Data Min Knowl Disc
DOI 10.1007/s10618-014-0367-9

Classy: fast clustering streams of call-graphs

Orestis Kostakis

Received: 28 February 2014 / Accepted: 18 June 2014

Abstract An abstraction resilient to common malware obfuscation techniques is the call-graph. A call-graph is the representation of an executable file as a directed graph with labeled vertices, where the vertices correspond to functions and the edges to function calls. Unfortunately, most of the interesting graph comparison problems, including full-graph comparison and computing the largest common subgraph, belong to the NP-hard class. This makes the study and use of graphs in large scale systems difficult. Existing work has focused only on offline clustering and has not addressed the issue of clustering streams of graphs. In this paper we present Classy, a scalable distributed system that clusters streams of large call-graphs for purposes including automated malware classification and facilitating malware analysts. Since algorithms aimed at clustering sets are not suitable for clustering streams of objects, we propose the use of a clustering algorithm that relies on the notion of candidate clusters and reference samples therein. We demonstrate via thorough experimentation that this approach yields results very close to the offline optimal. Graph similarity is determined by computing a graph edit distance (GED) of pairs of graphs using an adapted version of simulated annealing. Furthermore, we present a novel lower bound for the GED. We also study the problem of approximating statistics of clusters of graphs when the distances of only a fraction of all possible pairs have been computed. Finally, we

Responsible editors: Toon Calders, Floriana Esposito, Eyke Hüllermeier, Rosa Meo.

The work was done while the author was with F-Secure.

O. Kostakis (✉)
Labs, F-Secure, Tammasaarenkatu 7, 00180 Helsinki, Finland
e-mail: orestis.kostakis@f-secure.com

O. Kostakis
Aalto University, 02150 Espoo, Finland
e-mail: orestis.kostakis@aalto.fi

Published online: 10 July 2014

present results and statistics from a real production-side system that has clustered and contains more than 0.8 million graphs.

Keywords Clustering · Streams · Call-graphs · Malware · Graph edit distance

1 Introduction

The number of uniques files produced daily has long surpassed the capabilities of manual analysis. Meanwhile, malware (malicious software) remains a highly prevalent threat for computer systems, their users and ICT infrastructure. In recent years, we have witnessed cases of malware that steals credit card information, users' accounts and Internet-banking credentials, extorts users into paying ransom and even targets uranium enrichment centrifuges in nuclear reactors. The total cost incurred on the global economy by the presence of malware has established the detection of malicious files as a major task for data security organizations. Furthermore, a continuous race is underway between malware authors and security researchers, in which the first invent new attack vectors and the latter reactively create defense mechanisms. To facilitate the work of security researchers and analysts, enhanced automation is imperative.

Over the past years, researchers have focused on executable file analysis techniques. These techniques can be divided into two categories: behavioral (runtime) analysis and static analysis, with each one having its own advantages and limitations. The most widely used automated methods apply runtime analysis by executing files in controlled (sandbox) environments (Rieck et al. 2008; Bayer et al. 2009; Moser et al. 2007a; Willems et al. 2007; Kolbitsch et al. 2009). However, malware authors have developed a plethora of anti-virtualization techniques to detect whether the file is executed on a computer in the wild or in a lab's isolated environment. In addition, instances of malware might require special conditions, temporal or environmental, to hold in order to express any behavior. This is not the case with static analysis, which attempts to analyze a file by statically examining its code (Briones and Gomez 2008; Carrera and Erdélyi 2004; Dullien and Rolles 2005; Flake 2004; Hu et al. 2009; Kinable and Kostakis 2011; Kostakis et al. 2011). On the other hand, code re-ordering, routine re-ordering, self-mutation and code-obfuscation techniques are used to evade pattern-based detections (Christodorescu and Jha 2004). While pattern-based signatures could be modified to take into account such modifications by searching over the whole file, this solution would make the scanning process slower and more prone to false alarms. An abstraction resilient to such attempts against static analysis is the *call-graph* (Ryder 1979). A call-graph is the representation of an executable file as a directed graph with labeled vertices, where the vertices correspond to functions and the edges to function calls. An example of a very small call-graph is depicted in Fig. 1. For modern software it is not unusual to encounter call-graphs of several thousands of vertices.

On a daily basis, data security companies currently receive hundreds of thousand of unique files. These files are submitted by clients, partners or other entities and form a constant stream of data. For each incoming sample, it has to be decided whether it is of malicious nature or not. At the same time, the number of unique new files is increasing constantly. Due to that, automated means of analysis are essential. A

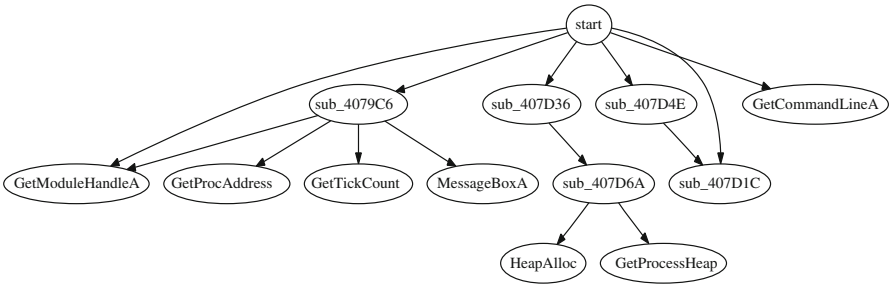


Fig. 1 A very small call-graph. Call-graphs are directed graphs with labeled vertices. Each *vertex* corresponds to a function and *edges* represent function calls. Call-graphs may contain cycles and self-loops

step toward achieving that is possessing the ability to identify similar files and group them together. For this purpose, we propose Classy, a system that clusters streams of call-graphs.

The benefits of a system like Classy is two-fold. First, it can be turned into an automated malware classification service by applying supervised learning through the clustering results. Second, it facilitates the whole stack of the Security Response Unit of an AV company; from the customer-care team, to the analysts dissecting malware and writing detections and the data scientists that strive to comprehend every aspect of malware and create new prototype solutions. For example, in the case of analysts writing pattern-based signatures, by providing them with real-time clustering information, Classy facilitates their work since they become aware of as many relevant samples as possible. Thus, the end result provides better and more accurate detection coverage, and the fact that information is provided in real time allows to save critical time for protecting the end-users. Alternatively, by examining a group of related malware samples, the analyst might decide that a different approach must be taken to better detect them in the future. Finally, the added knowledge gained from the clustering results may also be used for prioritizing samples in the queues of other automation systems.

Due to the fact that most interesting graph comparison-related problems are computationally hard, the use of call-graphs in large-scale systems becomes difficult. In particular, going from a restricted graph clustering application or offline experiment to a full-fledged online stream-clustering scalable production-side service is a very difficult and ambitious task. This is the fact that there has not yet been developed, to the best of our knowledge, a system such as the one that we describe.

In this work, we focus on clustering streams of data where each data object is a whole graph. This differs from the literature that shares the same title and deals with data that are part of one big graph, as in (Schaeffer 2007; Kulis et al. 2009; Zhou et al. 2009) and in social network analysis work (Mishra et al. 2007; Kollios et al. 2013). Consequently, our work differs from the existing publications that refer to clustering streams of graphs where the edges or vertices of a single large graph arrive in a streaming fashion (Aggarwal et al. 2010). In our application, one of the ultimate goals of clustering streams of graphs is to identify similarities between executables so the information can be used for facilitating the work of analysts in real-time and for automated malware classification. Due to that, the elapsed time from the moment

a sample enters the system until it has been clustered should be as short as possible. Given the rapid pace at which malware outbreaks occur, obtaining a result within an hour or even minutes is essential; most of the top anti-virus software vendors provide in-the-cloud services for real-time propagation of malware detections. An additional constraint is that since the system is operation-critical, there should not be any downtime for reasons such as re-computing the indexing structure. In our model of the stream, we can access data elements (in our case graphs) that have arrived in earlier time instances but we must handle every incoming data object independently and at the moment it arrives. Finally, and in addition to the reasons mentioned above, algorithms aimed at clustering sets of objects are not suitable for clustering streams. So, we face an additional challenge when designing such a system.

Our work focuses on call-graphs extracted from PE32 (Windows executable) files. However, the same file extraction techniques and clustering approach can be directly applied on Apple Disk Image (.dmg) files when the amount of malware for MacOS deems such system necessary. In general, our graph comparison and clustering methods are suitable for any application where graphs are data objects that arrive as a stream and must be clustered.

This paper mainly addresses the problem of transitioning from offline clustering to clustering streams of graphs in the context of a live production system. As an application of our technique, we use it to cluster call-graphs of binary executable files. The main contributions in this paper include:

- Classy, a scalable distributed system for fast clustering streams of call-graphs with the aim of facilitating the work of security analysts and automated malware classification. The graphs can consist of up to several thousands of vertices. The similarity of the graphs is determined on the basis of their graph edit distance (GED), a problem that is known to belong in the NP-hard complexity class. We are able to approximate the edit distance of graphs by using an adapted version of simulated annealing (SA). To the best of our knowledge, we are the first to present a system for clustering streams of graphs and especially graphs of such size.
- For the clustering phase, due to the fact that algorithms aimed at clustering sets of objects are not suitable for clustering streams, we propose and demonstrate the efficiency of an algorithm which relies on the notion of candidate clusters and reference samples in each cluster. We demonstrate, via thorough experimentation, that this approach yields results very close to the offline optimal; as those were presented in Kinable and Kostakis (2011).
- We present a novel lower bound of the GED with $O(n)$ time complexity that provides very high tightness and pruning power. The computation of the lower bound is used to prune the set of candidate clusters of the incoming sample.
- We study the problem of approximating statistics of clusters of graphs when the distances of only a fraction of all possible pairs have been computed.
- Finally, we present experimental results for those algorithms and provide statistics from a real active production-side system that has clustered and contains more than 0.8 million graphs. The results prove that such a system is highly valuable and under circumstances reliable enough to provide automatic classifications.

2 Related work

The study of call-graphs in the context of malware analysis was introduced by Flake in his seminal paper (Flake 2004). Similarly in his work jointly with Rolles (Dullien and Rolles 2005), the goal is to find differences between modified versions of software. By using the graph representation of executables and automatically finding commonalities, the work of a malware analyst is facilitated and reduced to having only to analyze the non-common sections.

Carrera and Erdélyi (2004) used call-graphs to create taxonomies for rapid analysis of multiple variants belonging to the same malware family. They use call-vector comparisons to heuristically match functions in an iterative match-and-filter manner. Improving on their work, Briones and Gomez (2008) incorporate control flow graphs of functions in the attempt to create a system which finds similar variants of malware from an existing DB. Recently, the first large-scale work on indexing and retrieving call-graphs, called SMIT, was presented (Hu et al. 2009). SMIT contains only malware, is intended to be used only by malware analysts and uses an improved version of bipartite matching for the graph comparison and the combination of $B+$ and VP trees for the indexing. However, despite its usefulness, the purpose of SMIT is to index call-graphs in order to provide k-NN search capability to the analysts and it is not intended for online clustering. Due to that, its complex combination of $B+$-trees and VP trees results into having to perform an excess amount of expensive graph comparisons per query; this becomes clear in Sect. 8.1.2. Finally, Kinable and Kostakis (2011) studied the feasibility of using call-graph clustering to identify groups of similar executables with the ultimate intention of automatically classifying malware.

