functions, where the two functions combined have exactly the same behavior as the non-refactored function.

In most of the works on malware analysis using call graphs, the similarity measure between malware variants is acquired either by using graph edit distance (GED) estimation, maximum common subgraphs or function matching. However, these techniques are still quite vulnerable to code obfuscation techniques that heavily alter the call graph's structure without altering the functionality of the software.

Trying to capitalize on this notion, we choose a structurally oriented feature

approach in this thesis. To compute a distance between call graphs, we do not try to explicitly match local functions between different programs. Instead, we consider as features the sets of k non-local functions. The non-local functions must be linked together by a call structure where there exists a local function root, such that each non-local function is at most dth successor of the root. We generate these feature sets from both call graphs, and the distance between the call graphs is then computed to be the distance between the feature sets.

The used distance measure is not limited to the malware analysis, and is, of course, applicable to programs of various kinds. One such task is to analyse how similar programming exercises are. Measuring the distance between the returned exercises can reveal common higher level structures in the exercise solutions, or in extreme cases may even give a hint of plagiarism[15].

We evaluate our features with two datasets: a malware dataset consisting of 193 variants in 24 different families, and a student dataset consisting of a total number of 366 exercises returned for five different exercise assignments. In our experiments we use retrieval tasks and clustering to analyse how well the proposed features perform. In addition, the proposed features are compared against results obtained with GED approximation for the malware dataset.

The rest of the thesis is organized as follows. First, previous related work in the field is reviewed. Then, basics of graphs and call graphs are explained. After different graph related properties, the graph features and in particular the proposed d-reachable k-gram features are discussed, and an attempt to analyse and define what makes a good call graph feature is made. After the proposed features are introduced, we give an overall view of the two datasets used in this thesis. Then, the call graph features are used to perform clustering and retrieval tasks on the two datasets and the performance of the features is evaluated. Following the experiments is a discussion about the validity of the results, and the thesis ends with conclusions and notions about the possible future work.

2 Related Work

In this section we will give a short overview of the work previously done in related areas. However, before we can delve deeper, we must make a clear cut between two different forms of malware analysis: malware detection, and malware classification. Even though the malware domain has been studied extensively, these terms are used vaguely or with mixed meanings in the existing literature. The most prominent one is malware detection. In this thesis, the malware detection

is used to specify the act of detecting when the existing malware variants infiltrate or compromise a system. The detection is typically run by anti-virus software and uses finger printing and/or heuristics to detect possible threats to the system. Of course, some finger prints or heuristics may also match new malware variants, but the methods do not separate these from the existing ones.

On the other hand, malware classification and clustering are done by anti-virus companies analysts in order to retrieve more information of the new and existing variants and families, and to define their relationships. However, as malware detection is based on the work done by the analysts, the two are closely related and the tools and techniques developed for one can often be applied to the other.

The history of malware classification is as old as the first damaging malwares. Early work based on treating the malware binary as a sequence of *operational codes* (opcode). Malware variants were detected and classified based on various versions of opcode fingerprinting techniques (opcode sequences with various ways to denote wildcards, etc.) [35].

The search for resilient methods to classify self-morphing and obfuscated malware variants into new and existing strains has lead to diverse field of study which can be roughly divided into two sections: behavioral[2] and static analysis. Although, in many occasions static and behavioral techniques are combined into hybrid methods for a more complete approach.

Behavioral analysis runs malware variants in a so called sandbox environment and collects information by monitoring run-time system calls. This information can then be used, e.g. to create a feature vector which is compared against vectors acquired from other variants.

Using call graphs acquired via static analysis to examine similarities in benign or malware binaries is another popular approach. Techniques revolving around call graphs can be generally divided into three closely related, but distinct approaches: finding maximum common subgraphs (MCS), finding graph edit distances (GED) [31], and finding vertex or edge matchings between the call graphs based on local neighborhoods.

Carrera and Erdélyi [6] developed a python package (IDAPython) for easy extraction of call graphs from IDA Pro [9]. Similarity between call graphs is based on function matching where local functions are matched based on their call-tree signature, which is a binary vector expressing called non-local functions, and control flow graph (CFG) signatures; the list of edges connecting basic blocks in function's CFG. The method is evaluated using several small case studies, and

[2]behavioral and dynamic analysis are used interchangeably in this thesis

constructing a phylogenetic tree from the whole dataset.

Briones and Gomez [4] continue the work done by Carrera and Erdélyi [6], and modify their function matching and similarity metric. For each local function, a signature is computed from the sequence of its opcodes and used for fast local function matching. Furthermore, the CFG signatures are altered to contain number of basic blocks, number of edges and number of subcalls.

Park et al. [25] use behavioral call graphs and define a similarity metric based on maximal common subgraphs (MCS). The similarity between two graphs is derived from the number of nodes in the MCS divided by the number of nodes on the larger call graph. They evaluate their method with 300 malware variants (6 families each consisting of 50 samples) and 80 benign executables, and obtain tentatively promising results.

The first large scale work which utilizes GED approximation to call graphs in malware classification was done by Hu et al. [12] . The GED approximation used is a bipartite graph matching proposed in [26, 27], which is further optimized. Bipartite graph matching uses the Hungarian algorithm originally proposed by Kuhn [20]. Fundamentally, Hu et al. construct the whole system to store and query malware in a two-layered hierarchical database to efficiently index the malware and therefore accelerate the queries to database. Both of the layers are trees, where the second layer resides in the first layer's leaf nodes. The first layer of the database is an altered B+ -tree which uses four very common features to index the malware into the optimistic Vantage Point Trees (VPT) in B+ -tree's leafs (the second layer). The search in VPT's is done by approximating the GED as mentioned and selecting k-nearest neighbors to return. In effect, the approach reduces the amount of costly GED approximations needed to produce precise query results from the whole database.

Kinable [16] tries to find good clusterings of a call graph set consisting of 194 malware variants in 24 different families. He uses two different algorithms to find good approximations of GED: bipartite graph matching [26, 27] and genetic search based method. The results of his experiments suggest that genetic algorithm takes too much time in order to reach as good results as bipartite graph matching. Furthermore, as GED needs a cost function for different graph operations, two different cost functions are proposed: random walk probability vectors (RWP), and simple relabeling and local neighbourhood matching. He reaches to a conclusion that RWP's computational complexity does not justify the minimal gains it gives when compared to a simple relabeling and local neighbourhood matching cost. For clustering, he tests k-medoids, G-means and DBSCAN in order to find the

optimal algorithm and cluster amount. Finally, a silhouette coefficient is calculated for all DBSCAN's density reachable parameter's values with several minimum cluster sizes to find the optimal parameter settings to produce the final clustering.

Kostakis et al. [18], and later Kinable and Kostakis [17], continue the work started by Kinable in [16]. Kostakis et al. [18] improve the GED approximation results with adapted simulated annealing algorithm, which uses opcode sequence matching for local functions, and Kinable and Kostakis [17] use the GED approximations obtained in [18] in the same experiment suite as in [16] to verify their suitability for malware clustering.

Xu et al. [37] propose a similarity metric based on matched edges in two call graphs. Local functions are matched by using coloring based on opcode types present in the functions and cosine similarity of opcode type histograms.

Even though a lot of different approaches for malware similarity analysis based on call graphs have been proposed, which are in most cases applicable to other program's also, there is still room for new approaches as the problem has not been thoroughly solved by any single method. In most cases call graph similarity measures are computationally costly, even though the analysis is accelerated by approximation techniques. Also, the computationally costly methods have to be done for each call graph pair separately. The approach proposed in this thesis overcomes these problems as the feature generation is fast and can be done first separately to all call graphs, making the actual distance computation between the call graphs efficient because of the precomputed feature sets.

3 Background

In this section we describe the background related to our work. The information presented here is needed in later sections, especially in Section 4 where proposed features are discussed and in Section 5 where an algorithm for mining the proposed features is introduced. First, basic graph notation and terminology is introduced in Section 3.1. Then, we move on to call graphs in Section 3.2, where we discuss what kind of properties call graphs obtained with static analysis have, and in Section 3.3 we define different similarity measures used with call graphs. The section finishes with a short examination of code obfuscation techniques used by the malware publishers, with an emphasis on techniques affecting call graph structures.

3.1 Graphs

Graphs are popular in representing structured data. Next, we will give basic definitions of unlabeled graphs, subgraphs and graph isomorphism used in this thesis. These definitions are needed later, when call graphs and similarities between graphs are discussed.

Definition 1 (Graph)**.** A graph G consists of a vertex set $V(G)$ and an edge set $E(G)$. Edge set $E(G)$ is a relation between vertices, i.e. for all $(u, v) \in E(G)$ if and only if $u, v \in V(G)$. For *undirected* graphs the relations are symmetrical: $(u, v) \in E(G)$ if and only if $(v, u) \in E(G)$. *Directed* graphs do not have this restriction.

Definition 2 (Subgraph)**.** Graph H is said to be a subgraph of graph G if and only if $|V(H)| \leq |V(G)|$ and there exists an injective function $f : V(H) \to V(G)$, such that for all $(u, v) \in E(H)$ if and only if $(f(u), f(v)) \in E(G)$.

Definition 3 (Graph isomorphism)**.** A graph H is isomorphic with a graph G if and only if $|V(H)| = |V(G)|$ and there exists a bijective function $f : V(H) \to V(G)$, such that $\forall (u, v) \in E(H)$ if and only if $(f(u), f(v)) \in E(G)$, i.e. both graphs are each others subgraphs.

A simple graph can have at most one edge between two vertices, whereas *multigraphs* allow several edges between vertice pairs. Unless otherwise stated, graphs in this thesis are simple graphs; with an addition that a directed graph G with edges (u, v), $(v, u) \in E(G)$, where $u, v \in V(G)$, is considered as a simple graph.

A graph can also be *labeled*. A labeling is a function $L : S \to \mathcal{L}$, where $\mathcal{L}$ is a label set and $S \subseteq V(G)$. A label set can contain any kind of objects; typical labels are positive integers, colours or character strings. Graph is *fully labeled* if $S = V(G)$, and *partially labeled* if $S \subset V(G)$. In many domains vertex labels are unique, which makes it trivial to construct an unique edge labeling for simple graphs by combining the vertice labels; this applies also for directed simple graphs with edges oriented in both directions between vertice pairs. A small, labeled, directed graph can be seen in Figure 1.

Edges of a graph link its vertices together: an edge (u, v) is said to *connect* vertices u and v. *A path* in an undirected graph G is a sequence of edges $(e_1, e_2, \ldots, e_n)$, $\forall e_i \in E(G), i \in \{1, 2, \ldots, n\}$, which connects a sequence of vertices $(v_1, v_2, \ldots, v_n, v_{n+1})$, where $\forall v_i \in V(G)$, and $e_j = (v_j, v_{j+1})$, $j \in \{1, 2, \ldots, n\}$. Similarly as in the case of a single edge, the vertices along the path are said to be *connected*. The length of a path p, $len(p)$, is the number of edges it contains. Furthermore, we say that the (directed) distance between the vertices u and v in a graph G is the length of the

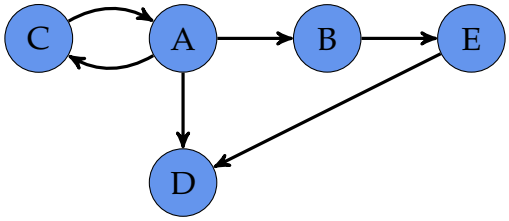


Figure 1: A small directed graph, with five labeled vertices and six unlabeled edges.

shortest path, $p^s(u, v)$, between u and v in the graph G. The shortest path from vertice to itself has zero length, $len(p^s(u, u)) = 0$ for all $u \in V(G)$. If there is no path from u to v, then $len(p^s(u, v)) = \infty$.

3.2 Call Graphs

In this section we introduce call graphs. We start by defining overall characteristics of call graphs and move on to issues and matters more focused to this thesis.

A call graph of a program is a labeled directed graph, which represent the caller-callee relationships of the program's functions. A call graph can be constructed either by using behavioral or static analysis. In the former case, the call graph is obtained at runtime by analysing program's behavior in a sandbox environment, and in the latter, by running a disassembler on the program's binary.

The call graphs obtained by the behavioral analysis tend to obtain only partial information of the caller-callee relationships, since all calling routines are not necessarily observed during a finite number of runs or a limited amount of runtime. On the other hand, they may capture sequential control flow information, which may prove to be essential in defining exact behavioral traits of the program.

Static analysis collects more thorough information of the program binary's structure, but may end up being largely redundant, e.g. in the case where several large common libraries have been included in the binary, but very few of their individual functions have been called. Furthermore, the static analysis tools may be unable to decide which exact function is called from the function set, and therefore can leave some possible call structures unnoticed.

