Improved Call Graph Comparison Using Simulated Annealing

Orestis Kostakis*
Department of Information and Computer Science,
Aalto University, Finland
Helsinki Institute for Information Technology
orestis.kostakis@tkk.fi

Joris Kinable*
Department of Information and Computer Science,
Aalto University, Finland
Helsinki Institute for Information Technology
j.kinable@gmail.com

Hamed Mahmoudi†
Department of Information and Computer Science,
Aalto University, Finland
hamed.mahmoudi@tkk.fi

Kimmo Mustonen*
F-Secure Corporation
Helsinki, Finland
kimmo.mustonen@f-secure.com

ABSTRACT

The amount of suspicious binary executables submitted to Anti-Virus (AV) companies are in the order of tens of thousands per day. Current hash-based signature methods are easy to deceive and are inefficient for identifying known malware that have undergone minor changes. Examining malware executables using their call graphs view is a suitable approach for overcoming the weaknesses of hash-based signatures. Unfortunately, many operations on graphs are of high computational complexity. One of these is the Graph Edit Distance (GED) between pairs of graphs, which seems a natural choice for static comparison of malware. We demonstrate how Simulated Annealing can be used to approximate the graph edit distance of call graphs, while outperforming previous approaches both in execution time and solution quality. Additionally, we experiment with opcode mnemonic vectors to reduce the problem size and examine how Simulated Annealing is affected.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g.,viruses, worms, Trojan horses)*

General Terms

Security

*This work has been supported by TEKES - the Finnish Funding Agency for Technology and Innovation as part of its ICT SHOK Future Internet research programme, grant 40212/09.

†Supported by the Academy of Finland as part of its Finland Distinguished Professor programme, project 129024/Aurell.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

Keywords

Call graph, Malware, Graph Edit Distance, Simulated Annealing, static analysis

1. INTRODUCTION

On a daily basis, over tens of thousands of samples with potentially harmful executable code are submitted for analysis to Anti-Virus companies. In order to protect against malware threats, a total of more than 5 million new malicious code signatures were added to a single company's signature database in 2009 [9]. To deal with these vast amounts of malware, autonomous and automated systems for detection, recognition and classification are required. Furthermore, cyber criminals constantly develop new versions of their malicious software to evade pattern-based detection by AV products. The most significant weakness of the current hash signature-based detection approach is that even minor changes in the body of the executable would modify its hash signature thus making it undetectable by the AV-products [5]. The ability to recognize malware families and in particular the common components responsible for the malicious behaviour of the executables within a family would allow AV products to pro actively detect both known samples as well as future releases of samples belonging to the same malware family.

To facilitate the recognition of highly similar executables or commonalities among multiple executables which have been subject to modification, a high-level structure, i.e. an abstraction, of the samples is required. One such abstraction is the *call graph* which is a graphical representation of a binary executable, where functions are modelled as vertices and calls between those functions as directed edges. Minor changes in the body of the code are not reflected in the structure of the graph.

Our work falls under the category of static analysis for detecting malware. Unlike dynamic analysis [1], which concentrates on comparing runtime behaviour of malware, we examine similarity of malware based on their call-graph structure. So far only a limited amount of research has been published on automated malware identification and classification through call graphs [8, 6, 4, 3, 13]. Most of the methods proposed so far were highly based on heuristic or approxima-

tion techniques. Formulating the graph edit distance as an objective function allows to deploy Simulated annealing to find the optimal solution. We demonstrate how an adapted version of Simulated annealing yields better results and performs faster than the approaches presented previously in [13, 23]. We also experiment with opcode mnemonics in an attempt to further improve both the accuracy and runtime of Simulated annealing by reducing the problem size.

The rest of the paper is organized as follows: Section 2 surveys related work. Section 3 provides the necessary background and Section 4 contains a description of the methods that appear in this paper. In Section 5 we demonstrate experimental results and report on the findings. Finally, in Section 6 we conclude and propose directions for future work.