All the aforementioned results have inspired multiple recent works that focus on variants of the call-graph or the trade-offs between accuracy and complexity requirements. For example, (Xu et al. 2013; Veeramani and Rai 2012; Bourquin et al. 2013; Elhadi et al. 2013) have focused on Windows PE32 files and (Gascon et al. 2013) have focused on Android `APK` files. The two works closer to Classy are SMIT and (Kinable and Kostakis 2011). Both of them use a similar graph edit distance as the distance function between graphs. The basic edit operations of the edit distance resemble very well the edit operations performed on executable files. Due to the fact that computing the graph edit distance is NP-complete, for approximating it in polynomial time, SMIT uses bipartite-matching that takes $O(n^3)$ time, while (Kinable and Kostakis 2011) employ a Simulated Annealing approach that has been demonstrated to be faster and more accurate than bipartite matching (Kostakis et al. 2011). Furthermore, in (Kinable and Kostakis 2011; Bunke 1997) the connection between GED and the Maximum Common Subgraph (MCS) is established; both problems are NP-hard.

Reducing the graph comparison problem to string or tree comparison creates ambiguities and violates basic distance function requirements, such as $\delta(G, H) > 0$ for $G \neq H$, that in turn may cause severe false positives. Furthermore, for the case of call-graphs, no assumption can be made regarding the statistical properties of the graphs. For instance, we cannot assume that the graphs have a bounded degree, nor can we assume any restriction on the (edge) density of the call-graphs. This renders methods designed for molecular graphs, such as (Schietgat et al. 2013), or other types of graphs, such as almost-trees (Akutsu 1993), unsuitable for our application. Similarly,

call-graphs may contain cycles and self-loops since those correspond to valid programming paradigms such as recursion. Consequently, methods designed exclusively for directed acyclic graphs (DAGs) (Lin and Kung 1997) are inapplicable, too. The interested reader may find an extensive review of matching algorithms for different types of graphs in (Conte et al. 2004). In general, any approximation approach would need to be extensively tested and its suitability must be benchmarked in the context of comparing call-graphs.

In recent years, graph kernels have been introduced for the task of learning structured data and have found success in the field of bioinformatics. They can be divided into three main categories: graph kernels based on walks or paths, based on subgraphs and based on subtrees. The work by Vishwanathan et al. (2010) improves the time complexity of the random-walk kernel computation between unlabeled graphs from $O(n^6)$ to $O(n^3)$. However, call-graphs are labeled and in that case their method takes $O(dn^3)$ time per iteration, where d is the size of the label set. Recently, Kriege and Mutzel (2012) have proposed subgraph kernels for attributed graphs. However, there method taken $O(kn_p^{k+1})$, where $n_p = |V_G| \cdot |V_H|$ and k is the allowed recursion depth. Another option is the classic subtree kernel (Ramon and Gärtner 2003) that takes $O(n^2 4^d h)$ time, where d is the maximum degree, and is clearly inefficient for call-graphs since the maximum degree can be equal to n.

Recently, the family of Weisfeiler–Lehman (WL) graph kernels have been proposed (Shervashidze et al. 2011) as a framework that promises fast comparisons. Still, the WL shortest path kernel requires $O(n^4)$ time for comparing a pair of graphs, while the WL edge kernel takes $O(m^2)$ per iteration (where m is the number of edges, in call-graphs m can be close to n^2). While the WL subtree kernel appears promising in terms of complexity, in the experiments of the original work, it performs worst than the other two benchmarked WL graph kernels for most test cases in terms of classification accuracy. In addition, WL kernels have been proposed for undirected graphs, while directivity in call-graphs is a very important aspect. Transforming a directed graph to an undirected simply by ignoring the direction of the arcs would make different call-graphs to appear as similar. This is highly undesired for the application of malware clustering since pairing different files could have detrimental results. Other potentially more complicated methods for transforming directed graphs to undirected would need to take into account additionally the time and space efficiency requirements related to clustering streams of call-graphs. Nevertheless, studying the applicability of graph kernels for classifying call-graphs would benefit the field of malware detection. Before choosing them as the graph comparison method in the stream setting, the performance and behavior of graph kernels needs to be thoroughly examined offline. However, the clustering algorithm that we propose is capable of handling the results of any distance function.

Furthermore, there are important limitations to applying graph kernels to the stream clustering setting. In the inductive setting where the kernel is computed based on a training set, and an unknown graph needs to be classified, the unknown graph must be mapped to the feature space of compressed labels occurring in the training set. This may additionally require to store, for each known graph, information related to the data structures encountered while computing the kernel. In particular for the Weisfeiler-Lehman kernels, this takes $O(Nmh)$ space, where N is the number of graphs, m is the number of edges and h is the number of iterations; in our application m is often in the

order of thousands. Moreover, this introduces the problem of selecting and maintaining a training set that offers good coverage of malware and benign software, and is up to date with current and past threat landscapes; in general, this is a serious problem for large-scale malware classifiers. Furthermore, this would require that the kernel is computed periodically and that would result to down-time of the system, or additional infrastructure. If graph kernels were to be applied in a transductive setting, the test set would need to be known. While information about past data structures would not need to be stored, the majority of the computations would need to be repeated every time. Applying a trade-off between the number of repetitions of the computations and handling several files every time, would convert the stream setting into a batch setting. The incurred delay might not be tolerable in certain time-critical applications.

Since graphs are a very prevalent method to depict structure and relations, work on indexing and retrieving them exceeds the boundaries of the malware analysis domain. In the past years, significant work has been performed on this subject in the database field. Most of the work is focused on finding occurrences of small query subgraphs in databases of graphs (Cheng et al. 2009; Giugno and Shasha 2002; He and Singh 2006; Jiang et al. 2007; Tian and Patel 2008; Williams et al. 2007; Yan et al. 2005; Zhao et al. 2007). However, these cannot capture the typical modifications applied to software. Instead, we need to identify those graphs in the database which are similar to the query graph using full-graph comparison techniques. Closure-Tree (He and Singh 2006) was recently proposed as an indexing solution for this purpose but focuses only on very small graphs. Only SMIT, as mentioned earlier, focuses on large-scale graph indexing. In Sect. 8.1.2 we demonstrate how our approach results to fewer graph comparisons than SMIT and other embedding-based approaches.

Significant work has also been published on clustering streams of data. Aggarwal et al. (2003) proposed CluStream, a framework for clustering evolving data streams. Furthermore, they propose HPStream (Aggarwal et al. 2004) for projected clustering of high dimensional data. By defining clusters for groups of dimensions, they attempt to tackle the sparsity problem in high-dimensional spaces. HPStream was shown to be more effective than CluStream's full dimensional clustering. Guha et al. (2003) presented a single-pass variation of k-median which has a small memory footprint. Charikar et al. (2003) improve on (Guha et al. 2003) with their own version of the k-median technique. All the the above focus on the k-means or k-median approach which require an exact or maximum value for k, the number of clusters. In our problem, the value of k cannot be known in advance and can change over time due to the nature of streams. If one selects a small value for k, highly different files would be clustered together, while a big value for k would force the algorithm to divide the data, thus resulting in low grouping power. Furthermore, k-means/median approaches tend to form spherical clusters, which is not always the case with clusters of call-graphs. Intuitively, the life cycle of modern software would suggest clusters of arbitrary shape.

In our method, instead of guessing the value for k, it is enough to define a "similarity threshold" value; this is the limit on the distance value for two objects to be considered close. Such a threshold is a basic component of density-based methods. In (Kinable and Kostakis 2011) it is shown that density-based clustering approaches should be preferred over k-means or k-median techniques for clustering call-graphs. Recently,

Cao et al. (2006) presented DenStream, an approach for density-based clustering for data streams suitable for identifying clusters of arbitrary shape. The method relies on the presence of potential micro-clusters and outlier micro-clusters and manages to handle large amounts of data by effectively pruning those two sets. DenStream, despite being efficient, cannot be applied out-of-the-box to our problem for the reason that it requires being able to identify the micro-clusters that are closest to the arriving point. This cannot be done in our case due to the fact that, first, our data are graphs and not points in a multi-dimensional space, and secondly, that computing the distances between graphs is very expensive. Thus it would be practically impossible to compute all the required distance values for an incoming graph. We must note that the methods proposed in this work can be combined with other methods to provide greater efficiency. Examining approaches that combine elements of other methods with our method would be a very interesting direction for future work.

Finally, well-established indexing methods such as Locality Sensitive Hashing (Datar et al. 2004; Gionis et al. 1999; Indyk and Motwani 1998) cannot be applied to our setting. For LSH in particular, it remains an open problem to derive a scheme for GED. Even for even easier problems such as the string edit distance there have been no solutions that satisfy any performance guarantees.

3 Background

In this section we provide the reader with the necessary background. In Sect. 3.1 we describe our data and the means used for pre-processing the binary executable files. In Sect. 3.2 we introduce the used terminology and notation while in Sect. 3.3 we provide a formal definition of our graph comparison problem.

3.1 Data description and extraction

Call-graphs are extracted using static analysis. In other words, the file is not executed but instead it is analyzed by examining its contents. To statically extract the call-graph from a binary file we use a very popular disassembler, IDA Pro (Hex-Rays 2008), in conjunction with IDA Python (Carrera and Erdélyi 2004) and our in-house unpacking suite. The product is an instruction-level disassembly of the binary file from which we are able to get a list of functions and inter-function calls. Basic block information and control-flow graphs can also be extracted. The functions are labeled and belong to two categories: (file-specific) local functions and imported (external) functions. For the latter, their names can be retrieved (e.g. GetSystemDirectoryA) and serve as the labels of those vertices, while for the first the returned names are based on the address of their starting point. The names of imported functions are common across executables and can be retrieved from the Import Address Table (IAT), for those external functions that are imported dynamically. Alternatively, statically linked library functions and their canonical names can be retrieved with tools such as IDA Pro's FLIRT. Similarity of imported functions can be determined simply by comparing the function names. The same does not hold for the rest of the functions. No valid function calls are discarded. Thus, cycles and loops are present in our graph model.

Before a call-graph is extracted, it is first examined to determine whether it is packed or protected; in the remainder of this paper we will refer to both conditions simply as packed. Packing refers to techniques that are applied, in the usual case, during the post-compilation phase and aim to protect the intellectual property of the software author from reverse engineering efforts; typical examples include `UPX` and `ASPack`. Packers do not affect the behavior of an executable, but they alter its contents by compressing and/or obfuscating them. When a packed file is executed, an extraction component, also known as the unpacker's stub, recovers (decompresses and/or decrypts) the original contents of the file, restores the execution context, and the execution resumes from the original starting point.