Call graphs can be either *context sensitive* or *context insensitive*. Context sensitive call graphs add an additional vertex for every call configuration (e.g. parameter configuration) that is used to call the function. Context insensitive call graphs add only one vertex for all configurations. Call graps discussed in this thesis are

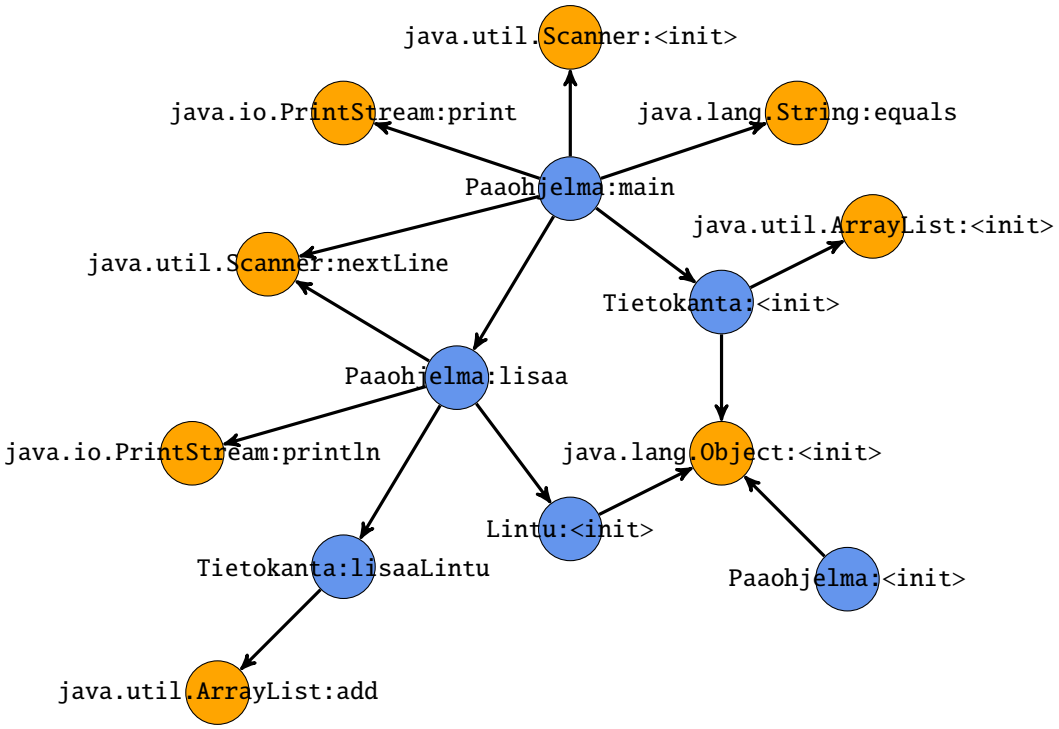


Figure 2: An example of a small call graph extracted from a Java program. The orange nodes with methods starting with 'java' are library (non-local) functions, and the blue nodes are user defined (local) functions.

context insensitive. An example of call a graph structure can be seen in the Figure 2. Typically, extracted call graphs are heavily tree like with a root in a program's entry point, e.g. the `main`-method of a Java program's main class. However, due to multiple reasons (some of which are elaborated in the Section 3.4), they can contain areas not connected by the call structure, or even singular nodes with no incoming or outcoming calls.

The call graph's vertice set can be divided between different kind of functions: local (user defined), common library, external and thunk functions; the latter three called together as non-local functions. In this thesis, only the difference between local and non-local functions is considered. This partitioning is done because, at compile time, the compiler can alter the names of the local functions inside the binary, but the names of the non-local functions remain the same.

Since there is a one-to-one match between program functions and call graph vertices, all the vertices can be uniquely labeled based on the name of the function they represent.[3] Moreover, on similar platforms, matching non-local function

[3]As there is a one-to-one mapping from extracted binary functions to call graph vertices and

names generally point to the same function, making it possible to partially label and match the functions between different call graphs based on non-local function names.

Definition 4 (Call graph)**.** A program's call graph is a labeled directed graph defined by 5-tuple $G = (V(G), E(G), \mathcal{L}, L, T)$, where $V(G)$ is the set of vertices each representing a single function in a program; $E(G)$ is a set of directed edges where $(u, v) \in E(G)$ marks that there is a call from function represented by u to a function represented by v; $\mathcal{L}$ is a set of possible labels, e.g. program's function names; $L : V(G) \to \mathcal{L}$ is a labeling function that assigns an unique label to each vertex; $T : V(G) \to \{L, N\}$ is a function that assigns a type for each vertex, L means that corresponding function in the program is local and N that it is non-local. Moreover, we will denote the set of all local functions in a call graph G by $V_L(G) = \{\, v \mid v \in V(G) : T(v) = L \,\}$, and similarly the set of all non-local functions by $V_N(G) = \{\, v \mid v \in V(G) : T(v) = N \,\}$.

It is important to notice that in our definition a single call graph is *fully* labeled, but when we are dealing with a set of call graphs we treat them as *partially* labeled. Each call graph G in a call graph set is partially labeled so that the set of non-local functions, $V_N(G)$, is labeled, and the set of local functions, $V_L(G)$, is not. The need for this change arises from the fact that the local function names can be heavily altered in the compiling phase, and should not be trusted to have any real meaning. The names of the non-local functions on the other hand are not usually altered at the compiling phase, and matching non-local function names can be thought to point into the same function as mentioned earlier.

Call graphs obtained via static analysis can contain also other relevant information from the binary code. In general, it is possible to extract operational code sequences for each local function in the binary. These sequences can be used to create control flow graphs (CFG) of the local functions. Control flow graph of a function represents all the possible paths the function's basic block structure can be traversed and the jump points to other basic blocks inside the function or to the entry points of other functions. However, we restrict ourselves to only consider caller-callee relationships, and no additional information of the local functions' inner structure is used.

from function calls to call graph edges, we will make no distinction between nodes and underlying functions and may refer to nodes as functions and directed edges as calls.

3.3 Graph Similarity Measures

Graph similarity measures try to define, or approximate, how similar two graphs G and H are. Various graph similarity measures have been proposed for different purposes. The main approaches that have been used with call graphs are: approximate graph edit distance (GED), maximum common subgraphs (MCS), and matching functions based on local neighborhoods and control flow graphs.

Graph edit distance [31] is a derivation of the Levenshtein (or edit) distance for sequences [21]. It is obtained by computing how many alterations have to be done to graph G for it to become isomorphic with graph H. Next, we will give more formal definition of graph edit distance.

Definition 5 (Graph edit distance)**.** Graph edit distance between labeled graphs G and H is a minimum cost elementary operation sequence which transforms graph G into graph H, where each of the elementary operations is assigned a certain cost. For labeled graphs, elementary operations included are vertex and edge relabeling, deletion and addition. In a basic situation, all operations have a fixed cost of 1.

As determining graph isomorphisms is neither known to be solvable in polynomial time nor NP-complete [14], GED is usually approximated via some method. Kinable [16] uses simple relabeling cost which is improved with simulated annealing in [17].

Maximum common subgraph techniques try to find maximal subgraph g that is contained in both graphs. Finding a subgraph that is isomorphic with graph g from a graph G is known to be NP-complete problem [8]. Therefore maximum common subgraph techniques tend to become slow as the graph sizes increase.

Definition 6 (Maximum common subgraph)**.** Maximum common subgraph between graphs G and H, is the subset of vertices in G, $g \subseteq V(G)$, for which there is a bijection to a subset of vertices in H, $h \subseteq V(H)$, g is isomorphic to h, and $|g|$ is maximal.

Function matching concentrates on optimizing a mapping of nodes from graph G to graph H. In call graphs they usually use local functions' control flow graph information to enhance the mapping algorithm (see, e.g. Carrera and Erdélyi [6]). However, each implementation uses their own heuristics; making this approach more diverse than the other two.

3.4 Code Obfuscation

Code obfuscation techniques are used by malware authors in order to hinder the detection of their malware by antivirus software, and once captured, the analysis of the malware's binary code. Obfuscation can be made beforehand, e.g. with packers [29], or it can happen during the malware's life in the wild by self-mutation [5]. Next, we will take a short look on some of the obfuscation techniques that can alter the structure of statically analysed call graphs. Interested readers can see e.g. [35, 33, 29] for more thorough information.

For analysts dealing with binaries, the first task is to capture the program's binary code in order to analyse the code itself. For benign programs this is easily done as static analysis techniques, like disassembly, can extract the program binary's code by simply reading it from the binary. However, binary code extraction becomes harder for programs that can create and overwrite their code at runtime, e.g. by packing the binary code to compressed or encrypted code, which is unpacked at runtime. Packed code can be reverse engineered to resemble the original executable by unpacking, but quality of the result is depended on the exact tools used to pack – and unpack – the binary. On the other hand, malware variants that overwrite their own code cause problems for static analysis as there is no single point in time when all of the program's code would be present in the binary [29].

Disassembly resisting malware complicates capture of the code structure in static analysis. Malware variants can have significant proportions of their codebase consisting of hand-written assembly code with irregular structure. They can hide code, corrupt the code analysis with non-code bytes, or even try to find errors in disassemblers by stress testing them with large quantities of exceptional inputs. Effectiveness of these techniques rest largely at the hands of disassembly tools used.

Instruction obfuscation techniques conceal control-flow information of the program. They can, for example, simulate function calls and returns with alternative instruction sequences, use call and return instructions in non-standard way, or use indirect calls.

Self-mutating malware variants are a special case of insctruction obfuscation techniques that alter their own instructions. They can, e.g. substitute instruction sets with semantically equivalent sets, permutate mutually independent instructions, insert dead-code[4] into the program, change variables in semantically

[4]code that does not do anything

irrelevant way, or alter control-flow information. However, as mutations occur directly in the machine code – and therefore in the code of malware variants themselves – they usually are rather simple. Observations have shown, that malware variants that have undergone large patches of self-mutation tend to consists of highly redundant and useless code [5]. Furthermore, call graphs are quite resistant to the code mutations, as they tend to take place locally in small patches, which do not alter the binary's call structure.

4 Call Graph Features

Feature generation tries to find descriptive bits of information from a set of complex data points. The generated features should aid to simplify the data and represent the data well enough for meaningful analysis. The feature set can then be used, e.g. to analyse similarities between the data points or to index data points for fast retrieval from a database. The feature generation can be coupled with *feature extraction* and *feature selection*; the latter being a special case of the former. Feature exraction methods are used to reduce the redundancy and/or dimensionality of the feature set by transforming the data into a lower dimension. The feature selection is used to select a representative subset of current features as a new feature set.

When the data is represented as graphs, the most typically features generated are frequent subgraphs. They are mined by one of the many frequent subgraph mining algorithms (FSM); such as gSpan [38], MoFa [3], FFSM [13] or Gaston [24]. For a given graph set $\mathcal{G}$ and a threshold t, FSM algorithms extract all subgraphs g, which are present in at least t graphs $G \in \mathcal{G}$. However, most of the FSM algorithms are constructed for undirected graphs and can not handle partially labeled directed graphs, i.e. the algorithms are not build to address special properties of call graphs and their underlying generative process.

In malware analysis, the feature generation approach is more frequently applied in dynamic analysis than in static analysis. In a typical case, the interesting features are defined beforehand, and their appearance is monitored during runtime. This kind of feature generation needs expert knowledge to define the interesting features, and does not (in its simplest form) recognize other interesting behavioral patterns.

In the remainder of this section we introduce the call graph features, d-reachable k-grams, proposed in this thesis. We begin by explaining the motivation behind the features based on some notions on how the previously used call graph similarity measures behave when underlying code is altered. Then, we present the features,

give an example how they are generated and describe their basic properties. The section ends by defining three types of values the generated features can have and how the distance between two feature sets is computed.

4.1 Motivation

The idea behind d-reachable k-grams is simple: the feature set extracted from a piece of software should exhibit similarity even though the underlying code has undergone refactoring phases which do not alter the main functionality of the code, and dissimmilarity when the functionality of the code has changed. To this extent the call graph representation is already quite robust as it is invariant of many basic refactoring techniques. However, typical call graph similarity measures, MCS and GED, are still quite vulnerable to very basic code morphing processes that alter the call graph structure.

Next, we will take a look at two of the morphing processes, which we will name as *local distortions* and *non-local distortions*. Local distortions do not change the code's functionality but alter the call graph's local topology. On the contrary, non-local distortions can move large patches of code to be called from another place in the program (therefore altering the call graph) and can have a substantial effect on the program's functionality.

4.1.1 Local Distortions

Local distortions are simple code refactoring processes that do not change the code's functionality, but alter the resulting call graph structure. Typically, they are operations that split user defined (local) functions into smaller or redundant pieces, or merge several local functions into one. The resulting call graph structure can deceive similarity measures that depend on graph edit distance or maximum common subgraphs. Example 1 introduces some local distortions and their effect on call graph structure.

Example 1 (Local distortions)**.** Consider naive Python code snippet and resulting call graph seen in Figure 3. In this example, A and B are non-local functions which can represent any imported library functions, and functions starting with L are local, i.e. they are defined in the source files.

A simple local refactoring of the code, where L1 from Figure 3 has been divided into two local functions L1 ja L2, is seen in Figure 4. Refactoring alters the call graph structure, affecting the MCS or GED between the two call graphs. Repeated

```
import A, B
def L1():
    A()
    B()
```

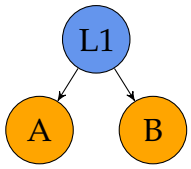


Figure 3: Original Python pseudocode and the resulting call graph

```
import A, B
def L1():
    L2()
    B()

def L2():
    A()
```

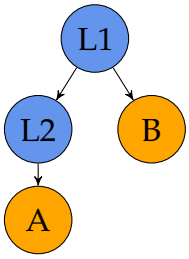


Figure 4: Refactoring the original local function into two local functions.

application of similar refactoring patterns can distort the call graph heavily without having any effect on code's functionality.

Graph edit distance is even more prone to the addition of dummy user functions with similar behaviour. This can be easily demonstrated by duplicating the original local function and choosing by random which of the equivalent functions to call, as seen in Figure 5. By replicating similar modifying process, the distance between original graph and generated graphs grows without a limit.[5]

4.1.2 Non-Local Distortions

On the contrary to the local distortions, non-local distortions can have a significant effect on the functionality, but can remain undetected if similarity is measured by MCS or GED. An example of a simplified non-local distortion is given in Figure 6.

Non-local distortions can include, e.g. a subprocedure involving several local functions that is first removed from the call structure and later applied into some other part of the call structure, with a distinctively different purpose. If the subprocedure itself has a single entrypoint, the GED would observe the change to be only one edge removal and one edge insertion. MCS might stay indifferent of the graph morphing process if the current MCS between the call graphs would not contain the subprocess, but also be affected if it was contained.