2. RELATED WORK

Flake's [8] seminal work followed by that jointly with Rolles [6] describe approaches to find isomorphisms between call graphs by mapping functions according to the similarity of their control flow graphs. Functions which could not be reliably mapped from one call graph to the other, have been subject to change. Via this approach, it is possible to reveal differences between versions of the same executable or detect code theft. Additionally, it is suggested that security experts could save valuable time by only analysing the differences among variants of the same malware.

Previous work on call graphs, specifically in the context of malware analysis, has been performed by Carrera and Erdélyi [4]. To speed up the process of malware analysis, call graphs are used to reveal similarities among multiple malware samples. Furthermore, after deriving similarity metrics to compare call graphs mutually, the metrics are applied to create a small malware taxonomy using a hierarchical clustering algorithm. Briones and Gomez [3] focus on the design of a distributed system to compare, analyse and store call graphs for automated malware classification. The first large scale experiments on malware comparisons using real malware samples were recently published [13]. Additionally, techniques are described for efficient indexing of call graphs in hierarchical databases to support fast malware lookups and comparisons.

Graph Edit Distance is a very fundamental problem with application in many fields [10]. Since exact solutions for GED are still computationally expensive to calculate, a large amount of research is devoted to approximation algorithms. A survey of three different approaches to perform GED calculations is conducted by Neuhaus, Riesen, et al. in [18, 19, 20]. They first give an exact GED algorithm using A* search algorithm, but this algorithm is only suitable for small graph instances [18]. Next, A*-Beamsearch, a variant of A* search which prunes the search tree more rigidly, is tested. The last algorithm they survey uses Munkre's bipartite graph matching algorithm as an underlying scheme. This approach, compared to the A*-search variations, handles large graphs well, without greatly affecting the accuracy. The GED problem is formulated as a Binary Linear Program [14], but the approach is not suitable for large graphs. Nevertheless, algorithms are derived to calculate the lower and upper bounds of the GED in polynomial time. These bounds can be deployed for large graph instances as estimators of the exact GED. The authors of [23] developed new polynomial algorithms which find tighter upper and lower bounds for the GED problem.

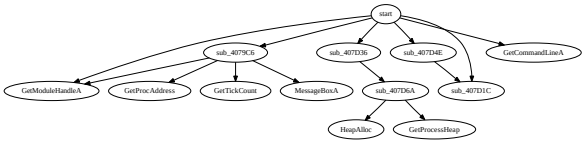


Figure 1: Example of a call graph generated from a real malware sample.

Because in the last part we experiment with function opcodes sequences we must state that it is not something new. Hu et al. describe how they use opcode sequences for finding similar functions [13]. Kolter et al. have used n-grams of byte sequences extracted from malicious files to classify files [16]. Walenstein et al. [22] suggest n-gram frequency vectors and consider their cosine similarity. Finally, Bilar examined the use of opcodes to distinguish malware from benign software [2].

3. BACKGROUND

This section provides a short overview of the terminology and notation used in this paper, along with the appropriate background.

A *graph* $G = (V, E)$ is composed of a set of *vertices* V and a set of *edges* $E \subseteq V \times V$. The *order* of a graph G is the number of vertices $|V(G)|$ in G. In this paper, we will only be dealing with *directed* graphs; an edge (or arc) is denoted by its endpoints as an ordered pair of vertices. Vertex v is said to be a *direct predecessor* of u if $(v, u) \in E$. In this case, u is called a *direct successor* of v. The *out-degree* $d^+(v)$ of vertex v is the number of direct successors of v. Similarly, *the in-degree* $d^-(v)$ is the number of direct predecessors of v. Finally, the *degree* $d(v)$ of vertex v equals $d^+(v) + d^-(v)$. The *out-neighbourhood* (direct successor set) $N^+(v)$ of vertex v consists of the vertices $\{w|(v, w) \in E\}$, and the *in-neighbourhood* (direct predecessor set) $N^-(v)$ is the set $\{w|(w, v) \in E\}$.