To detect whether a file is packed, it is common to use pattern-matching techniques. Tools such as PEiD (Snaker et al. 2006) contain signature databases for a series of known packers. Once a packer has been identified, one needs to apply the appropriate unpacker. This approach is fast and works well for the vast majority of known packers, but for each custom/unknown packer a new signature and the corresponding unpacking mechanism need to be added. An alternative solution is to use heuristics. Most of them can be considered as run-time unpacking tools (Kang et al. 2007; Martignoni et al. 2007) and may require an isolated environment. In general, unpacking heuristics are of questionable reliability and can be evaded. For example, they may fail under the presence of anti-virtualization and anti-emulator techniques. Finally, more than one packing technique may be applied simultaneously.

In our system, if the file is packed, it undergoes the appropriate unpacking procedure(s) to remove any such layer. This is necessary, otherwise the graph extraction would return the call-graph representation of the unpacker's module. Creating the call-graphs of the unpacker's stub would result in clustering binary executable files based on the used packer and not the actual executable. In the case of heuristic unpacking, the process may fail to produce a valid executable or the product may contain severe inconsistencies. In such case the file must be discarded, since the output may be arbitrary.

Finally, static analysis is preferred over run-time analysis due to the fact that the latter allows to analyze only the part of the code that managed to get executed. This is an important factor, since modern malware may apply a wide series of anti-debugging and anti-virtualization techniques and do not exhibit important behavior if they detect that they are being executed in a controlled (sandbox) environment. As a result, call-graphs produced via static analysis are more accurate and provide greater code coverage. In turn, static analysis suffers from several limitations of its own. Most importantly, some values, such as register values, are resolved at run-time; for example `call eax;`. Recovering those values statically can be NP-hard (Moser et al. 2007b). Similarly, static analysis retrieves all function calls regardless of whether they are performed during execution. Determining which function calls (control flow transfers) may actually be performed is undecidable in the general case.

However, one could combine static and run-time analysis to extract the call-graph from the memory image; limitations regarding the need for parts of the code to be executed still apply. Furthermore, the most important challenge is determining the appropriate moment to "dump" the memory to disk, and identifying the appropriate memory regions that contain the interesting code, in a manner that allows the whole

procedure to remain effective, efficient and accurate. Still, any combination would require infrastructure for both techniques, but our system can work with such graphs too. Techniques and methods on improving the accuracy of the call-graph extraction would be highly beneficial for our system, but this is beyond the scope of the current work.

After a preliminary study of call-graphs and in conjunction with the malware threat landscape, we have decided to handle only graphs of up to 3000 vertices. In addition, extensive sanity checks are performed on the call-graphs to determine the validity of the output. This includes simple checks on the number of edges and vertices and more complex checks such as verifying the validity of certain structures created by the compiler. We prefer to discard samples for which we are not confident about the validity of their call-graph rather than erroneously clustering them, which could ultimately result in a severe false detection.

3.2 Graphs

A *call-graph* is a directed graph $G = (V, E)$ with labeled vertices, where each vertex (or node) $u \in V$ represents a program function and each edge, or *arc*, $(u, v) \in E$ a call of function $v \in V$ by function $u \in V$. Between any ordered pair of vertices, there can be at most one arc. u_N denotes the label, or function name, of $u \in V$. In directed graphs, the direction of the edges matters; $(u, v) \in E$ is different than $(v, u) \in E$. The outdegree $d^+(u)$ of a vertex u is the number of its outgoing edges $|\{(u, v) \in E | v \in V\}|$, while its indegree $d^-(u)$ is the number of its incoming edges. The sum of the outdegree and indegree of a vertex is simply its degree. For a graph G, we denote the outdegree sequence of its nodes in ascending order by D_G^+. u_i^{G+} denotes the vertex whose outdegree is in the i-th position of D_G^+. Symbols D_G^- and u_i^{G-} denote the same for the ordered indegree sequence.

The vertices can be split into two main categories: local (file-specific) functions and external (imported) functions. The number of vertices in graph G, $|V_G|$, is the *order* of G, while the number of edges, $|E_G|$, is the *size* of G. The set of external functions of G is denoted by S_G

3.3 Graph edit distance

Definition 1 (*Graph edit distance*) The Graph Edit Distance between two graphs G, H is the minimum cost induced by transforming graph G into H using elementary edit operations. Each edit operation is assigned a cost value.

In our case, these elementary edit operations are: adding/ removing a vertex, adding/ removing an edge and relabeling a vertex. We assign unit costs to all operations. Computing the GED of a pair of graphs is NP-hard (Riesen and Bunke 2009) and makes use of graph matchings; these are bijective functions mapping the vertices of one graph to those of the other.

Definition 2 (*Graph matching*) For two graphs, G and H, of equal order, the graph matching problem consists of finding a bijective mapping ϕ that maps the vertices in

V_G to those in V_H such that $\forall i, j \in V_G, (i, j) \in E_G$, if and only if $(\phi(i), \phi(j)) \in E_H$. If such a matching exists, ϕ is an isomorphism between G and H and the two graphs are isomorphic.

In the case of graphs with labeled or attributed vertices, the above definition is extended so that the mapping preserves the vertex labels or attributes.

The above definition requires that G and H have the same number of vertices. This is hardly the case in general. This can be circumvented by adding extra ('dummy') zero-degree vertices, denoted by ϵ, to the one with the fewer vertices. This results to the augmented versions $G' = (V_{G'}, E_{G'})$ and $H' = (V_{H'}, E_{H'})$, with $|V_{G'}| = |V_{H'}|$. Matching a vertex $u \in V_{G'}$ to an added vertex ϵ corresponds to deleting u from G. The reverse corresponds to adding a vertex in H. Given the above, we can define the three partial costs of the GED, which are VertexCost, EdgeCost and RelabelCost.

VertexCost The number of deleted/inserted vertices: $|\{u : u \in [V_{G'} \cup V_{H'}] \wedge [\phi(u) \in \epsilon \vee \phi^{-1}(u) \in \epsilon]\}|$.

EdgeCost The number of deleted/inserted edges: $|E_{G'}| + |E_{H'}| - 2 \times |\{(i, j) : [(i, j) \in E_{G'} \wedge (\phi(i), \phi(j)) \in E_{H'}]\}|$.

RelabelCost The number of mismatched functions. A function is mismatched if it is either a local function and matched to any external function, or the reverse, or an external function matched to an external function with a different name.

Finally, the edit cost $\lambda_\phi(G, H)$, incurred by a matching ϕ, is the sum of these value:

$$\lambda_\phi(G, H) = VertexCost + EdgeCost + RelabelCost \tag{1}$$

Hence, the GED problem breaks down to computing the matching that yields the minimum cost, i.e. $\min_\phi \lambda_\phi$. The edit cost incurred by matching two graphs is an arbitrary non-negative integer value. There is a need to acquire a normalized value. For this, we use the *graph dissimilarity*.

Definition 3 (*Graph dissimilarity*) The dissimilarity $\delta(G, H)$ between two graphs, G and H is a real value in [0,1], where 0 indicates that the graphs are identical while a value near 1 indicates that the graphs are highly dissimilar. Furthermore, it holds that: $\delta(G, H) = \delta(H, G)$, $\delta(G, G) = 0$ and $\delta(G, K_0) = 1$, where K_0 is the order-zero graph.

Finally, the graph dissimilarity $\delta(G, H)$ can be obtained by the graph edit distance $\lambda_\phi(G, H)$:

$$\delta(G, H) = \frac{\lambda_\phi(G, H)}{|V_G| + |V_H| + |E_G| + |E_H|} \tag{2}$$

We will use the term GED to denote both the GED and the normalized GED interchangeably depending on the context.

4 Classy: architectural overview

In this section we provide an overview of the architecture of our system and the sample processing pipeline.

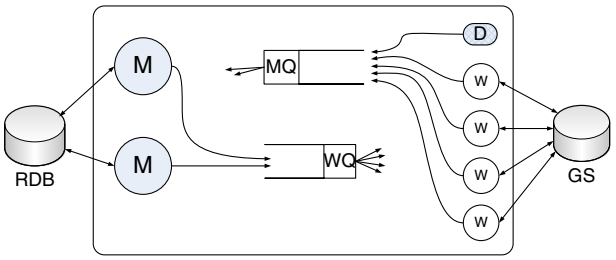


Fig. 2 High-level architectural view of Classy. The workers (*W*) perform the computationally intensive tasks, while the masters (*M*) craft the tasks. Communication happens using only two queues. The dispatcher (*D*) introduces tasks into the system. External dependencies are the graph-storage file-system (*GS*) and a relational database (*RDB*)

4.1 Architectural model

Our system is distributed and highly scalable. External dependencies are limited to a filesystem for storing and retrieving the graphs and a relational database (RDB) for maintaining information about the samples and the clusters. Any highly-available distributed filesystem (DFS) is sufficient since the use pattern is write-once-read-many; the graphs are never modified.

The architectural model relies on the presence of two types of processes: the workers and the masters. Communication between different types of processes happens through queues. There are two queues: the masters' queue, from which the masters fetch messages put by the workers and the workers' queue which is used in the opposite way. The workers are responsible for handling the most computationally intensive tasks (graph extraction, graph comparisons, etc). The masters are responsible for retrieving the results of the tasks completed by the workers and, based on the result, crafting a new worker task. They are the ones accessing and modifying the RDB. All actors are stateless so there can be multiple instances of them in parallel. Essentially, the masters push the information between the different stages of the sample-processing pipeline while the workers are the ones performing each stage. Finally, tasks are inserted to the system via a dispatcher who places the relevant messages in the masters' queue. The model can be seen in Fig. 2.

The architectural model can be seen as a double producer-consumer pattern; the masters produce tasks that are consumed by the workers, that in turn place the results in another queue to be processed by the masters. Consequently, it is possible to add more resources of each type on demand. The obvious requirement is that the DFS should scale with the amount of workers, and the RDB with the amount of masters. In Sect. 8.1.4 we provide more details and insights regarding our running instance of Classy, as well as a discussion about specific technologies that may be used and their limitations.

4.2 Process pipeline

Once our call-graph clustering system is presented with an incoming sample, the sample undergoes 5 phases as shown in Fig. 3.

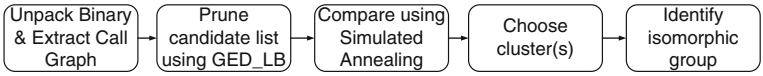


Fig. 3 The high-level sample flow pipeline. Each stage corresponds to a separate worker task

First is the call-graph extraction phase. This includes unpacking the sample. Many binary files, even benign ones, are packed and/or protected against debugging, dumping and disassembling. Removing the protective layer(s) before running the call-graph extraction tools is imperative, otherwise the call-graph would correspond to the protective layer and not the actual application. Additionally, at this phase samples can be discarded by the system if the unpacking is unsuccessful or if the unpacked binary turns out to be a file which should not be clustered (for example a RAR self-extractor). The second phase is the fast-filtering (or pruning) of the candidate-clusters list for that sample using the lower bounding method described in Sect. 6. This is the most I/O intensive part of the whole process. The third phase consists of comparing the incoming graph to those in the pruned candidate-cluster list. This phase is the most CPU-intensive. The fourth phase is the clustering phase. Given a list of graph distances the worker determines to which cluster(s) it should assign the incoming graph and if any more complex cluster operations, such as cluster merging, should be performed. Finally, if the incoming graph has not been identified as being isomorphic to any of the existing graphs, it gets compared to those graphs that belong to the same cluster and have the same order and size.