[5]It should be noted, that disassemblers try to capture this kind of behavior during static analysis of the binary.

```
import A, B
from random import random as R
def L1():
    if R() > 0.5:
        L2()
    else:
        L3()

def L2():
    A()
    B()

def L3():
    A()
    B()
```

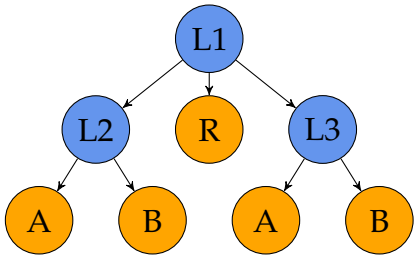
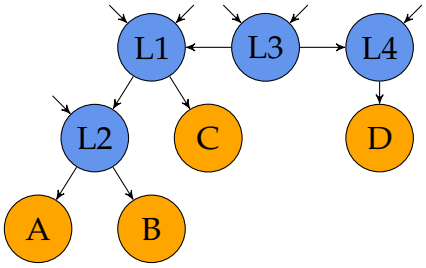
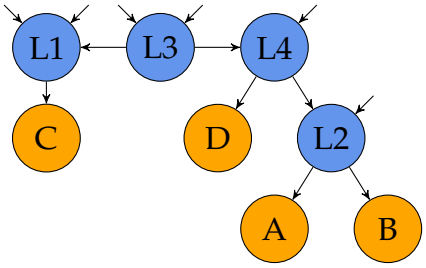


Figure 5: Duplicating original local function and adding a dummy function to choose which one to call.



(a) Original topology



(b) Topology after code transfer.

Figure 6: Simplified non-local distortion

It is important to notice, that similar graph morphing process can happen also in the case where only refactoring of the code is done without actual functionality changes. Thus, the similarity measure should (1) be able to either guess if the actual functionality has changed, or (2) be robust enough in the sense that in both situations some of the correct information is sustained. As the first option appears to be a complex (and interesting) problem by itself, and solving it reliably would probably require accessing the control flow graph information of the local functions, we choose to use the latter approach as is seen in the next section which introduces the chosen call graph features.

4.2 Definition

In this section we define the proposed features, d-reachable k-grams. We will look at the properties of the proposed features more closely in the next section.

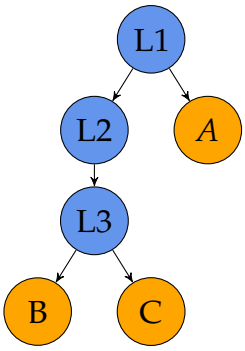
In contrast to current methods that rely on graph matching techniques, we propose a graph distance approximation based on features that are generated from *extended local neighborhoods*, and are *structurally flavoured*. The considered extended local neighborhoods do not only cover direct child nodes, but also further descendants limited by a cutoff parameter. Structurally flavoured means, that some structural properties are taken into account when similarity between generated features are calculated, but small changes in a call graph's call structure are not considered to change feature's identity into another. For now, we will only consider the call graph structures that are accepted as features. We will talk more about how exact value for each feature is computed in Section 4.4.

In essence, the proposed features are non-local function k-grams that are present in a call graph G. As our call graphs have exactly one node for each function, and non-local functions between call graphs can be directly matched, we can define our features as ordered tuples of call graph node (function) labels (function names). Furthermore, because we want our features to be locally oriented and sustain some structural information, each k-gram has to have a local function root r, so that each non-local function in the k-gram is reachable from r. To limit locality, we define a cutoff parameter d, so that r is connected to each function in k-gram with a shortest path at most length d, i.e. each function in k-gram is at most dth successor of r. An example of how features are accepted and rejected is given in Figure 7. Next, in Definition 7 we will give more formal definition for our features, which we name d-reachable k-grams based on the cutoff parameter d and the cardinality of the feature set k, respectively.

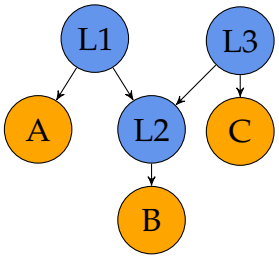
Definition 7 (d-reachable k-gram feature)**.** A d-reachable k-gram feature f in a call graph G is an ordered non-local function set of cardinality k, for which there exists a local function root r, such that r is connected to each function in f with a shortest path of length at most d. That is, $f \subseteq V_N(G)$ and $\left|f\right| = k$, and there exists $r \in V_L(G)$, such that $\forall n \in f : len(p^s(r, n)) \leq d$.

As an exception, if $d = 0$ we don't require any local function root r, but instead consider only non-local function sets that are directly connected with each other[6].

[6]Remember that the set of non-local functions may also contain e.g. thunk functions that are used to assist calls to other functions. This leads to the fact that non-local functions can also call other non-local functions in the call graphs.



(a) Acceptable feature



(b) Non-acceptable feature

Figure 7: An example of how local call structure restricts feature generation. In both figures we have $V_L(G) = \{\text{L1}, \text{L2}, \text{L3}\}$ and $V_N(G) = \{\text{A}, \text{B}, \text{C}\}$. Lets consider the case where $k = 3$, and $d = 3$. We can see that in Figure 7a, L1 serves as a local function root which is connected to all functions in $V_N(G)$. In Figure 7b there is no such local function.

In essence, if $k = 1$ and $d = 0$ the feature set generated is exactly the set of non-local functions, and the feature set generated with $k = 2$ and $d = 0$ would contain non-local function pairs where one is directly called by the other.

As features are mined with a bottom up procedure (see Section 5), where the k parameter defined is the maximum cardinality of non-local function set, we differentiate single features from the whole feature set by naming our feature sets as (k, d)-grams. A (k, d)-gram of a call graph consists of all d-reachable j-grams, where $j \in \{1, 2, \ldots, k-1, k\}$. Example 2 illustrates how (k, d)-gram feature set is generated from a simple call graph.

Example 2 (Feature generation)**.** As an illustrative example of the feature generation, let us consider a call graph extracted from a small Java program that is shown Figure 8. We will generate the features from the call graph up to $k = 5$ starting from $k = 1$ and $d = 0$.

The call graph, G, in Fig. 8 has four local functions, $V_L(G) = \{$`Paaohjelma:<init>`, `Paaohjelma:main`, `Kayttoliittyma:<init>`, `Kayttoliittyma:kaynnista`$\}$ and five non-local functions, $V_N(G) = \{$`java.lang.Object:<init>`, `java.util.Scanner:<init>`, `java.util.Scanner:nextLine`, `java.lang.String:equals`, `java.io.PrintStream:println`$\}$.

Now, if we generate features with $k = 1$ and $d = 0$, it is clear that the generated function set is the same as $V_N(G)$. When increasing k and d the generated function set will grow. With $k = 2$ and $d = 1$ the only local function than can serve as a possible root for the new features is `Kayttoliittyma:kaynnista`, because no other local function calls more than one non-local function. Making all the possible non-local

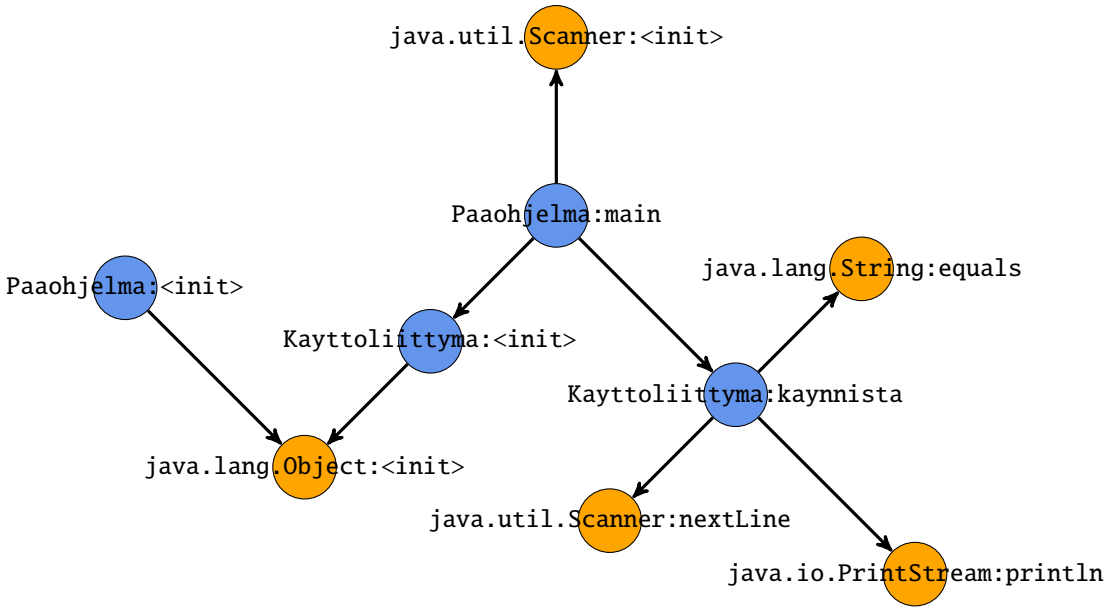


Figure 8: An example of a small call graph extracted from a Java program. The orange nodes with methods starting with 'java' are library (non-local) functions, and the blue nodes are user defined (local) functions.

function combinations from the functions called by `Kayttoliittyma:kaynnista` gives us $\binom{3}{2} = 3$ additional features:

- (`java.io.PrintStream:println`, `java.lang.String:equals`),
- (`java.io.PrintStream:println`, `java.util.Scanner:nextLine`) and
- (`java.lang.String:equals`, `java.util.Scanner:nextLine`).

If we increase $d = 2$, also `Paaohjelma:main` can serve as a root, and from there all the possible non-local function pairs are reachable when $d = 2$. Combining the rest of the possible non-local function pairs generates $\binom{5}{2} - 3 = 7$ features more:

- (`java.io.PrintStream:println`, `java.lang.Object:<init>`),
- (`java.io.PrintStream:println`, `java.util.Scanner:<init>`),
- (`java.lang.Object:<init>`, `java.lang.String:equals`),
- (`java.lang.Object:<init>`, `java.util.Scanner:<init>`),
- (`java.lang.Object:<init>`, `java.util.Scanner:nextLine`),
- (`java.lang.String:equals`, `java.util.Scanner:<init>`) and
- (`java.util.Scanner:<init>`, `java.util.Scanner:nextLine`).

With $k = 3$ and $d = 1$ we would only generate a single feature more: (`java.io.PrintStream:println`, `java.lang.String:equals`, `java.util.Scanner:nextLine`).

Then, increasing d to $d = 2$ would generate the rest $\binom{5}{3} - 1 = 9$ of the non-local function combinations of cardinality three as all the non-local functions are reachable from `Paaohjelma:main` when $d = 2$, similarly as it was in the case when $k = 2$ and $d = 2$.

If we increase $k = 4$, we do not generate any features with $d = 1$ as the maximum number of non-local functions any local functions calls is three. Increasing $d = 2$, we will then generate all the possible features with four non-local functions, giving us $\binom{5}{4} = 5$ features more. Lastly, if $k = 5$ we do not generate any features when $d = 1$, and when $d = 2$ we generate only a single feature which contains exactly the non-local function set $V_N(G)$.

4.3 Properties

We will now take a look at some of the elementary properties (and limitations) of our features, and how changes in the call graph structure affect our features. As the main focus of our features is to be invariant of small refactoring processes – and naive obfuscation techniques – that affect the call graph structure, we are mainly interested in the two refactoring processes mentined earlier: local distortions and non-local distortions.

Local distortions Local distortions affect the call graph's topology locally, e.g by inserting more local functions, deleting them, or applying some (typically gradual) control structure changes. Our features are to some extent resilient to adding or deleting local functions, as the cutoff parameter d gives us some flexibility on how features are generated from the root function. Also, as the root function is not fixed between the call graphs, a lot of local topology changes make us simply choose another root for the feature generation, in effect keeping the generated feature set the same.

However, the feature generation is affected if local distortions are applied consecutively. For example, suppose we apply similar distortion as in the Example 1 from Figure 3 to Figure 4 many times in a row so that each local function added is inserted to be called from the local function inserted in the previous distortion. At some point inserting new local function collides with our cutoff d and feature (A,B) cannot be generated from the root L1 anymore.

Non-local distortions Non-local distortions appear in a call graph what a portion of the code is transferred to another location in the call graph. For example, let us

look at the Figure 6 where the subtree with L2 as a root is transferred to be called from L4 instead of L1. If we have defined that the cutoff parameter $d = 3$, then any feature set generated from the subgraph shown in the image is not affected, as L3 serves as a root from which all non-local functions can be reached before and after the distortion. Same goes for $d = 1$ as local functions' calls to non-local functions is not changed. However, if $d = 2$ we can see that in Figure 6a we can generate second order features (A,C) and (B,C), but after the code transfer in Figure 6b we generate features (A,D) and (B,D) instead.

4.4 Call Graph Distance Using Features

We now have discussed which kind of features we are interested in, and their properties. However, we are yet to cover how we can compute a distance between two call graphs based on the generated feature sets and feature values. Next, we will define the distance between call graphs. The computed distance depends on the feature values, which we will discuss after we have defined the distance measure. For now, it is sufficient to know that we denote the value of feature f in a call graph G with $v_G(f)$. Howerer, if the call graph is clear from the context, we may also use a shorthand notation $v(f)$.

Definition 8 (Graph distance with d-reachable k-grams)**.** Distance between two call graphs G and H, $\text{dist}(G, H)$, is the summed distance between their feature sets, $F(G)$ and $F(H)$, respectively. More precisely,

$$\text{dist}(G, H) = \sum_{f \in F(G) \cup F(H)} \left| v_G(f) - v_H(f) \right| w(f), \tag{1}$$

where $v_G(f)$ is the value of feature f in call graph G, $0 \leq v_G(f) \leq 1$, and $w(f)$ is the weight for the feature f, $0 \leq w(f) \leq 1$, for all f. In a basic setting the weight of the features is constant, $\forall f : w(f) = 1$.

As the distance between call graphs is the distance between their feature sets, we need define the value $v_G(f)$ for each feature f in each call graph G. Even though the proposed features themselves try to be resistant to small changes in the call graph topology, we can include some information about the local topology into the values of the extracted features aiming for more precise representation of the call graph. In theory, more precise representation of a single call graph should result in a more precise computation of the distance between two call graphs.