A call graph is a directed graph G with a set of vertices $V = V(G)$, representing the functions, and a set of edges $E = E(G)$, where $E(G) \subseteq V(G) \times V(G)$ representing the function calls. An example can be seen in Figure 1. A vertex can represent either one of the following two types of functions: a *local function*, i.e. a function implemented by the program author or an *external function*, i.e. a system or library function. Given a vertex $v \in V$, we denote its name by $V_n(v)$ and by $V_f(v)$ the type of the function represented by v. $V_f(v) \in \{0, 1\}$, where $V_f(v) = 0$ if it is a local function and $V_f(v) = 1$ if it is an external function.

Detecting malware through the use of call graphs requires means to compare call graphs mutually.

Definition (Graph matching): For two graphs, G and H, of equal order, the graph matching problem consists of finding a bijective mapping $\boldsymbol{\phi} : V(G) \rightarrow V(H)$ of optimal value w.r.t. a cost function.

Definition (Isomorphic graphs): Two graphs G, H are isomorphic if there exists a bijective function $\boldsymbol{\phi}$ that maps the vertices $V(G)$ to $V(H)$ such that $\forall i, j \in V(G), (i, j) \in E(G)$, if and only if $(\phi(i), \phi(j)) \in E(H)$.

Graphs are hardly ever isomorphic, so there is a need to define a metric for determining how "far" away they are from being isomorphic.

Definition (Graph Edit Distance): The graph edit distance (GED) of two graphs G, H is the minimum cost induced by elementary edit operations required to transform a graph G into graph H. A cost is defined for each elementary edit operation.

In our case, the elementary edit operations considered are: vertex insertion/deletion, edge insertion/deletion and vertex relabelling. We assign unit cost to all operations. Unfortunately, calculating the GED of pairs of graphs is NP-hard [18, 11] and its decision version is NP-complete.

To find a bijection which maps the vertex set $V(G)$ to $V(H)$, the graphs G and H have to be of the same order. However, the latter is rarely the case when comparing call graphs. To circumvent this problem, the smaller of the vertex sets $V(G)$ and $V(H)$ can be supplemented with disconnected (dummy) vertices ϵ such that the resulting sets $V'(G)$, $V'(H)$ are of equal size. A mapping of a vertex v in graph G to a dummy vertex ϵ is then interpreted as deleting vertex v from graph G, whereas the opposite mapping implies a vertex insertion into graph H. Now, for a given graph matching, we can define three cost functions: VertexCost, EdgeCost and RelabelCost.

VertexCost The number of deleted/inserted vertices: $|\{ v : v \in [V'(G) \cup V'(H)] \wedge [\phi(v) = \epsilon \vee \phi(\epsilon) = v]\}|$

EdgeCost The number of unpreserved 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 is matched to any external function, or if it is an external function matched to a local function or an external function with a different name.

The sum of these cost functions results in the graph edit distance $\lambda_\phi(G, H)$:

$$\lambda_\phi(G, H) = VertexCost + EdgeCost + RelabelCost \quad (1)$$

Definition (Graph dissimilarity): The dissimilarity δ (G, H) between two graphs G and H is a real value on the interval [0,1], where 0 indicates that graphs G and H are identical whereas a value near 1 implies that the pair is highly dissimilar. In addition, the following constraints hold: $\delta(G, H) = \delta(H, G)$ (symmetry), $\delta(G, G) = 0$, and $\delta(G, K_0) = 1$ where K_0 is the null graph, $G \neq K_0$.

Finally, the dissimilarity $\delta(G, H)$ of two graphs is obtained from the graph edit distance $\lambda_\phi(G, H)$:

$$\delta(G, H) = \frac{\lambda_\phi(G, H)}{|V(G)| + |V(H)| + |E(G)| + |E(H)|} \quad (2)$$

As mentioned before, finding the minimum GED, i.e. $\min_\phi \lambda_\phi(G, H)$, is a NP-hard problem but can be approximated (see Section 4).