5 Graph comparison

The graph comparison is the most essential part of the whole system. Its accuracy and time requirements are in direct relation to the performance of the system from the scope of quality and throughput respectively.

Our work is aligned to that of (Hu et al. 2009) and (Kinable and Kostakis 2011) in using the same graph edit distance (GED) as the distance measure between pairs of graphs. In preliminary work (Kostakis et al. 2011) we have demonstrated that a Simulated Annealing (SA) comparison algorithm is faster and yields better results than the bipartite matching-based approaches for approximating GED; an example of the superior performance is depicted in Fig. 4, taken from the same work. While GED is a computationally hard problem, SA runs in polynomial time with respect to the input size.

Simulated annealing is a local search method. By starting from a random solution, it traverses the solution space in the attempt to find the global minimum (if it is used for minimization problems such as GED, global maximum otherwise). At every step of the process, a random neighboring solution is selected. Neighboring solutions are produced by choosing two vertices in G' and swapping the vertices in H' to which they are mapped by ϕ_i. If the new solution yields a better (lower) score for the objective function, that solution is always accepted and at the next step the process resumes with the new solution as its current. If the new solution does not yield a better score, it might be accepted with certain probability; this probability is controlled by a parameter called *temperature*. For computing the score of the neighboring solution it is sufficient

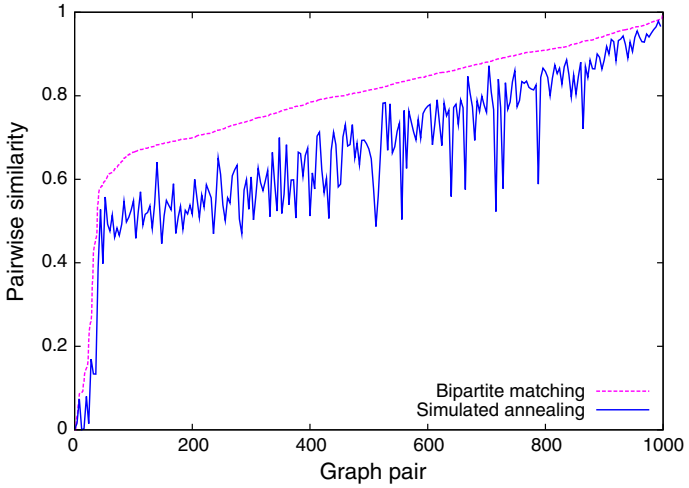


Fig. 4 Performance of simulated annealing versus bipartite matching when computing the graph edit distance for 1000 randomly chosen pairs of call-graphs; less is better. Taken from (Kostakis et al. 2011)

to simply compute the change to the score induced locally by swapping the matching counterparts. This alleviates the need to re-compute the total score at every step. The rationale for accepting worse solutions is to avoid getting trapped in local minima. Since the solutions returned by SA are of greater score than the ground truth, SA is an over-approximating algorithm. Algorithm 1 depicts the SA process in pseudocode.

The value returned is the lowest score witnessed for λ_{ϕ_i}. The time complexity of SA for comparing graphs G, H is equal to $O(k \cdot |V_{\max}|^2 \cdot d_{\max})$, where $|V_{\max}| = \max\{|V_G|, |V_H|\}$, k is the number of relaxation iterations and $d_{\max}$ is the maximum degree of the nodes in $V_G \cup V_H$. The value k can either be a fixed input or SA can be

Algorithm 1 Simulated Annealing for computing GED

Require: Graphs G, H, Annealed parameter values β_0,β_{final}, Cooling rate ϵ, Iterations per relaxation m

ϕ_i = random_ϕ()
$\beta = \beta_0$
while $\beta < \beta_{final}$ **do**
  for m iterations **do**
    $\phi_{i+1} = neighbor\_solution(\phi_i)$
    $\Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}}) = \lambda_{\phi_{i+1}} - \lambda_{\phi_i}$
    if $\Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}}) < 0$ **then**
      $\phi_{i+1} = \phi_i$
    else with $Pr(e^{-\beta\Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}})})$
      $\phi_{i+1} = \phi_i$
    end if
    if $(\min_{\phi} \lambda_{\phi} == \lambda_{\phi_i})$ OR no_progress() **then return** best ϕ
    end if
  end for
  $\beta = \beta/\epsilon$
end while

allowed to progress until its terminating conditions are met. Experiments have shown that when SA is allowed to reach either of its terminating conditions, the value of k (the number of iterations) is close to $\log n$. There are two terminating conditions for SA. The first is achieving the latest possible score. But since this is the problem SA is called to solve, a lower bound is computed and used instead. The second terminating condition comprises of terminating the SA process when no better solution has been identified within a certain number of the most recent neighboring solutions; this is the no_progress() function in Algorithm 1. For more details about the SA comparison method we refer readers to (Kostakis et al. 2011).

One of the advantages of SA is that it attempts to solve the problem directly instead of applying some sub-optimal heuristic like this is done by the bipartite-matching approaches. In addition, since it is a local search method, it has the advantages of an *anytime algorithm* (Dean and Boddy 1988). At any given point during the execution of a problem instance, a valid solution is maintained. So, in the case that the execution is terminated even before the halting conditions have been satisfied, a result can still be obtained. If given more time, SA continues to improve the solution. This provides the flexibility to impose timeout thresholds so to adhere to various Service Level Agreements (SLAs).

6 Lower bound of the graph edit distance

The number of candidate clusters that a sample has to be compared against might be very high. While the SA algorithm might be fast enough for the problem that it solves, it is necessary to keep the number of comparisons to a minimum. Essentially, we only need to know which comparisons might provide a score below the threshold for which the clustering algorithm would consider the samples to be similar. Since SA is an over-approximating algorithm, a lower bound on the value of GED for a pair of graphs is a lower value on the score returned by SA. We have devised a $O(n)$ lower bound for GED, which we denote as GED_{LB}. The lower bound presented in (Zeng et al. 2009) is not suitable since the $\Theta(n^3)$ time complexity is comparable to the complexity of approximating GED via bipartite matching or SA. GED_{LB} is based on lower bounding individually each of the coefficients of Eq. 2:

(i) For the cost of adding or removing vertices, we know that $VertexCost \geq ||V_G| - |V_H||$. By editing one graph to acquire the other, one would need to perform at least as many vertex deletion (or addition) operations as the difference of the two graph orders.

(ii) For the cost induced by relabeling nodes, we compute the intersection of the sets of system calls in each graph. We know that $\max\{|S_G|, |S_H|\} - |S_G \cap S_H|$ vertices do not have an equivalent counterpart in the other graph. However, these vertices might be eventually deleted, thus contributing to $VertexCost$. So, to avoid any double counting, $RelabelCost \geq \max\{|S_G|, |S_H|\} - |S_G \cap S_H| - V_c$. Of course, this value might be negative, so we take $\max\{0, \max\{|S_G|, |S_H|\} - |S_G \cap S_H| - V_c\}$.

(iii) For the cost of adding or removing edges, a trivial bound would be $EdgeCost \geq ||E_G| - |E_H||$. However, a tighter bound can be achieved by examining

the ordered outdegree or indegree sequences. The idea is that identical degree sequences are a requirement, but not sufficient, for two graphs to be isomorphic. Thus, the cost required to transform one sequence into the other is a lower bound for $EdgeCost$. With the distance between vertex degrees being $|d^+(u_i) - d^+(v_j)|$, $u_i \in V'_G$, $v_j \in V'_H$ (if we consider the outdegrees), the optimal solution can be retrieved using maximum weight bipartite matching which takes $O(n^3)$ time. Surprisingly, however, it can be done in linear time by computing $\sum_i^{|V'_G|} |d^+(u_i^{G'+}) - d^+(u_i^{H'+})|$, given $D^+_{G'}$ and $D^+_{H'}$. The correctness of the latter is proved below.

Lemma 1 *Given two sorted degree sequences, the cost of transforming one into the other can be computed in linear time, by* $\sum_i^{|V'_G|} |d^+(u_i^{G'+}) - d^+(u_i^{H'+})|$.

Proof It holds that $|a_1 - b_1| + |a_2 - b_2| \leq |a_1 - b_2| + |a_2 - b_1|$ with $0 \leq a_1 \leq a_2$, $0 \leq b_1 \leq b_2$ (trivial, the proof is omitted). So, if the optimal matching $h : D^+_{G'} \to D^+_{H'}$ would have matched $u_i^{G'+}$ to $u_k^{H'+}$ and $u_j^{G'+}$ to $u_l^{H'+}$, with $i < j$ and $k > l$, it would be possible to swap their matching counterparts and acquire a less or equal total score, because of the aforementioned inequality. However, since the matching is the optimal, the score should remain the same. This swapping operation can be repeated until $\forall i$, $u_i^{G'+}$ is matched to $u_i^{H'+}$. It remains to be proved that such a matching is indeed reachable by swaps from any possible optimal matching. A constructive proof exists: Given the optimal matching h, for each i in $1, \ldots, |D^+_{G'}|$, we iteratively swap the matching counterparts of $u_i^{G'+}$ and $h^{-1}(|u_i^{H'+}|)$. At step i, for each $1 \leq j \leq i$, the j-th element of the ordered degree sequence, $u_j^{G'+}$ is matched to $u_j^{H'+}$. □

The above hold for the indegree sequences, too. In practice, we have the capability to compute both, if necessary, and retain the one providing the highest score.

We combine all of the above inequalities to acquire a lower bound for Eq. 2, denote as GED_{LB}, in the formula depicted in Eq. 3. The total time complexity for computing the GED_{LB} of graphs G, H is $O(n)$, with $n = |V'_G|$. This is achieved by using Counting sort to sort them in linear time (Seward 1954). Sorting n non-negative integers whose maximum value is at most k, via Counting sort takes $O(n+k)$ time. Since k is a degree value, and $k \leq n$, given the degree sequences the whole process takes linear time.

$$GED_{LB}(G,H) = \frac{||V_G|-|V_H||+\max\{0,\max\{|S_G|,|S_H|\}-|S_G \cap S_H|-V_c\}+\sum_i^{|V'_G|}|d^+(u_i^{G'+})-d^+(u_i^{H'+})|}{|V_G|+|V_H|+|E_G|+|E_H|} \tag{3}$$

7 Clustering

In this Section we introduce the clustering algorithm that is in use by Classy. The proposed clustering algorithm is introduced in Sect. 7.1, while in Sect. 7.2 we describe how we can approximate the statistics of a cluster when there are available only a part of all possible pair-wised distances.

7.1 Clustering algorithm

Graph comparisons are expensive. We cannot afford to compute full distances matrices, nor compare an incoming sample with all the existing samples in the DB. Thus, it is imperative to keep the number of graph comparisons to a minimum while still acquiring the important information. Due to the cost of graph comparisons we cannot use NN-retrieval methods that would require comparing the query graph to a set of database reference samples (Burkhard and Keller 1973; Venkateswaran et al. 2006; Papapetrou et al. 2009). At this point we must make clear that improperly grouping two executables together is highly undesirable since it could result in a significant false alarm that would be propagated to the customers. So, while we aim for clustering together as many graphs as possible, we can tolerate improperly separated executables hoping that they will be detected by other systems.