Before we give the exact formulas to compute different value types, let us consider some traits that the feature values should exhibit. Let feature f be a

d-reachable k-gram for some $k > 1$ and $d > 0$, and $v_G(f)$, $v_H(f)$ and $v_X(f)$ be the values of feature f in call graphs G, H and X, respectively.

First, consider the case where feature is found from call graphs G and H, but not from X. To compute the difference for the feature f between G (or H) and X we need to give f a value in X, as only computing the difference between intersecting feature sets for two call graphs is hardly a meaningful distance measure. Furthermore, the value for $v_X(f)$ must be selected in such a way that if the feature f has been found from G with a smaller d than from H, then the difference between $v_X(f)$ and $v_H(f)$ is smaller than the difference between $v_X(f)$ and $v_G(f)$. The reason for this originates from a following observation: if we do not find f from X up to certain d, we have no proof that f could not be found from X with $d + 1$.

For the above mentioned reasons, we use a dummy feature value for each feature not found from a call graph G, $\forall f \notin V_N(G) : v_G(f) = 0$. Also, we give the features higher values when they are found closer to their root function by computing maximum value m for each k based on the feature value type (the m can vary between different values of k) and d, and then use this maximum value to "flip" feature values so that the features found further away from the root (that is, non-local functions are generally further away from the connecting local function root) have values closer to zero than features found closer to the root (e.g. each non-local function is directly called by the root).

Second trait to consider is that the features of higher order (features generated with larger k) should not have larger maximum values than the features of lower order. The reason for this is simple, the higher order features would start to increasingly dominate the distance measure (remember that there is already a high change that the higher order feature set is exponentially larger compared to lower order feature set), because the difference of a single feature value could be higher between call graphs. To overcome this aspect, we use the maximum value m for each k to normalize the feature values, giving us feature values that are constrained between zero and one for all call graphs and all the features. Furthermore, if the feature is generated with the exception case $d = 0$, its value in all the feature value types is one.

Next, we will describe three types of values the generated features can be assigned to: binary, maximum distance, and sum of distances.

Binary The simplest feature value type is binary. Binary feature value represents the existence of feature $f \in V_N(G)$. It is zero if the feature is not found from graph

G with the given d, and one if it is found. That is, the value of feature f, $v_G(f)$, is

$$v_G(f) = \begin{cases} 1 & \text{if } f \subseteq V_N(G) \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Maximum distance With maximum distance as the value of a feature f in a call graph G is based to be the maximum distance between the root, r, and any non-local function in a feature. Formally, the value of feature f, $v_G(f)$, is computed as follows:

$$v_G(f) = \frac{m(\left|f\right|, d) - \max\{\, len(p^s(r, n)) \mid n \in f\}}{d}, \tag{3}$$

where $p^s(\cdot)$ is the shortest path between two functions, d is the maximum reachability used to generate the features, and $\forall f : m(\left|f\right|, d) = d + 1$.

Sum of distances The most complex feature value is based on the sum of distances between root and all non-local functions in the feature. Given a feature f, we compute the value of feature, $v_G(f)$, as follows:

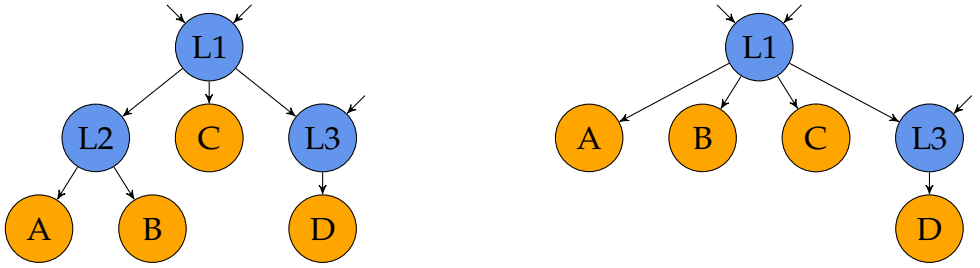
$$v_G(f) = \frac{m(\left|f\right|, d) - \left(\sum_{n \in f} len(p^s(r, n))\right)}{m(\left|f\right|, d) - 1}, \tag{4}$$

where $p^s(\cdot)$ is the shortest path between two functions, d is the maximum reachability used to generate the features, and $\forall f : m(\left|f\right|, d) = \left|f\right| \times d + 1$.

5 Algorithm

In this section we present an example algorithm for generating (k, d)-grams from a call graph, and optional pre- and postprocessing steps. Our preprocessing step tries to minimize local distortions in each call graph, and postprocessing step is used after the features from the whole dataset have been generated in order weight the features and optionally prune some of the features. The main algorithm is implemented in a straight forward manner, and is described here only as an illustrative example of the implementation, not as a heavily optimized version of the generation process.

Next, in Section 5.1 we will first look at the preprocessing step which aims to tighten the call graph layout. Then, we move to main algorithm implementation in Section 5.2. Last, in Section 5.3 we describe some postprocessing procedures, which aim to reduce the redundancy in the collected feature set and enhance overall effectiveness of the collected features.



(a) Original local topology

(b) Local topology after merging.

Figure 9: Tightening the call graph topology by merging single parent local functions with their parent. The local function L2 from Figure 9a is merged into its parent L1 in Figure 9b.

5.1 Preprocessing

Although the d-reachable k-grams try to diminish the effect of the local distortions themselves, we experiment with an optional preprocessing step in order to tighten the call graph's topology before the features are extracted. In this step, we prune away local functions that have only a single parent by combining them with their parent. An example of this procedure can be seen in Figure 9. In source code, this process has a similar meaning as defining the function to be merged as inline function.

5.2 Main Algorithm

The main algorithm for generating (k, d)-grams is simple, and can be seen to have some similarities to the frequent subgraph mining algorithms. Perhaps the most prominent difference is that the algorithm is run for each call graph separately. This can be done because we are interested in *exhaustively* mining all the function sets within maximum distance d. Thus, we cannot save computation time by first mining the $(k-1)$ function sets from *all* the call graphs in the dataset, computing their support, and then pruning from the feature set the features with lower support than the defined minimum support. In fact, we save computation time as only the $(k-1)$-function sets found in a single call graph are used to generate k-function sets. This approach allows addition of new call graphs iteratively, as the minimum support of the features does not need to be adjusted.

The main body of the algorithm is described in Algorithm 1. The algorithm takes as an input a call graph G and feature generation parameters k and d, and

Algorithm 1: Main algorithm for generating d-reachable k-grams

Input: G: a call graph
Input: k: maximum number of non-local functions in a feature
Input: d: maximum distance from root local function to non-local function
Output: generated features

```
1  begin
2      features ← generateZeros(G, k)
3      if k > 1 then
4          distances ← shortestPaths(G, d)
5          collect pairs of non-local functions that have same key in distances to
           pairs and note their maximal feature value
6          features ← features ∪ pairs
7          lastFeats ← pairs
8          while i ← 3 to k do
9              newFeats ← generate(distances, lastFeats)
10             features ← features ∪ newFeats
11             lastFeats ← newFeats
12             i ← i + 1
13     collect the maximal values for each feature from features and return them
```

returns the maximal value for each generated feature depending on the feature value choice (see Section 4.4 for possible feature values). The algorithm starts on line 2 by generating features for the exception $d = 0$, where the features do not have any local function root, i.e. it collects non-local feature sets where one of the non-local functions is the root and all other functions are its descendants. In line 4 it proceeds to compute (directed) distances between all local functions and non-local functions up to a cutoff d with a slightly modified all pairs shortest path algorithm, which is introduced in Algorithm 2. The shortest path algorithm outputs a mapping from local functions to lists of pairs, where each pair contains a non-local function and the distance from the local function to the non-local function.

After generating the shortest paths, in line 5, the algorithm gathers all non-local function pairs that are at most dth successor of some local function r to pairs, and in lines 8–13, the algorithm iteratively increments the cardinality of the features. For each cardinality i it uses the features generated for cardinality $i - 1$ as the starting poin, i.e. in the first iteration pairs generated in line 5 are used to generate triplets,

and so on. The main procedure to generate the features, `generate`, is called in line 9, and is described in Algorithm 3. The algorithm ends by collecting the maximal value for each feature f from the generated feature candidates.[7]

Algorithm 2 describes how shortest paths from all local functions to non-local functions are computed with a cutoff d. The algorithm runs iteratively a slightly modified single source shortest path with a cutoff for each local function in a call graph G as the source . For each source $l \in V_L(G)$, it sets the target to be the set of non-local functions $V_N(G)$. It outputs a dictionary where local functions are keys and values are lists of 2-tuples. Each 2-tuple contains a name of a non-local function and the length of the shortest path to that function from the local function used as the key. The algorithm behaves exactly same as normal single source shortest path with a cutoff for each source, but instead of calculating paths to all other nodes, we set a variable notFound to be $V_N(G)$ at the start of the loop for each local function root and terminate the computation also when notFound $= \emptyset$. In order to save computation time in the following generation phase, we do not construct full distance matrix (that is, set distances to other nodes to infinity), because then we should also process the nodes we already know we do not need to because of they are further away from the root than the cutoff factor d.

In Algorithm 3, the previously generated feature set for cardinality i is used as a starting point to generate the feature set for cardinality $i + 1$. The routine takes each feature from the previous feature set into consideration and looks from which local function it has been generated as root. Then, it looks from distances all the non-local functions that are d-reachable from root, and combines a new feature if the function is not already in the currently considered feature. Lastly, it computes the feature's value using the defined feature value option, and appends the feature to new feature set, which is returned at the end.

5.3 Post-processing

The post-processing step takes place after features from all call graphs have been collected. We experiment with two techniques: pruning and weighting of features.

Pruning Pruning attemps to reduce the noise in the distances computed from the feature sets by removing features that are only present in a single call graph in the dataset. These features are removed because they only add distance between

[7] In order to save computation time, this step can be done while generating the features in Algorithm 1 on lines 5 and 9.

Algorithm 2: shortestPaths Modified all pairs shortest paths algorithm that computes paths only from local functions to non-local functions with predetermined cutoff d. Called from Algorithm 1 on line 4.

Input: G : a call graph
Input: d : cutoff, only distances at max d are considered
Output: distances: a map from local functions (keys) to lists (values), where each list contains (non-local function, distance)-tuples for all non-local functions that are at most d-reachable from the local function used as the key.

```
1  begin
2      initialise distances to an empty map
3      for root ∈ V_L(G) do
4          notFound ← V_N(G)
5          level ← 1
6          paths ← {root: (root,)}
7          nextLevel ← {root }
8          while nextLevel ≠ ∅ and notFound ≠ ∅ and level ≤ d do
9              thisLevel ← nextLevel
10             nextLevel ← ∅
11             for func ∈ thisLevel do
12                 for child ∈ successors(func) do
13                     if child ∉ paths then
14                         paths(child) ← paths(func) + child
15                         add child to nextLevel
16                         if child ∈ notFound then
17                             delete child from notFound
18             level ← level + 1
19         remove paths that do not end in a non-local function from paths
20         compute the length of each path to non-local function in paths
21         store (non-local function, length)-pairs as a list to distances with key
           root
```

the graph they are in and all the other graphs. With a lot of features only present in a single graph, the effective distances may be strongly altered.

Algorithm 3: generate Routine to compute features of cardinality i from features of cardinality $i-1$. Called from Algorithm 1 on line 9.

Input: distances: distances from local functions to non-local functions
Input: lastFeats: (i, d)-gram features
Output: newFeats: $(i+1, d)$-gram features

```
1 begin
2     newFeats ← ∅
3     for feat ∈ lastFeats do
4         root ← getRoot(feat)
5         for func ∈ distances(root) do
6             if func ∉ feat then
7                 newFeat ← func ∪ feat
8                 assign a value to newFeat based on the chosen value type and
                  distances
9                 add newFeat to newFeats
```

Feature weighting Feature weighting aims to solve the effect of the combinatorial explosion of the number of generated features when k increases. With each increase in k the feature set gets proportionally larger, as the amount of non-local function combinations increases. This raises implicitly some questions, e.g. "Which features should we be interested in?" and "Are features of order three more interesting than features of order two?". Our answer to these questions is to limit the effect the feature set of each order has on the distance measure. We do this by weighting each feature in the generated feature set by the number of features generated for that order from all call graphs in the dataset. For example, if the number of features for the order 2 is B, then the weight for each feature f of order 2 is set to $w(f) = 1/B$.

6 Datasets

In our experiments we use two distinct datasets: a programming exercise dataset gathered from university students at the Department of Computer Science of the University of Helsinki, and a malware dataset obtained from F-Secure Corporation. We will call the former as the student dataset, and the latter as the malware dataset. Next, we will give a short description of the two datasets, how they were obtained, and what they consist of.

The student dataset The student dataset was automatically gathered from the students majoring in computer science during the first mandatory programming course. During the course students used an IDE plug-in that automatically collected snapshots from their exercises, e.g. when they sent the exercise to be evaluated to a server that ran a set of predefined tests on the exercise code, or when they saved, ran or built the project. Large amount of snapshots were gathered this way from each exercise each student attempted to solve. Our student dataset is a part of a larger dataset previously used, e.g. to assess how students try to solve programming exercises of various difficulty levels (see Hosseini et al. [11]), and how to automatically recognize which students are in need of assistance using machine learning (see Ahadi et al. [2]).

We restrict ourselves to only few exercises from the whole course. The exercises are hand picked so that the final programs are fairly complex and the assignments give the students the freedom to design the program architecture on their own. Furthermore, we only consider the last snapshot for each exercise individual student has returned. These snapshots are all successful solving attempts (i.e. all tests for the particular exercise are passed). In our experiments we handle the identity of the exercise as the true class and each successfully returned solving attempt for an exercise is a data point belonging to that class.