3.1 Acquiring the call graphs

The call graphs used in the analysis are extracted from binary executable objects, using static methods (similar to those in [12]). The extraction method produces a function call graph and an additional graph representing basic block relationships. Function-level graphs are built from the subroutines represented as nodes and their call references as edges. Basic block call graphs represent the relationships of basic blocks inside a single subroutine.

Static extraction of call graphs starts with a full disassembly of the binary object to assembly language. The disassembled object is then split into individual functions and the functions to basic blocks. Using code analysis, the extraction tool maps the call references between functions and adds them as edges to the graph. A similar operation is performed to map the code references between basic blocks. The type of each function, whether local or external, can be identified and for external functions their name is also retrieved. The latter allows for easy comparison of external functions simply by comparing their names.

While the extraction of call graphs from a proper disassembly is relatively easy, there are a number of obstacles in the real-world implementation. The most common of these are executable packing and obfuscation, which are found in executable objects. Before the call graph can be extracted, the obfuscation or packer envelope must be removed, which can be very difficult and in some cases impossible. Another typical problem arises from inner workings of complex, object-oriented languages, for example C++, Delphi and Visual Basic, which implement a portion of code calls through data references which are very challenging to follow with static analysis.

4. DESCRIPTION OF METHODS

In this section we describe the two main methods mentioned in this paper. These are an approximation using Bipartite matching, which is the main competitor, and Simulated annealing, an adapted version of which we deploy to achieve better results. In 4.3 we also provide an overview of the experimental method by which we attempt to identify similar local functions and reduce the problem size for simulated annealing

4.1 Bipartite Matching

Finding a graph matching ϕ which minimizes the number of edit operations is proven to be an NP-Complete problem [23]. Indeed, empirical results show that finding such a matching is only feasible for low order graphs, due to the time complexity [18]. To overcome this issue, Riesen and Bunke introduced an approximation algorithm which has a good trade-off between accuracy and speed [20, 19]. Their algorithm uses a $(|V(G)| + |V(H)|) \times (|V(H)| + |V(G)|)$ cost matrix C, which gives the cost of mapping a vertex $v \in V'(G)$ to a vertex $v \in V'(H)$. Next, Munkre's algorithm [17] (also known as the Hungarian algorithm), which runs in polynomial time, is applied to find an exact one-to-one vertex assignment which minimizes the total mapping cost. Each entry in this cost matrix C represents the cost of matching vertex $v \in V'(G)$ to a vertex $u \in V'(H)$. Clearly, more accurate cost estimation allows one to find better graph matchings and hence more accurate edit distances. The cost of matching a pair of nodes, $C_{i,j}$ could equal the relabel cost as defined for the graph edit distance in Equation 1:

$$C^{rel}(i,j) = \begin{cases} 0 & \text{if } V_f(i) = V_f(j) = 0 \\ 0 & \text{if } V_f(i) = V_f(j) = 1 \ \wedge \ V_n(i) = V_n(j) \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

Using this relabel cost function, Munkres' algorithm is capable of matching identical external functions in a pair of

graphs, but the local functions pose a problem because the relabel cost function yields no information about the different local functions. As a solution, the authors of [13, 23] independently suggest to embed structural information in the matching cost of two functions. The following equation achieves the latter by also taking the neighborhoods of the vertices (functions) i and j into consideration:

$$\begin{aligned} C_{i,j} =& C^{rel}(i,j)+ \\ & d^+(i)+d^+(j)-2\times(N^+(i)\wedge N^+(j))+ \\ & d^-(i)+d^-(j)-2\times(N^-(i)\wedge N^-(j)) \end{aligned} \quad (4)$$

where the notation $N \wedge M$ denotes the similarity of the neighborhoods N and M, defined as follows:

$$\begin{aligned} N \wedge M = max\{ & \sum_{i \in N}(1 - C^{rel}(i, P(i))| \\ & P : N \rightarrow M \textit{(injective)}\} \end{aligned}$$

In short, the above equation makes the assumption that if two functions i, and j are identical, then they should also invoke the same functions. Similarly, if i and j indeed represent the same function, it is likely that they are also called upon by functions with a high mutual similarity.