As a first step, a procedure is needed to determine the possible clusters to which an incoming sample would be added. Graph invariants prove very useful for this problem. We can expect that pairs of graphs with a very large difference in the number of edges and vertices would give a high edit distance. Performing such a comparison would just verify that they should not belong to the same cluster. Conversely, it is graphs with similar numbers of edges and graphs that are useful in this context. So, for each incoming sample q, we query the database for clusters that contain graphs with similar graph order. The results of such query are the *candidate clusters*, $\mathcal{C}^q$, for that sample.

Given those clusters, the second step involves deciding which of the candidate clusters is the appropriate. Again, comparing the incoming sample to every sample of each cluster is not affordable, nor could the result justify such a cost. In most of the cases, comparing to a few samples provides all the necessary information. Based on that, we identify *reference samples*, r_i^c, for each cluster C in the database. These are the samples of the candidate cluster that would be compared to the incoming sample. There can be several strategies for selecting the reference samples: the first graphs seen by the system, the ones that are in the geometric center of the cluster (or their sub-clusters), the outliers of that cluster, the major malware variants for that family, etc.

In practice, we have found that selecting the first sample of the cluster to be its reference graph works rather well. The main reason is that given the first version of a piece of software, either benign or malicious, subsequent versions share with it a great amount of code and structure. So, we can expect the first graph of a cluster to be a central element. Furthermore, over the lifetime of the system, we can assume that the first version of a software would be the one to arrive first into our system. This is a valid assumption given the great number of sample exchange-feeds shared between AV companies and the file upstream functionality of modern AV clients; consequently the time it takes for a file that first appears in the wild to reach the AV company's backend is very short.

Our approach can be seen as belonging to the family of clustering algorithms where at first, one selects an arbitrary sample, takes all near-by elements to form a cluster and removes that cluster. Several such algorithms have been presented in the past and some of those have approximation guarantees under different objective functions. One example is `BALLS` studied in (Gionis et al. 2005). However, we leave it as future

work to investigate other methods for selecting the candidate samples and study the trade-offs in each case.

Based on the above, upon the arrival of a new sample, its candidate clusters are defined and then it is compared against $\mathcal{S}$, the set denoting the union of their reference samples. This stage is divided into two parts. The first comprises of using the aforementioned lower bound GED_{LB} on S to quickly prune many of the comparisons, which yields $\mathcal{P}^q$. SA is then used for comparing q to the graphs in $\mathcal{P}^q$ in order to identify the suitable candidate clusters for q. Suitable clusters are determined based on whether the incoming sample q and any of their reference samples are within $threshold$ distance. This is a common notion for density-based clustering algorithms (Ester et al. 1996). So instead of defining the number of clusters, which is often very hard to know when clustering streams, it is sufficient to define the distance threshold for two data elements to be considered similar. Clearly, the choice for the value of the threshold is often application dependent and may be determined empirically.

Several actions are taken depending on the outcome of determining suitable candidate clusters. If no suitable candidate cluster is found, the incoming sample forms a new one of its own. If only one suitable cluster is found, the incoming sample is assigned to that (the cluster is augmented with the new sample). If several suitable clusters have been identified, then those are merged into a single one and the set of reference samples is recalculated based on the used strategy. The rationale of merging those clusters together is the fact that the appearance of sample q has indicated the existence of an edit path involving their samples. This piece of information will be used by the analysts and other systems aimed at providing added-value. There is no concern that the samples will all be added to just a few clusters; the authors of (Kinable and Kostakis 2011) demonstrated that for reasonable parameter values, density based algorithms perform very well in terms of not lumping unrelated samples. Variations of the clustering algorithm itself include breaking on the comparisons as soon as a suitable candidate has been found. The way the lower bound is combined with the aforementioned clustering approach in Classy is described in Algorithm 2. The exact parameter values used for Classy are discussed in Sect. 8.1.

To facilitate the work of our human analysts, a final step involves finding isomorphisms between the incoming sample q and the samples in the cluster to which it was assigned. For that, it is enough to identify which ones have the same number of edges and vertices and determine how they are grouped in sets of isomorphic based on the comparisons of their own classifications. Then, it is simply an issue of comparing q to only one of the graphs of each existing set of isomorphic graphs.

7.2 Numerical approximations of cluster statistics

Both the automation systems and malware analysts require more information than simply the fact that several files are clustered together. In order to evaluate and make use of the clustering results, they require numerical information regarding each cluster. However, due to the aforementioned clustering algorithm an incoming sample is only compared against the cluster's reference graphs and those that might be isomorphic to it. As a result, for most of the possible pairs of graphs in a cluster the system has not

Algorithm 2 Classy's Clustering algorithm (described in Sect. 7.1)

Require: Incoming call-graph q, similarity threshold $threshold$

```
/* Get candidate clusters C^q and their reference samples r_i^C */
Get C^q from DB.
S = ⋃_{C∈C^q} r_i^C
if S == ∅ then
    return newCluster(q)
end if
/* Prune set using GED_LB */
P^q = ∅
for s ∈ S do
    if GED_LB(s, q) < threshold then
        P^q = P^q ∪ s
    end if
end for
if P^q == ∅ then
    return newCluster(q)
end if
/* Find suitable clusters by using SA to compute GED of q and reference samples */
Suitable = ∅
for p ∈ P^q do
    if SA(p, q) < threshold then
        Suitable=Suitable ∪p
    end if
end for
if Suitable== ∅ then
    return newCluster(q)
else if |Suitable| = 1 then
    c = s.cluster : s ∈ Suitable
    return addToExistingCluster(q, c)
else if |Suitable| > 1 then
    A = {s.cluster : ∀s ∈ Suitable}
    return mergeClustersAndAdd(q,A)
end if
```

executed any comparison. Computing the distance between all pairs of samples in a cluster can be performed on special occasions but it is not feasible for every cluster. Thus, we resort to approximating the distances.

Our approach relies on using the computed distances to infer a score for those distances that should be approximated. We assume that each call-graph is a point in a metric space, and the distances between pairs of call-graphs are either a known distance if that has been computed or 1.0 otherwise. An alternative view of this is a complete graph where the call-graphs are its nodes and their distances are its weighted edges. Then it is a matter of using an all-pairs shortest path algorithm to approximate the unknown distances. Such an algorithm is the Floyd–Warshall algorithm (1962, 1962).

The time complexity of the Floyd–Warshall algorithm is $\Theta(|V|^3)$. For large clusters with several thousand samples, such complexity renders the approach nonpractical. For that reason, we devised a speed-up pre-filtering step. The samples are bagged into sets of isomorphic graphs. Each set contains those samples for which there is a chain

of isomorphic relations. For example, given that we know that graph A is isomorphic to graph B, B to C and C to D, then A, B, C, and D would be assigned to the same set. Given n samples, m known pair isomorphisms (distances between samples with value equal to 0) and k the number of returned sets, the total complexity of approximating the distances becomes $O(n + \log^* m + k^3)$, when using union-find algorithms (Tarjan and Leeuwen 1984) to compute the sets.

Finally, the speed-up setting allows to efficiently compute the cluster statistics immediately instead of computing them over all n^2 distances. The cluster diameter is the maximum value of the k^2 distances, while the mean distance for that cluster can be computed by:

$$\frac{\sum_{i=1}^{k} \sum_{j=i+1}^{k} |k_i| \cdot |k_j| \cdot d(k_i, k_j)}{n \cdot (n-1)/2},$$

where k_i,k_j are the i-th and j-th sets respectively and $d(k_i, k_j)$ is the distance between them.

8 Experiments

In Sect. 8.1 we evaluate the performance of our system and provide relevant statistics, while in Sect. 8.2 we benchmark our proposed on-line clustering algorithm against the competitor which in our case is the offline optimal, that uses DBSCAN, described in Kinable and Kostakis (2011).

8.1 Live system performance and statistics

Results have been extracted from a live production-side system that has classified more than 0.8 million graphs. These were part of the raw stream of PE32 files that have arrived in the backend of F-Secure, and were not discarded by the simple filtering rules described in Sect. 4.2. We choose to limit the valid graphs to those that contain at least 20 vertices, but not more than 3000; no limit is imposed on the edges. Malware components tend to be small in size and compact, for a series of reasons. Flame/Flamer/Skywipper was a notable exception. We make the trade-off of focusing on graphs of such order range. We decided on the upper limit of 3000 after looking into tens of thousands of samples. For software larger in size, other approaches, such as md5 checksum verification, certificates and software signing are more appropriate for making sure they are not trojanized. The value of 3000 is a configuration value that can be changed at any time. Furthermore, as the order of call-graphs increases the witnessed call-graph space should become "sparser", so we can anticipate that the task of grouping becomes easier since GED_{LB} would be more effective.

The following results and statistics correspond to an instance of Classy with at most 48 workers. The majority of the system has been implemented using Python, apart from the SA method that was implemented in C. The relational database is PostgreSQL

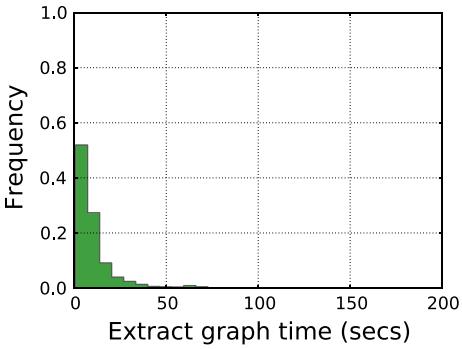


Fig. 5 Normalized histogram of the time required for extracting the call-graphs. The mean extraction time is 9 s

and the distributed file-system comprises of GlusterFS. More information such as parameter values may be found in the following sections.

8.1.1 Preprocessing and call-graph extraction.

First, we provide numbers related to the preprocessing phase. There is little room for improving the performance of the binary unpacking and graph extraction phase. However, from the system perspective it is an unavoidable cost. To gain an intuition about the required resources we investigate the time required for this phase. We chose to study the unpacking and extraction steps separately, since the extraction time depends to a great extent on the size of the graph and the size of the binary file, contrary to the unpacking which depends primarily on the technique used to pack the file. Figure 5 depicts the normalized histogram of the call-graph extraction phase. The original data values correspond to successfully extracted graphs and the chosen set of graphs is the 10454 graphs clustered in the last 24 h interval of the system. For the unpacking phase, we witness that the mean value is 0.6 s (graph omitted). The mean extraction time is 9 s while we notice in some cases values over a minute.

The percentage of the samples that are successfully preprocessed depend to a great extent on the distribution of each file type and the quality of the pre-filtering. In particular, cases such as Zip and RAR self-extracting executables should not be given as input to Classy, and their contents should be extracted separately. The same applies for known installers such as `InstallShield` and `Inno Setup`. Without counting such files, an average ratio of 70–75 % files are successfuly preprocessed. This includes potential unpacking and producing the call-graph.

8.1.2 Lower bounding the GED

The performance of the lower bounding method is crucial to the overall throughput of the system. This is due to the fact that GED_{LB} is responsible for reducing the number of graph comparisons the system will need to perform.