The dataset consists of five Java programming exercises and the number of students returning different exercises vary from 47 to 101 with a total number of 366 exercises returned. On average returned exercises have 38 functions from which 19.63 are non-local (Java's standard library functions). Minimum number of functions in any exercise is 2 and non-local functions 1, whereas maximum number of all functions is 78 and non-local functions is 36. Total amount of unique non-local functions called in the dataset is 170.

The malware dataset The malware dataset was obtained from F-Secure Corporation, and is the same that is used in the works of Kinable et al. [16, 18, 17]. It contains 194 malware variants – distinguished by hash codes, i.e. character strings derived from the file's content – in 24 different families, family sizes ranging from 2 to 17.[8] Family labels for the malware variants were given by experts; variants labeled to the same family are considered to have more similarities among them than variants labeled to different families. Later, we refer to these labelings by

[8]One hash code of a malware variant was discovered to be tagged into two different families, reducing the amount of unique call graphs to 193. To avoid confusion, we removed the variant from the larger malware family.

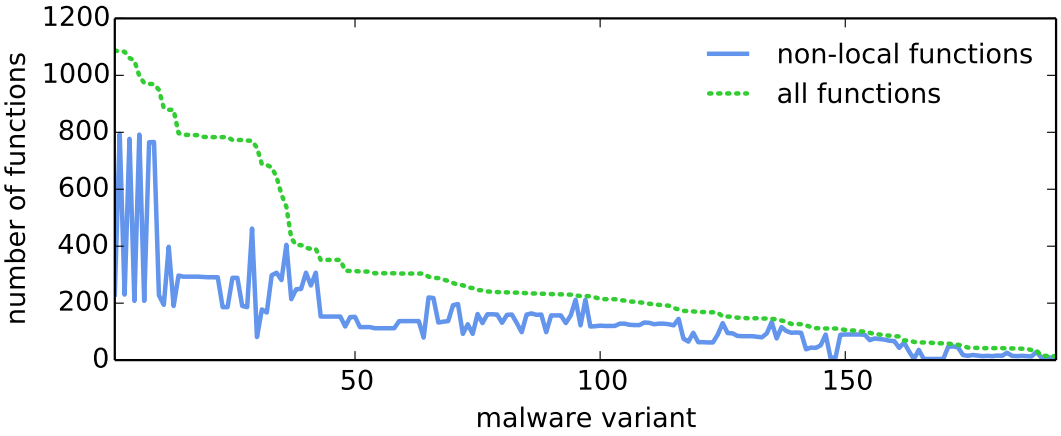


Figure 10: Number of all functions and non-local functions per malware variant, sorted by the number of all functions.

experts as true labels or true classes.

The call graphs in the datasets are context insensitive, and constructed from the information gained via static analysis. The IDA Pro [9] disassembler and some other utility tools were used to obtain call graphs from files in 32-bit Portable Executable (PE) file format – which is used in, e.g., executables and DLLs on Windows operating system – on the x86 architecture. As all the call graphs are obtained from binaries built to the same operating system and architecture, we can use our assumption that non-local functions with matching names can be considered equivalent.

As the amount of families and varying family sizes suggests: the dataset is quite diverse. The following, are some key aspects of the dataset presented in figures.

Figure 10 shows the number of all functions and non-local function per malware variant, sorted by the number of all functions. Here, we observe that the amount of local functions does not predict the amount of non-local functions as the two curves do not have similar characteristics. In some variants nearly all functions are non-local and in some variants about two-thirds can be local functions.

Figure 11 shows the number of malware variants containing each non-local function. The immediate observation is that about ten non-local functions are in nearly all of the malware variants. Another interesting observation lies at the other end of the curve, as two-thirds of the non-local functions called in the whole dataset appear in only few variants, and over a thousand non-local functions are found in exactly one variant. This suggests that feature pruning will have strong

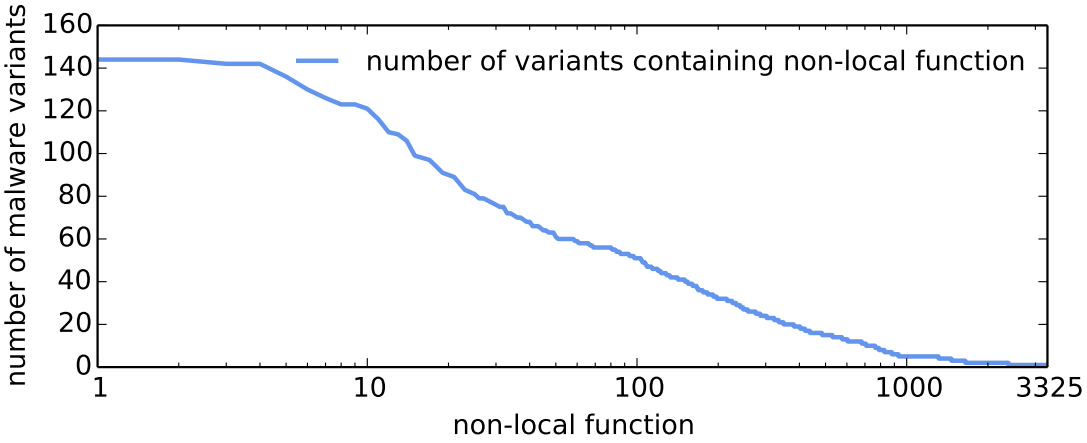


Figure 11: Number of malware variants containing each non-local function found in the malware dataset, sorted by the number of variants containing the non-local functions.

effect on the size of the feature sets as each feature that has a non-local function found from only one variant is pruned. For example, in the case of $k = 1$ the feature set is pruned to about two-thirds.

Figure 12 shows the fraction of matching non-local function names between malware variants (darker colors indicate more matching functions), sorted by the family and family size. The ticks in the graphs are between the different malware families. Looking at the figure we immediately observe that generally the amount of matching non-local functions is higher inside the malware family than between families. However, there are some families for which several variants share more functions with another family's variants than with variants of the same family. It is important to notice here that the true family labels are set by analyst, not a ground truth which could not vary depending on the analyst applying the labeling.

7 Experiments

Having defined our features, inspected some of their basic properties and described an algorithm to generate them. We will move to evaluate how well our features actually perform in different experiments. By performance we mean both the success in various tasks, and the execution time to generate the features. The goal of our experiments is to find answers to the following questions:

1. How do the choices of k and d affect the performance?

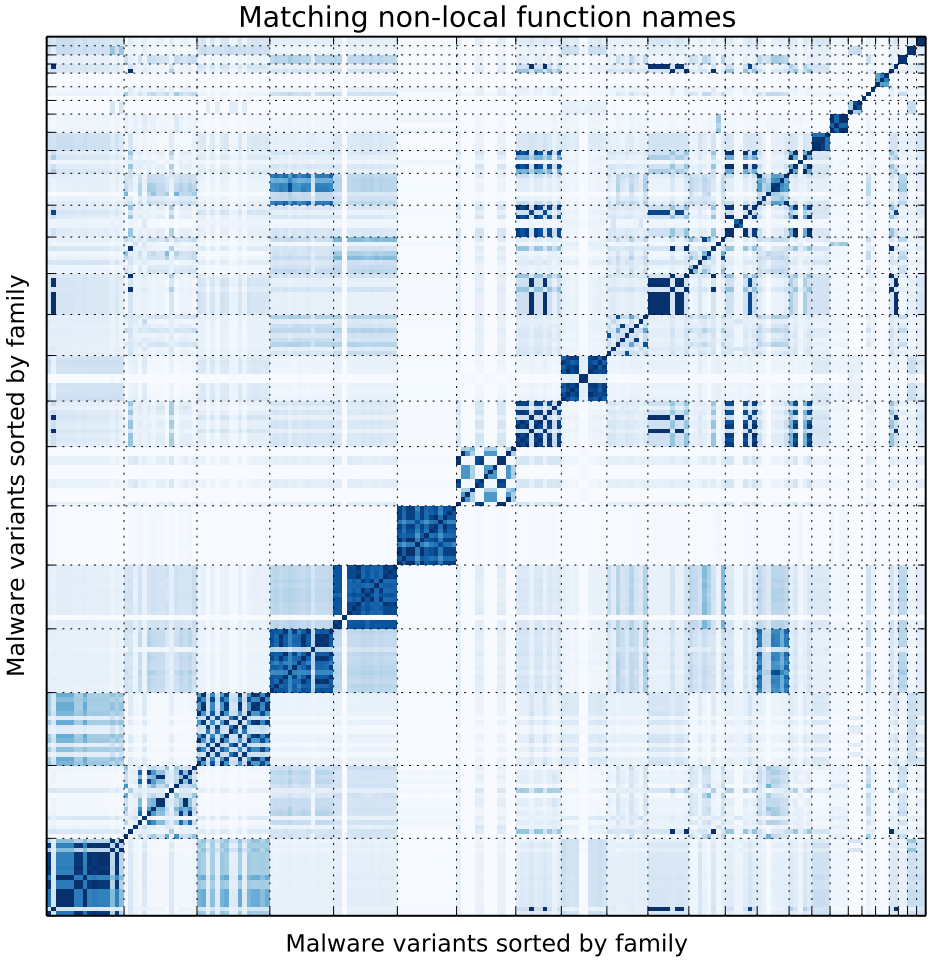


Figure 12: The fraction of matching non-local function names between pairs of malware variants. Dataset has been sorted based on malware families. Darker colors indicate a larger proportion of matching functions.

2. How do (k, d)-grams perform in comparison to other methods?

The experiments to answer the questions above are twofold. For Question 1., we evaluate how well (k, d)-grams perform with different values of k and d in retrieval tasks in Section 7.1. Based on the results of the retrieval tasks, we fix the values of k and d to the ones with the best performance, and perform clustering experiment with these values in Section 7.2. To answer Question 2., we compare the performance of the (k, d)-grams to the baseline of non-local function sets (effectively $(1, 0)$-grams) for both datasets, and for the malware dataset to the

Experiment setting	Default value
maximum cardinality of the feature sets, k	2
maximum distance of the non-local functions, d	3
feature value type	binary
preprocessing: tightening of topology	No
postprocessing: pruning	Yes
postprocessing: feature weighting	Yes

Table 1: Default values used thorough the experiments.

GED approximations of Kinable and Kostakis [17].

We do not report in the experiments all the possible combinations of the pre- and postprocessing steps of the algorithm, or the exact performance of each feature value type in each experiment. However, we will give a rough estimation of how large performance differences between our default settings, shown in Table 1, and some of the combinations are in the retrieval tasks. We have chosen this approach because (1) it keeps the experiments section uncluttered, and (2) the performance difference between the different setting decisions is in most cases nominal.

We have chosen the default values based on preliminar empirial testing. We do not use tightening of the topology as a preprocessing step because it gave us worse performance across the board. Also, we use pruning and feature weighting as they seemed to be effective in increasing the performance. However, the performance increase was in most cases negligible, and they could be left out without a significant drop in performance. For the feature value type, we have chosen to use binary value for its simplicity. In our preliminar tests all the feature value types gave similar performance, the overall winner varying between different values of k and d.

7.1 Information Retrieval Tasks

We use information retrieval tasks to get a basic understanding of how (k, d)-grams perform. For testing the information retrieval task we use both datasets, and compare our results to a baseline and in addition the results of the malware dataset to the results obtained with GED approximation of Kinable and Kostakis [17]. The information retrieval tasks we use are precision and mean average precision (MAP). Next, we will describe both tasks, and then show the results for the tasks for different k and d values, how the selection affects the feature generation time, and how the best performing k and d perform in comparison to GED approximation

(for the malware dataset) and baseline.

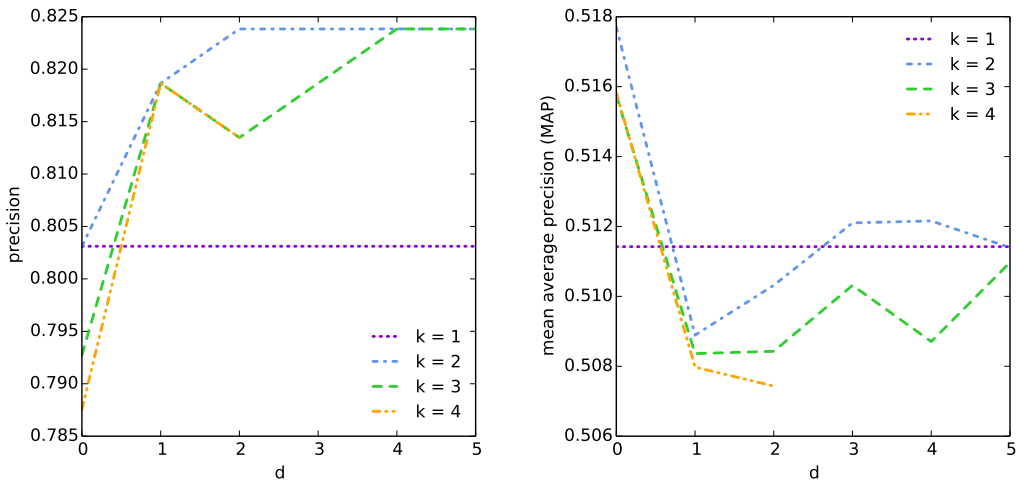
Precision Precision computes the percent of the retrieved data points that are relevant to the query. In our case, as we have a number of classes in the datasets, we calculate precision to be the percent of data points that have the nearest data point in the same class. Let S be the dataset; dist$(\cdot)$ be a distance function defined for all data point pairs in the dataset S; C be a set of possible labels; a labeling function $L : S \to C$ assigns the true label for each data point $s_i \in S$; and c_i be the data point closest to data point s_i, that is $c_i = \arg\min_{x \in S \setminus s_i} \text{dist}(s_i, x)$. Now, precision is calculated as

$$\text{precision} = \frac{|\{\, s_i \mid s_i \in S : L(s_i) = L(c_i) \,\}|}{|S|}. \tag{5}$$