4.2 Simulated Annealing

Simulated annealing (SA) is a generic probabilistic method that was first proposed in 1983 by Kirkpatrick et al. [15] to solve hard combinatorial optimization problems. No specific knowledge about the way to approach the problem is required for implementing SA. This allows the use of SA in a variety of problems without changing the basic structure of the algorithm. SA aims at finding the global optimum of a cost function over a set of feasible solutions.

In the call graph matching problem the search space is defined over all the possible bijective mappings $\boldsymbol{\phi}$ between two graphs. The SA process starts from an arbitrary bijective mapping as an initial state. Then a neighbouring state in the search space is selected randomly. Neighbouring states are created by choosing a pair of vertices in one of the graphs and swapping their matching vertices. The difference in the cost function (Equation 1) for the two states determines whether the current state must be replaced by the new state or not. We denote the difference in the cost function evaluated for two states by $\Delta(\lambda_{\boldsymbol{\phi}_t}, \lambda_{\boldsymbol{\phi}_{t+1}})$. If the new state (bijective mapping) gives a lower value for the cost function, it replaces the current state. Otherwise the move is accepted with probability $e^{-\beta\,\Delta(\lambda_{\boldsymbol{\phi}_t}, \lambda_{\boldsymbol{\phi}_{t+1}})}$. SA is allowed to run for a predefined number of steps before the value of β is increased.

The annealed parameter β is the inverse temperature used in statistical physics. For small values of β almost any move is accepted in the process. For $\beta \rightarrow \infty$ the process is essentially a downhill move in which the SA state will be replaced by the new bijective mapping only if the new state gives a lower cost. The reason to introduce the annealed parameter is to overcome the problem of getting stuck in local minima by allowing non preferential moves.

The sequence of β can be considered an *annealing schedule*. The annealing schedule contains the initial and final values of the annealed parameter, denoted by β_0 and β_{final}, together with the cooling rate, ϵ, which determines the changes in β. In our implementation we chose the cooling rate to be a multiplier factor in β which takes values on the interval $[0, 1]$. Then the sequence of the values of β is determined by $\beta_{t+1} = \beta_t/\epsilon$. We will refer to the number of times that β changes with the term *relaxation iterations*. The process terminates either when β reaches β_{final} or when the rate of improving the best solution so far drops below a certain threshold. Upon termination the best encountered solution is returned. The annealing schedule is usually determined empirically.

It has been proven that the probability of reaching the global minimum by simulated annealing approaches 1 in a properly chosen annealing schedule [21]. However, the time needed to reach this global minimum is not guaranteed to be polynomial. In fact in many combinatorial problems it diverges exponentially for large systems and therefore SA becomes inefficient. Interestingly for the graph matching problem it scales efficiently w.r.t. the system size and that allows us to perform SA on very large call graphs.

4.3 Function Code comparison

When call graphs are extracted, the names of the external functions can be identified, allowing external functions to be easily compared using their name. The same does not hold for local functions. Furthermore, local functions can be modified by the malware author. Determining correspondence among local functions requires comparing their code. We attempt to identify similarity among local functions by using the opcode mnemonics of their code.

A simple change in the source code can make the resulting machine code very different, i.e. register allocations may change and cause extended changes to the generated binary. Furthermore, inserting new code may change most of the reference offsets. Instead of just using the byte sequence of the file, using a disassembler to generate structured output of the binary can be used to eliminate some of these changes. Thus, comparison of the binaries can be done with much more detailed information about their structure.

Concentrating on plain opcode mnemonics and ignoring the arguments can overcome the problem of register allocation and offset changes. Reordering of the mnemonics can be eliminated by instead calculating their frequencies. Neighbouring mnemonics can also be treated as n-grams. Those n-grams can be handled either ordered or unordered.