For that reason, we conduct a deeper investigation into its performance. We studied its *tightness* and its *pruning power* for 1-NN searches in offline experiments. The tightness of GED_{LB} is defined as the average ratio of $\frac{GED_{LB}(A,B)}{GED(A,B)}$ of a dataset, for

Table 1 Tightness and pruning power of the lower bound on the offline experiments

Dataset	Graphs	Graph order	Tightness	Pruning power
Pre-classified	194	20–1800	0.749	79.6
Random	2867	50–500	0.885	98.6

graphs A and B. The tightness value is in [0, 1]. Values closer to 1 denote a tighter lower bound which in turn is more useful. The pruning power is defined as the number of comparisons using a distance function d that are pruned from a brute-force 1-NN search when its lower bound d_{LB} is first computed.

In practice, we have no exact algorithm, that is fast enough, to compute the true value of function GED in each instance. Instead, we compute the ratio of the lower bound over the distance returned by the Simulated Annealing algorithm. Thus, not only is the true tightness of the LB higher than the one witnessed, we also acquire an insight on how tight both SA and the lower bound turn out to be, since SA over-approximates GED.

For the offline experiments, we chose 2 rather small data-sets. The first, is a selection of 194 samples hand-picked by human analysts and manually pre-classified into malware families. The second, is a collection of 2,867 samples that arrived in F-Secure's backend systems in two consecutive days. For those days, we kept only the samples whose graphs contain 50–500 vertices. In Table 1 we present the dataset descriptions and the related results. As expected, we observe that the presence of clusters of similar files in the pre-classified set result into a lower tightness and weaker pruning power, compared to the randomly selected graphs. It is interesting to note the tightness of GED_{LB} over SA. For the random set of graphs we witnessed a 0.885 tightness value resulting in a 98.6 % pruning power. Such a high tightness value shows that both GED_{LB} and SA are tight. For the dataset containing hand-picked samples belonging to a restricted set of malware families, we notice that the tightness and prunning power are lower. This provides the insight that GED_{LB} suffices for the case where the compared samples are very different, but for more difficult cases it is imperative to use a more expensive comparison algorithm. Finally, the run-time for computing the GED_{LB} we had witnessed in our offline experiments was approximately 0.15 s for graphs of order 500 while for graphs of order 3000 it was less than 1 s.

In the live system, the performance of the GED_{LB} is measured by the ratio of comparisons that are pruned from the combination of the incoming sample and the reference samples of its candidate clusters. Given an incoming sample q with graph $G_q = (V_q, E_q)$, The candidate clusters $\mathcal{C}^q$ are selected by querying for clusters whose reference samples contain $a \cdot |V_q|$ vertices, with $a \in [0.95, 1.05]$. The threshold for which candidate comparisons are pruned was set to 0.1 (there is no ground truth for determining this value, it was chosen empirically after consulting with the malware analysts). Naturally, this is also the similarity threshold for our clustering algorithm. For the last 98806 successfully extracted graphs, the total sum of candidate comparisons was 105374736, while after the pruning phase remained 2383025, resulting in a 97.7 % pruning power. Such a speedup proves significant especially in the case of samples that are not similar to any existing sample and form a cluster of their own. The GED_{LB} is the component that makes the whole task of clustering streams of

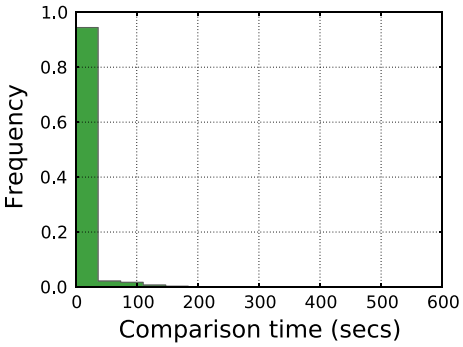


Fig. 6 Normalized histogram of the total time consumed in the comparison stage for each graph. The mean comparison time is 23 s. We have witnessed 0.6 % of the graphs requiring more than 200 s

call-graphs feasible, since in average a set of 1066 comparisons are reduced to 24 by performing very cheap operations.

Suppose we had selected indexing schemes that rely on embeddings such as (Venkateswaran et al. 2006; Papapetrou et al. 2009) and had selected an embedding set with cardinality of 24; a low number for the graph cardinality range that we study. Then, immediately at the first stage of the process (comparing the query call-graph with the reference objects to determine the suitable sub-space) the number of expensive graph comparisons would exceed that required by the average case of our approach. Similarly, SMIT (Hu et al. 2009) requires, based on the authors' claims, on average 112 graph comparisons (median is 78, maximum is 918) for 5-NN queries over a dataset that contains only 102391 samples; significantly fewer indexed samples than in our database.

8.1.3 Graph comparisons using simulated annealing

The most important part of the system is the SA graph comparison algorithm. We omit its comparison with existing bipartite matching-based methods in the literature, since this aspect has been the main subject of Kostakis et al. (2011). In our Classy instance, SA is used with parameter values $\beta_0 = 4.0$ and $\epsilon = 0.9$. In Fig. 6 we present a normalized histogram for the time required in the comparison phase for the samples clustered during the last 24 h. The mean comparison time is 23 s, however we do witness 0.6 % of samples requiring more than 200 s. We must make clear that each sample is not necessarily compared against every sample in its pruned candidates' list. Instead, we take advantage of the fact that the lower bounds have already been computed. So, the list is sorted in ascending order of the LB scores and the comparisons stop as soon as a suitable candidate has been found. This trade-off offers a significant speedup and allows to maximize the system's throughput. Despite this trade-off, two isomorphic samples would still be assigned to the same cluster since their candidate list would be the same and thus the comparisons would happen in the same order with (approximately) the same results. The average ratio of comparisons avoided only in this stage is 84.42 %.

The last phase of the whole process is to identify isomorphic graphs within the cluster(s) which the incoming sample has been assigned to. This step is performed if

and only if the incoming samples has not been found to be isomorphic to any other sample it had been compared against, during the clustering phase. For the last 81064 graphs reaching this phase, 34047 graphs had already been identified as isomorphic to an existing database graph, while for the rest 47017, 184710 comparisons were performed. This is an average 3.9 additional comparisons for the samples that had not been identified as isomorphic to any database graphs in the comparison stage, and a system average of 2.2 comparisons per clustered sample.

Finally, we compute the count of identified sets of isomorphic graphs in each cluster. This provides an intuition about a combination of things: the power of the SA comparison algorithm, the clustering quality cost induced by applying the aforementioned trade-off in the comparison stage and more directly, the actual speedup in computing the cluster statistics (presented in Sect. 7.2). For all clusters containing more than 20 graphs the mean ratio of the number of sets of isomorphic graphs (k) over the total number of samples in a cluster is 0.132. There is no clear indication, by only looking at the number of isomorphic sets and/or samples in a cluster, whether the samples are benign or malicious.

8.1.4 Scalability

The fact that our system follows a double producer-consumer design pattern allows us to add more resources of each type. In particular, we are able to freely add more masters or slaves if that becomes necessary. Clearly, more workers allow for greater system throughput. On a typical day, we observe a per-worker average of 240 samples clustered in 24 h, each one using its own dedicated CPU core. The statistics presented so far are drawn from our instance of Classy that has up to 48 workers, hence running on 48 CPU cores. Hence, Classy is able to cluster more than 11000 call-graphs per day.

The throughput of the system appears to scale linearly with the number of available workers. Since the workers only access the distributed file-system for storing the call-graphs that should scale to handle the added workers. Regarding the file-system, one must make a trade-off between storage space and latency. In particular, Network File System (NFS) can be used for small to medium clusters and for data that fit in a RAID configuration. For larger clusters and graph repositories, more scalable filesystems are needed; Classy uses GlusterFS. In practice, we have observed that a dual-layer (in-memory and local disk) caching under an LRU scheme works very well to bridge the latency gap. Since the call-graphs are never modified, there is no concern for maintaining consistency of data among different caches and across different servers.

Regarding the masters, the number of instances does not significantly affect the system performance. A single master process can support up to 48 workers without a humanly observable delay. However, we would suggest the presence of at least two master processes. This is to avoid having a single point of failure, to minimize the wait time of tasks in the masters' queue, and to minimize the delay incurred by potentially slow queries to the relational database. The database that is accessed by the masters is used only for maintaining the information about the samples and the clusters. Hence, Classy is far from saturating the scalability capabilities of that.

8.1.5 Application: malware clustering and detection

Classy can be used for clustering graphs related to any application. The original motivation was that of clustering executable programs for facilitating the work of security response teams in data security companies. Below we provide information related to the application of large-scale malware clustering and provide some insights.

The feature that clearly separates Classy from other automation systems is the ability to process and group DLL (dynamic-link library) files. In the typical case, DLL files are imported by a host executable file and they cannot be executed in standalone mode. Hence behavior-based analysis systems, such as Hegedus et al. (2011), Rieck et al. (2008), Bayer et al. (2009), are incapable or inefficient, since this imposes the requirement to have acquired both the host executable and the DLL files during the time of execution. Instead, Classy can process and cluster together similar DLL files independently of their launching executable. Modern malware make extensive use of DLL files; reasons for this include software modularity and code injection capabilities. Furthermore, DLL files are susceptible to most of the aforementioned obfuscation techniques. The amount of DLL files clustered by Classy corresponds to 13.2 % of the total files, and 18.8 % of those are malicious. The number of malicious DLLs in the wild is in the order of tens of millions. So, this is a clear indicator of part of the added value Classy may provide for the application of malware clustering.

Apart from knowledge extraction capabilities aimed at human analysts, clustering data objects allows the task of supervised classification. While the classification results of offline clustering of call-graphs (Kinable and Kostakis 2011) carry over to the stream setting (as we demonstrate in Sect. 8.2), we are able to provide results for a larger set of files. The first important insight is that, in the general case, automatically classifying files as malware should not be performed by default by taking into account all the samples' labels in a cluster. This is due to the fact that malware is often masqueraded to appear as legitimate software (by *trojanizing* them); a legitimate and a trojanized version of the same software would get clustered together by SA but they would not be isomorphic. Similarly, benign and malware could share great portions of common functionality. Hence, this could lead to severe false alarms. Instead, unless it is possible to use additional information or an analyst's intervention, automated classifications should be restricted to only groups of isomorphic graphs. This does not mean that the clustering task should be restricted only to identifying isomorphisms, since, as discussed above, grouping similar but not necessarily isomorphic call-graphs is a highly valuable tool for the whole stack of a security response team.