Mean average presicion (MAP) Mean average precision captures performance over the whole dataset better than precision. It is an aggregate measure over average precisions of all the data points in the dataset and is shown to have good stability [23]. An average precision of a data point is computed using a precision-at-k, where k is the amount of documents we want to retrieve. It is computed as the percentage of k closest data points that belong to the same class as the data point used as a query. An average precision of a data point s_i is then the average of each precision-at-k up to a maximum k_i for the data point s_i. In our case, we fix $k_i = t_i - 1$ to be the maximum k for the data point s_i, where t_i is the number of data points in the class the data point s_i belongs to in the dataset S. Mean average precision is then the mean of all the average precisions of the data points in the dataset S. Let $p_i(k)$ be the precision-at-k for a data point s_i, then, the mean average precision over the whole dataset S, MAP(S), is computed as

$$\text{MAP}(S) = \frac{1}{|S|} \sum_{s_i \in S} \frac{1}{k_i} \sum_{k=1}^{k_i} p_i(k). \tag{6}$$

The malware dataset The results for the malware dataset's retrieval tasks are shown in the Fig. 13 and Fig. 14. Figure 13 shows the results for precisions and mean average presicions, while Figure 14 shows the number of features generated and their generation time. In both figures, the horisontal line for $k = 1$ shows only the result of $k = 1$ and $d = 0$, as there is not much point in generating features for $d > 0$, when $k = 1$ (the resulting feature set is the same or smaller). Furthermore, due to exponential running time increase, the results for $k = 4$ are only computed for $d \in \{0, 1, 2\}$



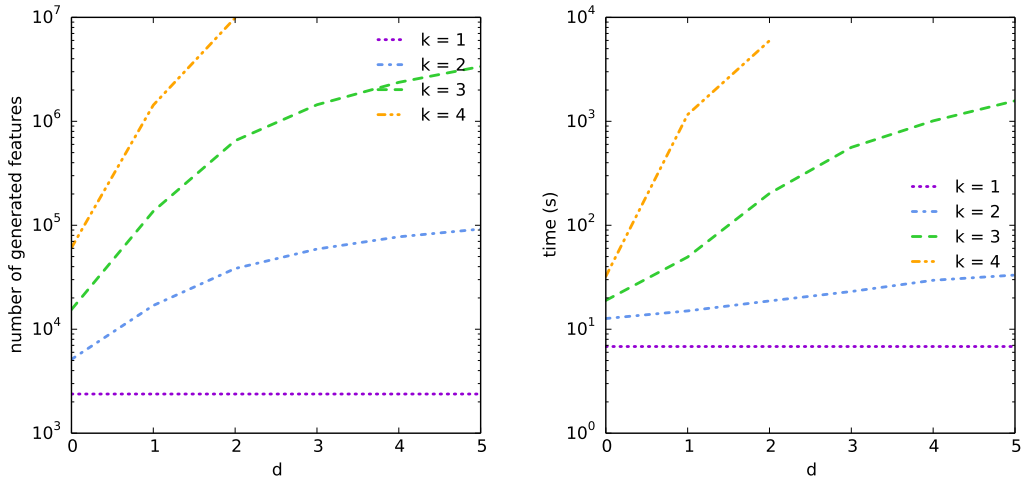
(a) Precision (b) MAP

Figure 13: Precisions acquired for the malware dataset with different k and d values, other parameters are set to default values as described in Table 1. Figure 13a shows precision and Figure 13b shows results for mean average precision.

Looking at Figure 13 we can immediately see two things. First, the choice for k and d does not seem to have much effect on the results. Second, the performance of the best parameter values ($k = 2$ and $d \in \{3, 4\}$) differs only slightly from the baseline results with $k = 1$ and $d = 0$, which effectively compares the non-local function sets between the call graphs. Here, it is interesting to note that, as the precise choice of d does not affect the results much, it could suggest that (1) the call graphs in the malware dataset are already quite compact, and (2) the refactoring or obfuscation processes affect the local topology of the call graphs much more often than non-local topology. On the otherhand, as also increasing the k does not have much effect on the results, we could say that the d-reachable k-grams are quite robust features for capturing similarity between the call graphs in the malware dataset.

In Figure 14, we have much clearer – and expected – result: the higher the k and d the more features are generated, which has a straightforward effect to the generation time. Unsurprisingly, the effect of k is much bigger than d, as the number of possible non-local function combinations grows exponentially with every increase of k, while d only affects the localness of the feature search.

Looking at the Figures 13 and 14 together we can come to a conclusion that, for the malware dataset, the $k = 2$ and $d \in \{3, 4\}$ seems to acquire the best performance, as the retrieval tasks give their best scores, and the generation time of the features



(a) Features

(b) Time

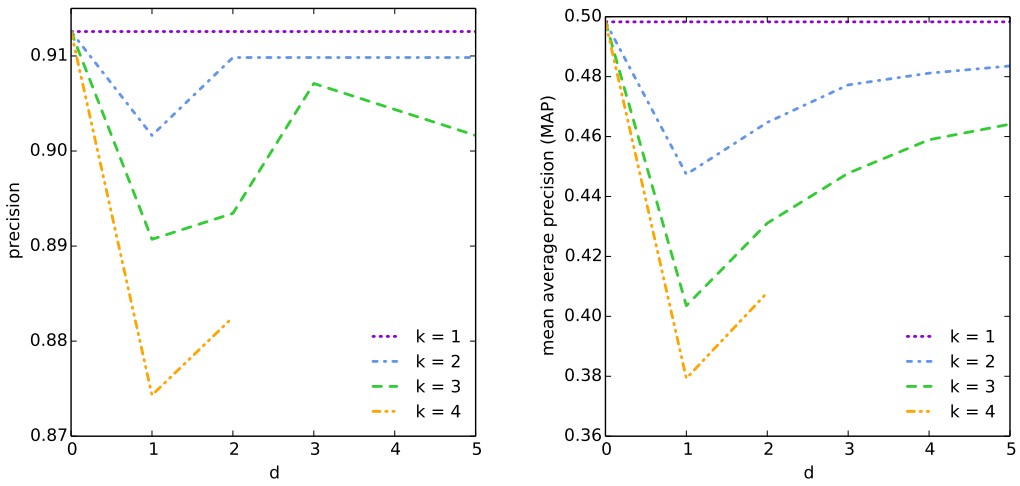
Figure 14: Number of generated features from the malware dataset (Figure 14a) and the time needed to generate them (Figure 14b) for different k and d values.

is still quite low.

To give an image of how different setting choices affect the results, we generated $(2,3)$-grams with different feature value types and without postprocessing steps separately. We observed that the feature value type choice had very little effect in the retrieval task performance. The maximum distance and the sum of distances feature value types for $(2,3)$-grams gave results that differ less than 0.01 for the precision and 0.003 for MAP when compared to the results obtained with binary feature values. Postprocessing had more pronounced effect on the results. When we removed both postprocessing steps, pruning and weighting of features, the precision dropped about 0.02 and MAP about 0.05 for $(2,3)$-grams with binary feature value. Removing only the pruning recudes the precision about 0.01 and MAP less than 0.004.

Now, let's compare our results to the results obtained with graph edit distance approximation by Kinable and Kostakis [17]. We were given a distance matrix computed with the method, and we ran the same precision and MAP tests for the distance matrix. The results for GED approximation are shown in Table 2. In the table we can see that the GED approximation outperforms our features when comparing mean average precisions, but is slightly inferior in precision.

To see how meaningful the results are, we run the retrieval tasks with all the methods (i.e. $(1,0)$-grams, $(2,3)$-grams and GED approximations) 1000 times with bootstrapping and get 95% confidence intervals of +/- 5% for each method.



(a) Precision (b) MAP

Figure 15: Precisions acquired for the student dataset with different k and d values, other parameters are set to default values as described in Table 1. Figure 15a shows precision and Figure 15b shows results for mean average precision.

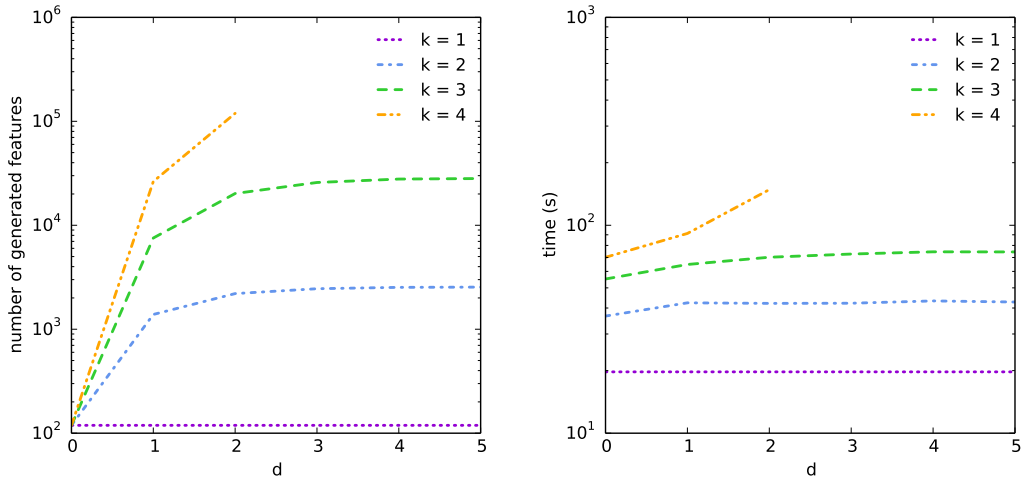
Such a wide interval implies that all the methods are actually performing on a similar level, and the choices of k and d for (k, d)-grams have little to do with the performance.

Task	GED	$(2,3)$-grams	$(1,0)$-grams
precision	0.8135	0.8238	0.8031
MAP	0.5492	0.5121	0.5114

Table 2: Retrieval task results for the malware dataset obtained with graph edit distance approximation of Kinable and Kostakis [17], $(2,3)$-grams and $(1,0)$-grams.

The student dataset For the student dataset, we perform the same retrieval tasks but do not compare the results to the GED approximations. Figure 15 and Figure 16 show the obtained results. In Figure 15 we observe that the best performance for the retrieval tasks is actually obtained with the baseline non-local function sets of $(1,0)$-grams, but the differences in this dataset are also quite negligible between various k and d selections. Peculiarly, the performance seems to drop from $d = 0$ to $d = 1$, but increases again when $d > 1$. We will get back to possible causes of the above mentioned performance in Section 8.

In Figure 16 we see the same unsurprising result for the student dataset as with



(a) Features (b) Time

Figure 16: Number of generated features from the student dataset (Figure 16a) and the time needed to generate them (Figure 16b) for different k and d values.

the malware dataset: increasing k and d generates more features (Fig. 16a) and increases the time needed to generate them (Fig. 16b). However, here the overall increase in the number of features or in the runtime is not so large as in case of the malware dataset, because (1) the call graphs in the student dataset are on average much smaller than in the malware dataset, and (2) the student dataset is devoid of larger programs with hundreds of non-local and local functions.

Choosing the best values for k and d Having performed the retrieval tasks we will select the best performing k and d values for the clustering that is done in the next section. In our selection we will give a higher weight to the results of the malware dataset as it represents more well structured problem for call graph distance measurements (the assumption is that many of the malware variants are just obfuscated or altered versions of other variants in the same malware family). With this in mind we select $k = 2$ and $d = 3$ as our clustering parameters, as they give the best performance for the malware dataset, and have reasonably good performance for the student dataset.

7.2 Clustering

As the second and final experiment, we use our $(2, 3)$-grams to perform clustering for the datasets. The goal of clustering is to divide a set of data points into multiple

groups so that data points inside a group are more similar to each other than to data points in other groups. Clustering can be done in several ways, for example:

hierarchical (top-down) Splits the dataset consecutively into smaller groups.

hierarchical (bottom-up) Starts with each data point in its own group and consecutively selects groups to merge.

centroid-based Finds representative centroids from the data and builds clusters around them.

density-based Clusters are found in areas with high density of data points.

Each of the above mentioned methods have their own advantages and disadvantages, and selecting a suitable clustering method is usually acquired by analysing the data in hand. We will next describe how we selected the clustering method, spectral clustering, for our malware dataset. Then, we move to describe basic parameter settings for spectral clustering, and how we evaluate (without knowledge of the true labels) which parameter setting performs best. Finally, we perform the clustering with the chosen parameters, and evaluate performance with a set of key values used to describe clustering performance. The approach is similar to the one used by Kinable and Kostakis [16, 17], but we (1) use different clustering algorithm, and (2) compare the results obtained with our method to the results obtained with the method described in [17].

Choosing the clustering algorithm Selection of the clustering algorithm is often open for several choices, because of the plethora of clustering algorithms available (see, e.g [1]). Our choice for the clustering algorithm comes from the inspection of how the acquired distances behave when the dimensionality of the data is reduced. We use multi-dimensional scaling (MDS) [36, 19] for the dimension reduction. In effect, MDS aims to lower the dimensionality of the data by placing each data point into a low-dimensional space so that the distances between the data points are preserved as well as possible.