Two functions can be considered similar if they produce identical sequences. A score in the scale of $[0, 1]$ is used to express the similarity of two functions. To compute such a score, the number of occurrences of each mnemonic in pairs of functions is calculated and a vector for each function is generated. The similarity of two functions $u \in G_1, v \in G_2$ based on their frequency vectors, is calculated using the Jaccard index as follows:

$$\sigma(u, v) = \frac{\sum_{i=1}^{n} \min(u_i, v_i)}{\sum_{i=1}^{n} \max(u_i, v_i)}. \quad (5)$$

The above methodology is applied in order to reduce the problem size before SA is applied. The similarity scores for all pairs of functions are calculated in an attempt to detect identical or highly similar functions. Pairs that yield high scores are matched in advance and simulated annealing is used to find a matching for the remaining vertices.

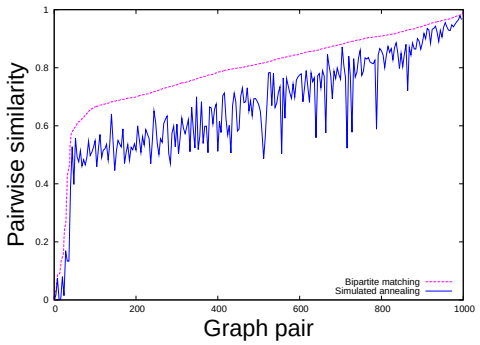


Figure 2: Comparing Simulated annealing and Bipartite matching. 1000 graph comparisons ordered based on the green line.

5. EXPERIMENTS

5.1 Datasets

Our datasets consist of 5 different batches of call graphs. The first batch (we call this dataset D_1) is made up of 340 call graphs classified into 24 labelled families, each one containing 3 to 20 samples. The sizes of the graphs range from 14 to 1087 nodes and from 8 to 2194 edges. D_2, D_3, D_4 are sets of cardinality 675 to 1807 containing graphs of order 50 to 500. These are graphs of executables submitted to F-Secure Corporation on 3 different (close, but not continuous) days respectively. They are a portion of the executables that did not match against any of the existing malware residing in the company's database by using their current methods, potentially a mixture of both malware and benign binary executables. D_5 consists of call graphs belonging to 3 other families, 68 to 137 graphs each. All the mentioned samples have different hash signatures so there is no prior information regarding their similarity. Although many were classified by a human analyst as being in the same family, we do not know the criteria on which the decision was made and even if these criteria are reflected somehow in the call graph structure.

5.2 Experiments & Results

Our experiments include tests to decide upon the annealing schedule and a benchmark for the two competing methods. We also present the results of the comparison of unclassified graphs against an existing library of families using SA. SA is also applied to detect isomorphic graphs.

Initial experiments are needed to determine the annealing schedule of SA. As described earlier, for very low values of β, worse solutions are accepted with great probability which results to steering away from the optimal solution in the search space. On the other hand, setting a high value for β, very few worse solutions are accepted and the method converges more aggressively towards the potentially local extrema. Determining a value for ϵ is equally important. After experimenting with a range of values for the parameters β_0 and ϵ over a set of pairs of graphs, we come to the conclusion that values should be selected from $[3.5, 4.5]$ and $[0.87, 0.93]$ for β_0 and ϵ respectively in order to be able to achieve the best score, not far from the minimum number of overall steps. All the experiments presented here are performed with $\beta_0 = 4, \epsilon = 0.9$.

Having decided on the annealing schedule of SA, we test it independently over each one of the 23 families in D_1. Many isomorphisms among pairs of graphs are identified. By keeping only a single graph from sets of isomorphic within the same family we reduced D_1 to 194 distinct samples (D_1').

The most important experiment in this paper is the comparison of our SA implementation against the approximation that uses the Hungarian algorithm ([13]). In Figure 2 we present the scores achieved by each method for 1000 randomly chosen comparisons out of all possible pairwise comparisons for D_1'. In Figure 2 lower values are better. The reason is that a shorter edit path is discovered and the GED is better approximated.