We proceed to evaluate the clustering results. To be sure that a cluster has enough samples to make a confident decision, we investigate those clusters that contain more than 20 graphs. In total, they consist of 74.3 % of all clustered files. Table 2 depicts the percentage of clusters that contain only malware, only benign files or both. The percentage of respecitive samples are in brackets. We witness that only 5.4 % of clusters contain both clean and malware samples, affecting 12.42 % of the samples (larger clusters are expected to contain misclassifications produced by human analysts or imprecise pattern based detections). Clearly, by decreasing the distance threshold used in clustering, the value of clusters containing both malware and clean files should decrease. Table 3 depicts the percentage of samples that belong to groups of isomor-

Table 2 Percentage of clusters that contain only malware, clean files or both; in brackets the percentage of corresponding samples. The numbers correspond to the samples in the clusters with more than 20 graphs

		Contains clean files	
		Yes	No
Cluster contains malware	Yes	5.4 (12.42) %	12.5 (10.2) %
	No	82.1 (77.38) %	–

Table 3 Percentage of samples belonging to groups of isomorphic graphs that contain only malware, clean files or both. The numbers correspond to the samples in the clusters with more than 20 graphs

		Contains clean files	
		Yes	No
Group contains malware	Yes	2.9 %	16.3 %
	No	80.8 %	–

phic call-graphs that contain only malware, only benign files or both. So, the groups containing only malware correspond to 16.3 % of the total samples. The number of samples present in sets containing both benign and malware is equal to 2.9 % of the total samples. This is the source of the false-positives. The strategy by which the labels are assigned to new files would determine the amount of experienced false-positives or missed files. At this stage, assigning the label 'malware' preferentially, the ratio of false-positive files is 1.2 % over all samples. With information from other means and systems, we can identify which of those call-graphs correspond to installers, custom packers etc, dropping the actual false-positive rate to 0.5 %. In practice, the results of any automated malware classifier are passed onto a rule-based system. Through that, we provide all involved parties with the capability to write sophisticated rules on top of the clustering results and to combine file metadata and information from other sources. As a result, this allows to reduce the false positives that affect the end-users while also being able to increase the coverage of automated malware classification beyond isomorphism relations on a per-case basis. Overall, the combination of Classy and a rule-based decision system enables not only a more powerfull automated classification mechanism but also the basis for other knowledge extraction and recommender systems.

The structural similarity aspect is a big advantage for Classy over backend run-time behavior analysis systems in the cases of malware that rely on *downloader* components to fetch the actual malicious payload from the Internet. In an isolated execution environment, it is impossible for the payload to be downloaded and consequently any relevant behavior to be expressed. This problem is usually circumvented by implementing a host-based intrusion prevention system on the user's machine. We witness that with Classy, we were able to detect different instances of similar downloading components in the backend, by clustering them together; including variants of downloader components of `ZeroAccess`. This is also beneficial since it allows to handle right away the downloading component instead of the downloaded files.

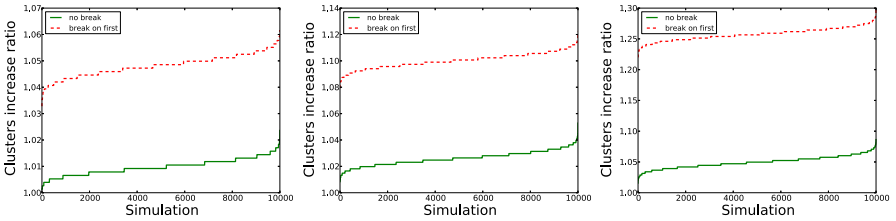
Additional interesting cases where Classy proved superior to other systems comprise of instances of files that, upon execution, write to disk a modified version of themselves (with some element of randomness). In a sandboxed execution setting, the newly dropped file would have to be analyzed too via execution that would in turn produce yet another modified version and so on, thus resulting in an endless inheritance path. With Classy, we are able to cluster the original file immediately, and also verify that a potential new file is similar to the original. We have observed such cases for both benign and malicious software.

As discussed previously, one of the limitations of Classy concerns unknown packers and the case where the extracted call-graphs correspond to the common unpacking module. This is the source of some of the false alarms. However, we witness that in certain occasions this can be an advantage, since certain packers are used exclusively by malware authors. Hence, while new malware might exhibit behavior never seen before, and thus go undetected by run-time analysis systems, the produced call-graphs would be similar to previously known. By using the clustering results by Classy, the malware analysts are enabled to write rules that capture and assign guilty-by-association verdicts to previously unknown malware. Example cases of malware that share common packers include variants of the `W32/Carberp` bootkit and `W32/Lockscreen` ransomware. Furthermore, in some of those cases, to avoid pattern-based detection of the common unpacking module that remains visible, the authors modify them manually, or the packer self-mutates; the latter is known widely as *polymorphic malware* (Xu et al. 2004) and pose an important challenge in malware detection. Even in such cases, Classy is expected to be able to group them together, provided that the call-graph representation is not severely altered.

Similarly to packers, other problematic cases that we have observed include files that enclose interpreters. Such examples include `AutoIt` files, where custom bytecode is produced for execution by the interpreter. As a result, any call-graphs extracted via static analysis fail to encapsulate the actual functionality that is performed through the interpreter. Under these circumstances, two options are available. The first option is to extend the file-filtering rules and avoid clustering them altogether, or to gray-list the file type and avoid making automated decisions for those. The second option is to improve the call-graph extraction procedure in a way that can trace the functionality of the interpreter. As discussed in Sect. 3.1, this entails additional technical challenges and requirements, but our clustering framework would be capable to handle the data.

8.2 Comparison to offline optimal clustering

The statistics from the live system prove that the used techniques provide very good speedups. Still, we need to gain an insight on the quality of the clustering. At this point we investigate the performance of our stream clustering algorithm versus the optimal results, or the ground truth, as that is provided by an offline algorithm. The baseline method in this case is the DBSCAN method (Ester et al. 1996), used by the authors in Kinable and Kostakis (2011), and is applied on whole distance matrices. So, for this experiment we produce the distance matrix for the Random dataset of Table 1, and we apply DBSCAN to obtain the optimal clustering; optimal in the



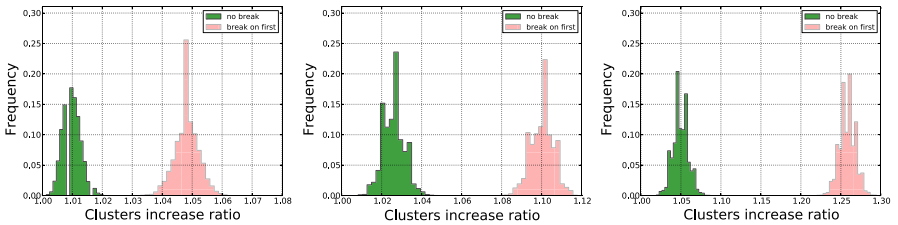
(a) $eps = 0.1$ **(b)** $eps = 0.2$ **(c)** $eps = 0.3$

Fig. 7 Sorted distribution of the increase in the number of clusters

sense that it is computed offline with no performance trade-offs. We then proceed to compute the clustering produced by our stream clustering algorithm. Comparing the two algorithms is meaningful only if the same parameters are used. In particular, the threshold parameter of our algorithm is equivalent to the `eps` parameter of DBSCAN. Since in our clustering algorithm a single sample can form a cluster of its own, for DBSCAN this translates to setting `minpts` to 1. The algorithm we propose will never produce less clusters than DBSCAN. It is trivial to prove that if two samples would be clustered together in the stream setting, the conditions for doing so would suffice for DBSCAN to cluster them together, too.

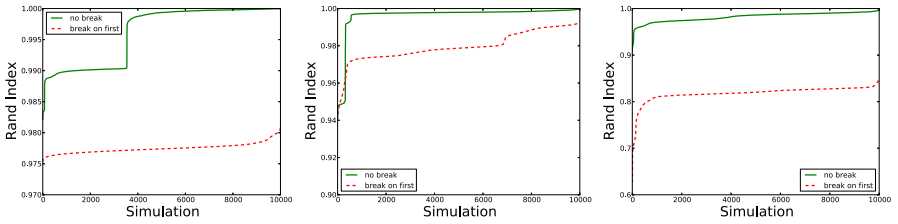
We benchmark the stream clustering algorithm in terms of the number of clusters produced, compared to the optimal, and we also compute the Adjusted Rand Index (Hubert and Arabie 1985). The number of clusters can be misleading since it is not clear how the samples are distributed among the clusters. It might be that the extra clusters consist of individual samples or that a real cluster is split into two equal parts. For this reason we believe the Adjusted Rand Index is a more accurate indication of clustering quality. The Adjusted Rand Index compares two clusterings and provides a measure of agreement. It takes values between -1 and 1, with 1 indicating that the two clusterings are identical. Due to the fact that in the stream algorithm there is a strong element of randomness, i.e. the order in which the samples arrive through the stream, we perform a significant number of simulations. In particular, we perform 10000 simulations with a random permutation of the stream in each case. We benchmark the algorithms with values 0.1, 0.2 and 0.3 for the threshold parameter and `eps`. While 0.1 is the one used in Classy, we investigate the behavior under other parameter values in order to get a strong insight on the algorithm. We also simulate the case where the streaming algorithm breaks as soon as a first candidate sample has been found.

Figure 7a–c depict the sorted distribution of the increase in the number of produced clusters. Figure 8a–c depict the histograms of the those distributions. We notice, in all cases of Fig. 8, that the distributions resemble a Gaussian-like bell curve. As expected, their curve centers deviate from the optimal solution as the value of `eps` increases. The deviation remains limited, even for `eps`$= 0.3$, for the case of computing the distance with the reference samples of all candidate clusters. Essentially, when the comparison breaks on the first suitable candidate, no cluster merges happen, so the number of produced clusters is greater. Figure 9a–c depict the sorted distributions of the Adjusted Rand index for all `eps` values. We witness that for the vast majority of simulations,the value of the Adjusted Rand index remains above 0.975 for both



(a) $eps = 0.1$ **(b)** $eps = 0.2$ **(c)** $eps = 0.3$

Fig. 8 Histogram of the increase in the number of clusters produced by our stream clustering algorithm versus offline DBSCAN



(a) $eps = 0.1$ **(b)** $eps = 0.2$ **(c)** $eps = 0.3$

Fig. 9 Sorted distribution of the values of the adjusted rand index between our stream clustering algorithm and offline DBSCAN

scenarios and eps= 0.1, 0.2. For eps= 0.3, when the clustering does not break once a suitable candidate has been found, the Index remains above 0.95 for almost all permutations. Finally, the value of the Adjusted Rand Index deteriorates for eps= 0.3 when the algorithm does not merge clusters.

9 Conclusions

To handle the ever increasing number of binary executable files received by anti-virus companies every day, the design and use of system automation is required. Furthermore, instruction-level obfuscation techniques are used by malware authors in an attempt to prevent the detection of their malware by pattern-based techniques. An abstraction resilient to many obfuscation methods is the call-graph; the representation of a binary executable file as a directed graph. Unfortunately, interesting graph comparison problems such as full-graph comparison belong to the class of NP-hard problems.

We presented Classy, a distributed system that clusters streams of large graphs. The graph comparison measure is the GED and it is approximated using an adapted version of Simulated Annealing. In addition, we presented a linear-time lower bound for the GED that proved very tight and with great pruning power. We proposed a suitable clustering algorithm that demonstrates a performance not far from the offline optimal. Finally, we studied the problem of approximating the statistics of clusters for extracting cluster summaries when only part of all distances are available. Experiments were conducted on the proposed algorithms and statistics were provided of a system

instance that has clustered and contains over 0.8 million graphs. The results were highly promising. Overall, we demonstrated how a set of seemingly simple algorithms can be assembled together in order to build a system that adequately approximates instances of NP-hard problems tens of thousands of times per day for delicate tasks such as automated malware classification.