In Fig. 17 and Fig. 18 multi-dimensional scaling ($N = 2$) is applied to the distance matrix computed from the $(2, 3)$-grams of the datasets. Fig. 17 shows the result for the malware dataset and Fig. 18 shows the result for the student dataset.The different color-marker combinations of the data points correspond to the true classes.[9] In Figure 17 we observe that although a lot of data points are gathered in

[9]We use color and marker style to distinguish the true classes for visualisation purposes, but in

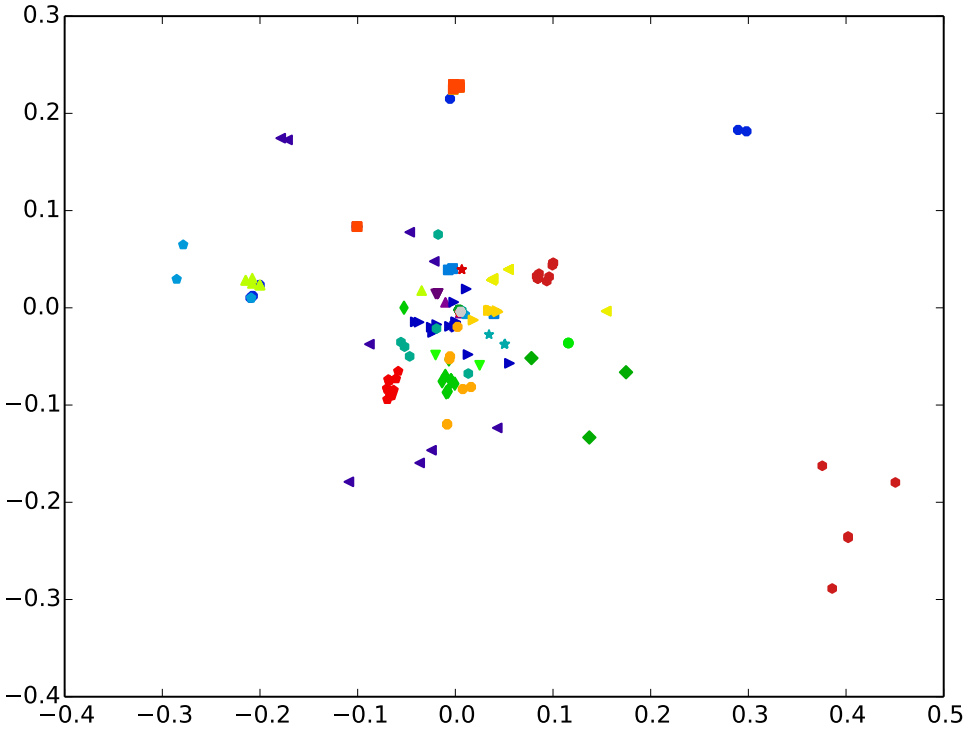


Figure 17: Multi-dimensional scaling for the malware dataset.

the middle there are also several distinct malware family clusters. Similarly, in Figure 18 we see that the data points for different exercises are quite well separated, with the two exercises in the middle overlapping each other. However, there are no clear gaps between the clouds of data points for different exercises.

Classical MDS uses eigenvalue decomposition to derive coordinate matrix. This suggests that our distance matrices could be suitable for spectral clustering (see, e.g. [32, 22]) as spectral clustering uses eigenvalues of the distance matrix to first reduce the data into lower dimension before applying the actual clustering.[10]

Spectral clustering The term spectral clustering refers to a family of clustering methods that use eigenvalues of the provided distance matrix to first project the data into a lower dimension before applying the clustering. The algorithm

a real case we would not have this information. Also, recall that the true classes of the malware dataset are determined by the malware analysts and the true classes of the student datase are the exercise identities.

[10]We also experimented with other clustering algorithms, but choose to use spectral clustering because it gave the best performance in preliminary empirical testing.

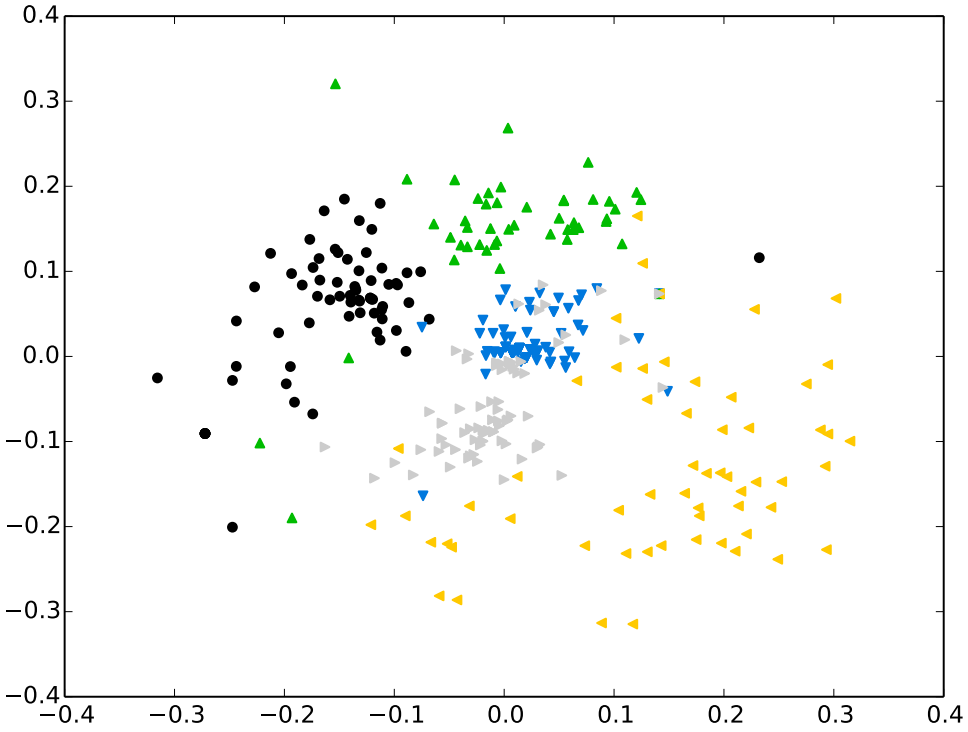


Figure 18: Multi-dimensional scaling for the student dataset.

starts with building a graph from the distance matrix with a chosen method, lowering the dimensionality according to the graph to selected amount of clusters K (effectively selecting the first K eigenvectors of the so called graph Laplacian), and then applying the well known k-means clustering algorithm with the selected K to the data in the lower dimension [22]. For building our graph, we use the simple k nearest neighbour (k-nn) method, where each data point is connected to its k closest datapoints in the distance matrix. We choose k-nn since it has the ability to connect data points from areas of different densities, but can also provide a graph with several disconnected components if the high density areas are well separated [22]. Since the selection of k in k-nn does not have a well studied basis, we will experiment with nearest neighbor values, $k\text{-nn} \in \{7, 9, 11, 13\}$.

Choosing the number of clusters After choosing the clustering algorithm and selecting its basic settings, we are left with the fundamental problem of selecting the proper amount of clusters, K, to which we will divide our dataset. We choose to evaluate both the k-nn and the number of clusters at the same time with a

silhouette value, a clustering measurement originally suggested by Rousseeuw [30]. Silhouette value aims to capture the fundamental definitions of clustering without explicit information of the true classes, i.e. it measures how similar a data point is to the data points in the same cluster, and how dissimilar it is to the data points in other clusters. In effect, it tries to measure how well a data point fits into its cluster. Silhouette value is always between -1 and 1, where high value indicates that the data points are well matched to their clusters.[11] Let $a(s_i)$ be the average dissimilarity of a data point s_i to other data points in the same cluster, and $b(s_i)$ be the average dissimilarity of s_i to the data points in the nearest cluster where s_i is not a member. Now silhouette value for a data point s_i, $\text{sil}(s_i)$, can be defined as

$$\text{sil}(s_i) = \frac{b(s_i) - a(s_i)}{\max(b(s_i), a(s_i))}. \tag{7}$$

With silhouette value for a single data point, we can define an aggregate measure of average silhoutte value over all datapoints in a dataset S to be

$$\text{sil}(S) = \frac{1}{|S|} \sum_{s_i \in S} \text{sil}(s_i). \tag{8}$$

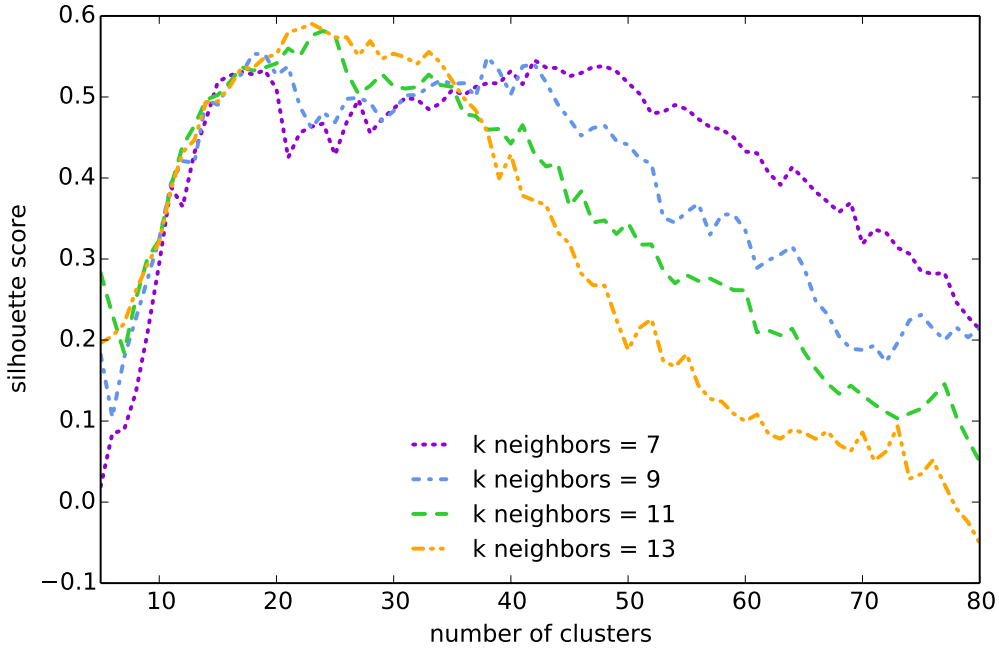
We use the average silhoutte score as the measure to verify how well the data points as a whole fit into their clusters. We run spectral clustering algorithm with a range of number of clusters and k-nn $\in \{7, 9, 11, 13\}$, and compute the average silhouette value from ten runs for each setting. For the malware dataset we let the number of clusters range from five to 80 and for the student dataset from two to 14.

The results of the experiment are shown in Figure 19. Figure 19a shows the results for the malware dataset, where we can see that selection of k-nn has some effect on the silhouette value, but the actual number of clusters has more effect as expected. The highest silhouette value for the malware dataset, 0.59, with a narrow marging stands in 23 clusters with k-nn = 13.[12]

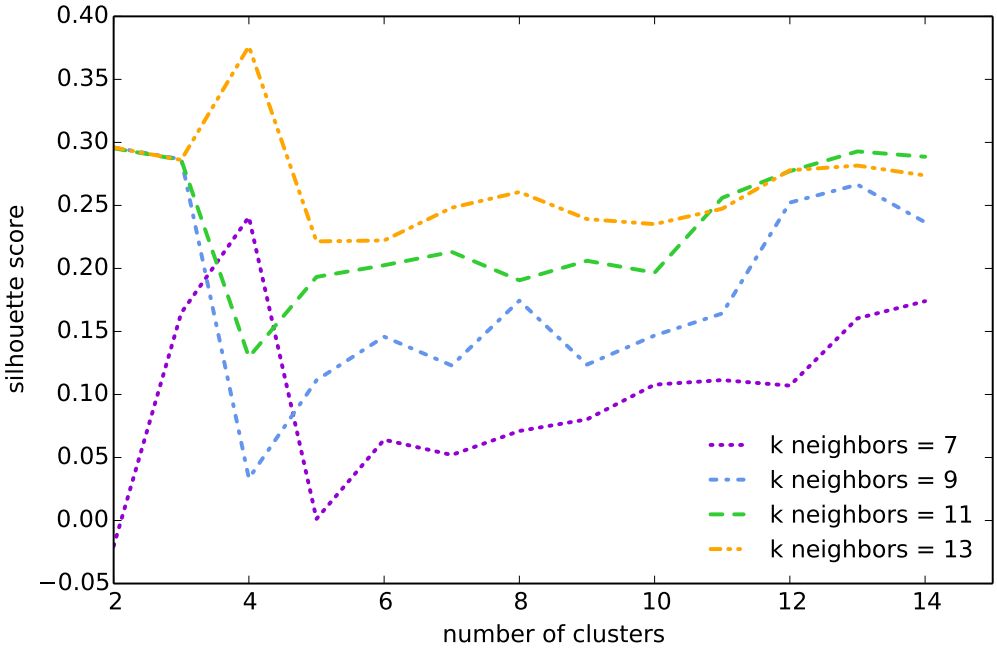
The results for the student dataset in Figure 19b are more complex. Overall, we observe that the silhouette values are not so high as in the malware dataset. Furthermore, there is a spike in the silhouette values when the number of clusters is four. For k-nn $\in \{7, 13\}$ the spike it is upwards, but for k-nn $\in \{9, 11\}$ it is downwards. This might be an anomaly on, e.g. how the graph from the data points is constructed in spectral clustering, but we still choose to select the clustering parameters with

[11] As a rule of thumb one could say that the values over 0.5 indicate that a reasonable structure has been found.

[12] Recall that the number of malware families (classes) in the dataset was 24.



(a) Silhouette values for the malware dataset.



(b) Silhouette values for the student dataset.

Figure 19: Silhouette values for $(2, 3)$-grams using spectral clustering with nearest neighbor kernel and different nearest neighbor options.

the best silhouette value, 0.37, which is obtained when the number of clusters is four and k-nn = 13.[13]

Next, we will use these clustering parameters to evaluate our clustering results for both datasets.

Results As we now have selected the clustering algorithm, and have evaluated the optimal clustering parameters for the datasets, we can analyse the clustering results with the selected parameters. In order to acquire meaningul comparison results for the malware dataset's $(2,3)$-grams, we use the same average silhouette value test find optimal parameter values for both GED approximation and $(1,0)$-grams, and acquire the number of clusters and k-nn value that maximizes the average silhouette value. We find that for GED approximation the optimal number of clusters is 21 and k-nn = 13, and for $(1,0)$-grams the number of clusters is 23 and k-nn = 13. As we do not have GED approximations for the student data set, we will only compare our $(2,3)$-grams to the baseline of $(1,0)$-grams. The optimal clustering parameters for the $(1,0)$-grams in the student dataset are ten for the number of clusters and k-nn = 13.

To evaluate our final clustering results we use external evaluation, where the knowledge of the true labels (or classes) is applied. The external evaluation measurement we use is V-measure, an aggregate measure which gives a singular value for the clustering performance. Next, we will give a short introduction to V-measure, however, we do not go over it in detail, and an interested reader is suggested to read the original paper by Rosenberg and Hirschberg [28].

V-measure V-measure is a conditional entropy-based external evaluation measure for clustering [28]. It addresses two important factors that intuitively define good clustering result: homogeneity and completeness. Homogeneity is satisfied in a clustering if all clusters contain only data points from a single class. On the other hand, completeness is satisfied in a clustering if all data points of each class are in a single cluster. V-measure is then the weighted harmonic mean of the homogeneity and completeness. Let a_{ij} be the number of data points belonging to a class c_i that are members of a cluster k_j, and N be the number of data points. Then, homogeneity is computed as

$$\text{homogeneity} = \begin{cases} 1 & \text{if } H(C,K) = 0 \\ 1 - \frac{H(C|K)}{H(C)} & \text{otherwise.} \end{cases} \tag{9}$$

[13]Recall that the number of different exercises (classes) in the student dataset was five.

where

$$H(C \mid K) = -\sum_{k=1}^{|K|}\sum_{c=1}^{|C|}\frac{a_{ck}}{N}\log\frac{a_{ck}}{\sum_{c=1}^{|C|}a_{ck}}, \text{ and}$$
$$H(C) = -\sum_{c=1}^{|C|}\frac{\sum_{k=1}^{|K|}a_{ck}}{N}\log\frac{\sum_{k=1}^{|K|}a_{ck}}{N}.$$

Similarly, completeness is computed as

$$\text{completeness} = \begin{cases} 1 & \text{if } H(K,C) = 0 \\ 1 - \frac{H(K|C)}{H(K)} & \text{otherwise.} \end{cases} \tag{10}$$

where

$$H(K \mid C) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{a_{ck}}{N}\log\frac{a_{ck}}{\sum_{k=1}^{|K|}a_{ck}}, \text{ and}$$
$$H(K) = -\sum_{k=1}^{|K|}\frac{\sum_{c=1}^{|C|}a_{ck}}{N}\log\frac{\sum_{c=1}^{|C|}a_{ck}}{N}.$$

V-measure is now defined as

$$V_\beta = \frac{(1+\beta)\times\text{homogeneity}\times\text{completeness}}{(\beta\times\text{homogeneity}) + \text{completeness}},$$

where β is a weighting factor. If $\beta > 1$, then completeness is weighter over homogeneity, and if $\beta < 1$ then homogeneity is weighted more strongly. We use $\beta = 1$ to give completeness and homogeneity the same weight.

Because of the normalizing factors $H(C)$ and $H(K)$ in Eq. 9 and Eq. 10, all the measures are invariant with respect to the number of classes in the dataset and the number of clusters. In addition, V-measure does not require a post-processing step where each cluster is assigned to a class, like many other measures that evaluate clustering performance [28]. This makes it an especially good clustering measurement, as it does not include in its evaluation how clusters are assigned to classes, which might be based on vague premises.

The computed homogeneity, completeness and V-measure scores for different clusterings for the datasets are shown in Table 3. For the malware dataset we can observe that all the clusterings perform on the same level, but both the $(2,3)$-grams and $(1,0)$-grams are slightly better in performance than the clustering done from the GED approximations. However, the differences are so neglible that the overall winner in performance is left undecided.

The difference of performance in the student dataset is larger between $(2,3)$-grams and $(1,0)$-grams. Surprisinly $(1,0)$-grams perform better than $(2,3)$-grams,

but closer inspection reveals that this is mainly due to the optimal clustering amount, determined by the maximal silhouette value, is larger for the $(1,0)$-grams. Larger number of clusters almost guarantees that the homogeneity of the clusters is higher. Overall the student dataset is not so well clustered as the malware dataset, but this was expected as the students are supposed to design their programs on their own, not modify existing programs.

	The malware dataset			The student dataset	
	GED	$(2,3)$-grams	$(1,0)$-grams	$(2,3)$-grams	$(1,0)$-grams
Clusters	21	23	23	4	10
k-nn	13	13	13	13	13
Homogeneity	0.7726	0.7983	0.7941	0.4340	0.6821
Completeness	0.7707	0.7779	0.7726	0.5404	0.5066
V-measure	0.7717	0.7880	0.7832	0.4814	0.5814

Table 3: The clustering parameters and the results for the datasets. Only the malware dataset is compared against the GED approximation.

8 Discussion

Having the experiment results we will now discuss about the results and some factors that might affect them. The discussed topics are not in any particular order.

The malware dataset The results for the malware dataset are a bit surprising as $(1,0)$-grams perform nearly identically to $(2,3)$-grams. Some explanation for this can be found if we look back at the Figure 12 on page 32. The figure shows us the fraction of matching non-local functions (i.e. the functions have the same name) between the malware variants. We observe immediately, that the fraction is generally higher inside the family than between variants of different families. From this observation, we draw two possible hypotheses for the cause: (1) the malware publishing groups and the automatic code obfuscators do not alter the non-local function calls so much, and (2) malware variants are often classified by analysts to families based on the vulnerabilities they exploit (e.g. calling some non-local function raises a possiblity for a buffer overflow that gives the malware an access to other parts of the system).

The above mentioned two hypotheses are of course related. If a certain malware variant aims to attack a known vulnerability in some part of the system, it might

not be able to change calls that are directly related to the system vulnerability as freely as it can change calls to other parts of the system, as the vulnerability may be only exploited with a certain sequence of actions. On the otherhand, as the analysts are likely aware of these exploits, they tend to group variants that act similarly (use the same vulnerabilities) to the same families as the family name is then associated with a known attack sequence.

Other matter is how well the results obtained with (k, d)-grams compare to the results of GED approximations. On the first sight it seems that the (k, d)-grams do astonishingly well compared to the more complex GED approximation (Table 2 and Table 3). However, we have to keep in mind that the malware dataset is really small, and the variants in the dataset have not been chosen with a spesific purpose to be a fair comparison between the methods. Certainly more experiments with larger datasets are needed to come to a definite conclusion of how well (k, d)-grams perform in malware classification and clustering.

The student dataset In the student dataset we had a similar revelation as with the malware dataset: $(1, 0)$-grams performed even better than more complex $(2, 3)$-grams in retrieval (Figure 15) and clustering (Table 3). The reason behind this might be twofold. Firstly, as the students are only on their first semester in the university, most of them are likely novices in programming. During the programming course they incrementally learn to use new library methods and code architectures, making the exercise classification easier as library functions are mainly used after they have been introduced in the course (giving larger set of possible library functions to the exercises appearing later in the course). Secondly, some exercises may implicitly require to use some library functions, even though the assignments are not directly forcing to use them, e.g. random movement of a game piece needs `java.util.Random`.

The clustering results of the student dataset seem at first little bit strange as the $(1, 0)$-grams have much higher homogeneity score than $(2, 3)$-grams, which affects the V-measure. However, this seems to be purely a cause of the optimal number of clusters determined by the maximal silhouette value. For $(1, 0)$-grams the optimal number of clusters was ten and for $(2, 3)$-grams four. It is almost guaranteed, that the clusters are more homogeneous when the number of clusters increases[14], and especially if it exceeds the number of true classes. As a comparison we run the clustering for $(2, 3)$-grams with ten clusters (other parameters unaltered) resulting

[14]Given that some of the true class structure is contained in the distance matrix from which the clustering is computed.

in homogeneity of 0.7464 and V-measure of 0.6209, which both are better than the results for $(1, 0)$-grams.

Speed and quality In our experiments we observed, that $k > 2$ did not give us better performance, but the increase in the generated features and in the runtime was exponential. With an assumption that the generated feature set grows faster the larger the dataset is, w.r.t k and d, we could say that mining more than $(2, d)$-grams is not reasonable. This would make the whole mining algorithm more streamlined, as the main algorithm for generating (k, d)-grams described in Algorithm 1 on page 25 would be stopped at line 6.

Pruning and weighting When increasing k we generate exponentially more potential feature candidates, a fraction of which are only present in one malware variant. This has a negative effect on distance computation between feature sets as a feature that is present in only one variant does not capture similarity between any two variants, and might only produce more noise to the distance computation.

Pruning can reduce the above mentioned problem, but it means that all the features in all the variants have to be taken into account, which can be time and space consuming in a large dataset. On the other hand, our feature weighting addresses the same problem from another point of view: the larger the k the smaller weights its features can have. Again, we need to know the number of all features for each k before we can weight them. As we have our best performance with both options turned on, it could be worthwhile to look into this matter more closely, and develop a feature set distance computation that inherently addresses these problems.

Reproducibility and overfitting The datasets we use to evaluate (k, d)-grams are small, and especiallly the malware dataset is tiny compared to the amount of malware variants found in the wild. Furthermore, we have mostly selected our methods by tweaking the parameters to give the best performance for our datasets. In effect, we need more experiments on larger datasets to evaluate how well the (k, d)-grams perform, and how do the values k and d affect the results on a larger scale.

However, comparing to the performance of GED approximation and baseline method (non-local function sets) we can see that the method gives comparable results. Also, as we get very similar results with many different parameter settings

(different k and d), we can say that our the (k, d)-grams are quite robust on how the parameter values are chosen.

9 Conclusions and Future Work

We have proposed structurally oriented call graph features, d-reachable k-grams, and discussed about their proterties and limits. For the features we have given an algorithm that mines from a call graph all the features up to a certain k and d, and named these feature sets as (k, d)-grams. Using the (k, d)-grams we have defined a distance measure between two call graphs and given several ways to compute the actual values for the features based on the call graph they are generated from. We have experimented with the (k, d)-grams using two datasets and have shown that the features perform relatively well in information retrieval and clustering tasks. Moreover, the (k, d)-grams perform in our experiments on a level similar to the more complex GED approximation for one of the datasets. However, more experiments are needed to evaluate their performance on larger datasets.

Even though our proposed features might fall short when using larger datasets, they have some advantages over the GED approximations. Foremost, the costly GED approximation has to be done to each call graph pair separately, which makes it even more costly on larger datasets, wheras the call graph features can be generated from each call graph individually, and the distance measure can be computed efficiently over the generated feature sets. This makes the features suitable candidates for a similar system design as was done by Hu et al. [12]. The generated feature sets could be used in the first layer to efficiently compare a given malware variant to a large set of possible variants or their prototypical feature sets. Then, based on the decisions made in the first layer, the malware variant would be compared to a limited set of variants using GED approximation in order to find a small set of most similar variants from the whole database.

As a future work, in order to make the (k, d)-grams truly efficient in a larger scale, at least one issue needs to be addressed: how well (k, d)-grams that are not pruned and weighted handle large databases. If the performance drop without pruning and weighting is similar as with the small datasets, there is no issue, but if it is noticeably larger, we need to either come up with a distance measure that inherently takes pruning and weighting into account or make an efficient indexing scheme where the weights and features to prune are consistent across the whole database and are both efficiently accessable and alterable.

All in all, further research on behavior of d-reachable k-grams seems to be an

interesting future path to take, e.g. to see how well feature selection techniques could be applied to them in order to reduce the noise in the feature set.

Acknowledgments

This work has been supported in part by the Finnish Funding Agency for Innovation (Tekes) in the D2I project, and by the Academy of Finland in the COIN Center of Excellence.

References

[1] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications*. Chapman & Hall/CRC, 1st edition, 2013.

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 121–130, New York, NY, USA, 2015. ACM.

[3] C. Borgelt and M.R. Berthold. Mining molecular fragments: finding relevant substructures of molecules. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 51–58, 2002.

[4] I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of the 2008 Virus Bulletin Conference*, 2008.

[5] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, March 2007.

[6] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus Bulletin Conference*, 2004.

[7] Fred Cohen. Computer viruses: Theory and experiments. *Computers & Security*, 6(1):22 – 35, 1987.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, first edition edition, 1979.

[9] Hex-rays. The IDA Pro disassembler and debugger. `http://www.hex-rays.com/products/ida/index.html`.

[10] Harold Joseph Highland. A history of computer viruses - introduction. *Computers & Security*, 16(5):412–415, 1997.

[11] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a java programming course. In *Psychology of Programming Interest Group Conference, PPIG 2014*, pages 65–76, 2014.

[12] Xin Hu, Tzi C. Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[13] Jun Huan, Wei Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552, Nov 2003.

[14] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28:75–105, 3 2013.

[15] M. L. Kammer. Plagiarism detection in haskell programs using call graph matching. Master's thesis, Utrecht University, 2011.

[16] Joris Kinable. Malware detection through call graphs. Master's thesis, Aalto University, 2010.

[17] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *J. Comput. Virol.*, 7(4):233–245, November 2011.

[18] Orestis Kostakis, Joris Kinable, Hamed Mahmoudi, and Kimmo Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1516–1523, New York, NY, USA, 2011. ACM.

[19] J.B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[20] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[21] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[22] Ulrike Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[23] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[24] S. Nijssen and J.N. Kok. Frequent graph mining and its application to molecular databases. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 5, pages 4571–4577 vol.5, Oct 2004.

[25] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45:1–45:4, New York, NY, USA, 2010. ACM.

[26] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, June 2009.

[27] Kaspar Riesen, Michel Neuhaus, and Horst Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Proceedings of the 6th IAPR-TC-15 international conference on Graph-based representations in pattern recognition*, GbRPR'07, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.

[28] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning(EMNLP-CoNLL)*, pages 410–420, 2007.

[29] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[30] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, November 1986.

[31] A. Sanfeliu and K.S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:353–362, 1983.

[32] J. Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, Aug 2000.

[33] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition, 2012.

[34] Symantec. Internet security threat report 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf, 2013.

[35] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[36] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17:401–419, 1952.

[37] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. A similarity metric method of obfuscated malware using function-call graph. *J. Comput. Virol.*, 9(1):35–47, February 2013.

[38] X. Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724, 2002.