Acknowledgments This work was supported by TEKES as part of the Future Internet Programme of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT). Special thanks to Paolo Palumbo for providing the file filtering rules, Gergely Erdélyi for his support on IDA Python and the call-graph unpacking code, and Stefan Lundström for the early integration of the system with the backend APIs.

References

Aggarwal CC, Han J, Wang J, Yu PS (2003) A framework for clustering evolving data streams. In: Proceedings of the 29th international conference on very large data bases-volume 29, VLDB Endowment, pp 81–92

Aggarwal CC, Han J, Wang J, Yu PS (2004) A framework for projected clustering of high dimensional data streams. In: Proceedings of the thirtieth international conference on very large data bases-volume 30, VLDB Endowment, pp 852–863

Aggarwal C, Zhao Y, Yu P (2010) On clustering graph streams. In: Proceedings of the SIAM international conference on data mining, pp 478–489

Akutsu T (1993) A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. IEICE Trans Fundam Electron Commun Comput Sci 76(9):1488–1493

Bayer U, Comparetti PM, Hlauschek C, Kruegel C, Kirda E (2009) Scalable, behavior-based malware clustering. In: 16th Network & distributed system security conference, vol 9, pp 8–11

Bourquin M, King A, Robbins E (2013) Binslayer: accurate comparison of binary executables. In: Proceedings of the 2nd ACM SIGPLAN program protection and reverse engineering workshop, ACM, p 4

Briones I, Gomez A (2008) Graphs, entropy and grid computing: automatic comparison of malware. Proceedings of the virus bulletin conference, pp 1–12

Bunke H (1997) On a relation between graph edit distance and maximum common subgraph. Pattern Recognit Lett 18(8):689–694

Burkhard W, Keller R (1973) Some approaches to best-match file searching. Commun ACM 16(4):230–236

Cao F, Ester M, Qian W, Zhou A (2006) Density-based clustering over an evolving data stream with noise. In: Proceedings of the SIAM international conference on data mining, pp 328–339

Carrera E, Erdélyi G (2004) Digital genome mapping-advanced binary malware analysis. In: Proceedings of the virus bulletin conference, pp 187–197

Charikar M, O'Callaghan L, Panigrahy R (2003) Better streaming algorithms for clustering problems. In: Proceedings of the ACM symposium on theory of computing, ACM, pp 30–39

Cheng J, Ke Y, Ng W (2009) Efficient query processing on graph databases. ACM Trans Database Syst (TODS) 34(1):2

Christodorescu M, Jha S (2004) Testing malware detectors. ACM SIGSOFT Softw Eng Notes 29(4):34–44

Conte D, Foggia P, Sansone C, Vento M (2004) Thirty years of graph matching in pattern recognition. Int J Pattern Recognit Artif Intell 18(03):265–298

Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on computational geometry, ACM, pp 253–262

Dean T, Boddy M (1988) An analysis of time-dependent planning. In: Proceedings of the 17th national conference on artificial intelligence, pp 49–54

Dullien T, Rolles R (2005) Graph-based comparison of executable objects. SSTIC 5:1–3

Elhadi AAE, Maarof MA, Barry BI (2013) Improving the detection of malware behaviour using simplified data dependent api call graph. Int J Secur Appl 7(5):29–42

Ester M, Kriegel HP, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. KDD 96:226–231

Flake H (2004) Structural comparison of executable objects. In: Proceedings of the international GI workshop on detection of intrusions and malware & vulnerability assessment, pp 161–174
Floyd R (1962) Algorithm 97: shortest path. Commun ACM 5(6):345
Gascon H, Yamaguchi F, Arp D, Rieck K (2013) Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on artificial intelligence and security, ACM, pp 45–54
Gionis A, Indyk P, Motwani R et al (1999) Similarity search in high dimensions via hashing. VLDB 99:518–529
Gionis A, Mannila H, Tsaparas P (2005) Clustering aggregation. In: Proceedings of the 21st international conference on data engineering (ICDE), IEEE, pp 341–352
Giugno R, Shasha D (2002) Graphgrep: a fast and universal method for querying graphs. In: Proceedings of the 16th international conference on pattern recognition, IEEE, vol 2, pp 112–115
Guha S, Meyerson A, Mishra N, Motwani R, O'Callaghan L (2003) Clustering data streams: theory and practice. IEEE Trans Knowl Data Eng 15(3):515–528
He H, Singh A (2006) Closure-tree: an index structure for graph queries. In: Proceedings of the 22nd international conference on data engineering, IEEE, pp 38–38
Hegedus J, Miche Y, Ilin A, Lendasse A (2011) Methodology for behavioral-based malware analysis and detection using random projections and k-nearest neighbors classifiers. In: Seventh international conference on computational intelligence and security (CIS), IEEE, pp 1016–1023
Hex-Rays (2008) Ida pro. http://www.hex-rays.com/
Hu X, Chiueh T, Shin K (2009) Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM conference on computer and communications security, ACM, pp 611–620
Hubert L, Arabie P (1985) Comparing partitions. J Classif 2(1):193–218
Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, ACM, pp 604–613
Jiang H, Wang H, Yu P, Zhou S (2007) Gstring: a novel approach for efficient search in graph databases. In: Proceedings of the IEEE 23rd international conference on data engineering, IEEE, pp 566–575
Kang MG, Poosankam P, Yin H (2007) Renovo: a hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on recurring malcode, ACM, pp 46–53
Kinable J, Kostakis O (2011) Malware classification based on call graph clustering. J Comput Virol 7(4):233–245
Kolbitsch C, Comparetti PM, Kruegel C, Kirda E, Zhou Xy, Wang X (2009) Effective and efficient malware detection at the end host. In: USENIX security symposium, pp 351–366
Kollios G, Potamias M, Terzi E (2013) Clustering large probabilistic graphs. IEEE Trans Knowl Data Eng 25(2):325–336
Kostakis O, Kinable J, Mahmoudi H, Mustonen K (2011) Improved call graph comparison using simulated annealing. In: Proceedings of the 2011 ACM symposium on applied computing, ACM, pp 1516–1523
Kriege N, Mutzel P (2012) Subgraph matching kernels for attributed graphs. arXiv preprint arXiv:1206.6483
Kulis B, Basu S, Dhillon I, Mooney R (2009) Semi-supervised graph clustering: a kernel approach. Mach Learn 74(1):1–22
Lin IJ, Kung SY (1997) Coding and comparison of dag's as a novel neural structure with applications to on-line handwriting recognition. IEEE Trans Signal Process 45(11):2701–2708
Martignoni L, Christodorescu M, Jha S (2007) Omniunpack: fast, generic, and safe unpacking of malware. In: Twenty-third annual computer security applications conference (ACSAC) 2007, IEEE, pp 431–441
Mishra N, Schreiber R, Stanton I, Tarjan RE (2007) Clustering social networks. In: Algorithms and models for the web-graph. Springer, Berlin, pp 56–67
Moser A, Kruegel C, Kirda E (2007a) Exploring multiple execution paths for malware analysis. In: IEEE symposium on security and privacy, IEEE, pp 231–245
Moser A, Kruegel C, Kirda E, (2007b) Limits of static analysis for malware detection. In: Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual, IEEE, pp 421–430
Papapetrou P, Athitsos V, Kollios G, Gunopulos D (2009) Reference-based alignment in large sequence databases. Proc VLDB Endow 2(1):205–216
Ramon J, Gärtner T (2003) Expressivity versus efficiency of graph kernels. First international workshop on mining graphs, trees and sequences, pp 65–74
Rieck K, Holz T, Willems C, Düssel P, Laskov P (2008) Learning and classification of malware behavior. In: Detection of intrusions and malware, and vulnerability assessment. Springer, Berlin, pp 108–125

Riesen K, Bunke H (2009) Approximate graph edit distance computation by means of bipartite graph matching. Image Vis Comput 27(7):950–959
Ryder BG (1979) Constructing the call graph of a program. IEEE Trans Softw Eng 3:216–226
Schaeffer S (2007) Graph clustering. Comput Sci Rev 1(1):27–64
Schietgat L, Ramon J, Bruynooghe M (2013) A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. Ann Math Artif Intell 69(4):343–376
Seward HH (1954) Information sorting in the application of electronic digital computers to business operations. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology
Shervashidze N, Schweitzer P, Van Leeuwen EJ, Mehlhorn K, Borgwardt KM (2011) Weisfeiler–Lehman graph kernels. J Mach Learn Res 12:2539–2561
Snaker, Qwerton, Jibz (2006) Peid. http://www.aldeid.com/wiki/PEiD
Tarjan R, Van Leeuwen J (1984) Worst-case analysis of set union algorithms. J ACM 31(2):245–281
Tian Y, Patel J (2008) Tale: A tool for approximate large graph matching. In: Proceedings of the IEEE 24th international conference on data engineering, IEEE, pp 963–972
Veeramani R, Rai N (2012) Windows api based malware detection and framework analysis. In: International conference on networks and cyber security, p 25
Venkateswaran J, Lachwani D, Kahveci T, Jermaine C (2006) Reference-based indexing of sequence databases. In: Proceedings of the 32nd international conference on very large data bases, VLDB Endowment, pp 906–917
Vishwanathan S, Schraudolph NN, Kondor R, Borgwardt KM (2010) Graph kernels. J Mach Learn Res 11:1201–1242
Warshall S (1962) A theorem on Boolean matrices. J ACM 9(1):11–12
Willems C, Holz T, Freiling F (2007) Toward automated dynamic malware analysis using cwsandbox. Proceedings of the 28th IEEE symposium on security and privacy, vol 5(2), pp 32–39
Williams D, Huan J, Wang W (2007) Graph database indexing using structured graph decomposition. In: Proceedings of the IEEE 23rd international conference on data engineering, IEEE, pp 976–985
Xu JY, Sung AH, Chavez P, Mukkamala S (2004) Polymorphic malicious executable scanner by api sequence analysis. In: Fourth international conference on hybrid intelligent systems, HIS'04., IEEE, pp 378–383
Xu M, Wu L, Qi S, Xu J, Zhang H, Ren Y, Zheng N (2013) A similarity metric method of obfuscated malware using function-call graph. J Comput Virol Hacking Tech 9(1):35–47
Yan X, Yu P, Han J (2005) Substructure similarity search in graph databases. In: Proceedings of the 2005 ACM SIGMOD international conference on management of data, ACM, pp 766–777
Zeng Z, Tung A, Wang J, Feng J, Zhou L (2009) Comparing stars: on approximating graph edit distance. Proc VLDB Endow 2(1):25–36
Zhao P, Yu J, Yu P (2007) Graph indexing: tree+ delta≤ graph. In: Proceedings of the 33rd international conference on very large data bases, VLDB Endowment, pp 938–949
Zhou Y, Cheng H, Yu JX (2009) Graph clustering based on structural/attribute similarities. Proc VLDB Endow 2(1):718–729