A clear advantage of SA can be observed based on the score achieved. For almost all the comparisons it succeeds in finding a mapping closer to the optimal. For the vast majority the improvement is significant.

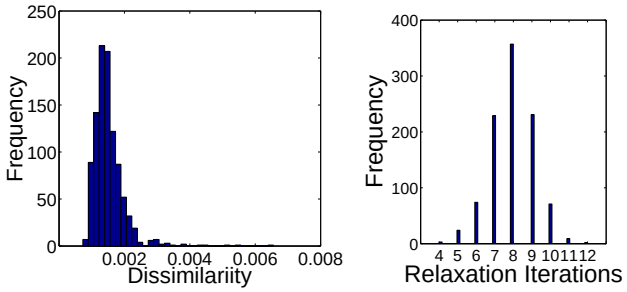
A note on complexity (speed comparison)

The Hungarian Algorithm has complexity $O(|V|^3)$ [7]. When used to decide a mapping among vertices of two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ the complexity becomes $O((|V_1| + |V_2|)^3) = O(|V_{max}|^3)$, where $|V_{max}| = \max\{|V_1|, |V_2|\}$. On the other hand, our implementation of Simulated Annealing has complexity equal to $O(k \cdot |V_{max}|^2 \cdot d_{max})$, where k is the number of relaxation iterations and d_{max} is the maximum degree value taken over all vertices appearing in the two graphs. That is because each relaxation iteration is allowed to run for $|V_{max}|^2$ steps and during each of these steps at most $O(d_{max})$ vertices have to be examined to determine the change in the score. The change in the score $\Delta(\lambda_{\phi_t}, \lambda_{\phi_{t+1}})$ can be determined by examining only the local changes induced by the swap. When SA is allowed to run only for a predetermined number of cooling iterations, k becomes a constant and the complexity is then $O(|V_{max}|^2 \cdot d_{max})$. Even when SA is allowed as many cooling iterations as necessary, the value of k does not exceed 16, based on our experiments. In the extreme case where the maximum value of degree (d_{max}) for any node in the graphs is bounded and does not increase w.r.t. the size of the graph then the overall complexity becomes $O(|V_{max}|^2)$.

An additional advantage of SA is the overhead required to produce the data structures and calculate their values. SA only requires the adjacency matrix and adjacency list of the graphs (for fast retrieval of both edges and neighbouring nodes). Given any of the two, the other can be calculated within $O(|V_{max}|^2)$ steps. For the bipartite matching approach, calculating the matching costs for each pair of vertices requires at least $O(|V_{max}|^2 \cdot d_{max})$ steps since neighbours of vertices must be examined.

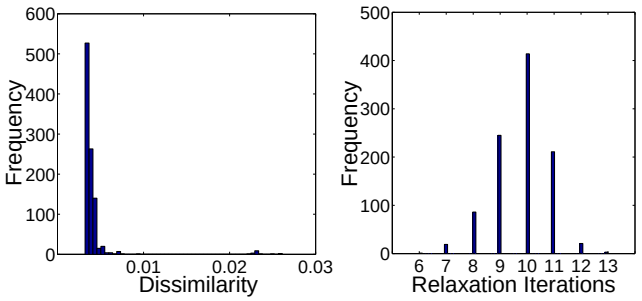
We chose to argue in terms of asymptotic notation in order to avoid results biased by our own implementations for the approximations that use the Hungarian algorithm. In practice, the running times we experience during our experiments are in accordance with the complexity analysis, Simulated Annealing does execute significantly faster and manages to produce better results than all the methods presented so far in this paper.

SA turns out to be consistent in the quality of solutions it outputs and good solutions are achieved on almost all of the attempts. We compare the same pair of graphs (order 790, Edit distance: 1 node & 7 edge insertions/deletions) for a total of 1000 times. The results can be seen in Figure 3. For most of the runs the final score achieved is very



(a) Histogram of dissimilarity scores (b) Histogram of relaxation iterations

Figure 3: 1000 executions of SA on a pair of highly similar graphs, ∼790 nodes, ∼1402 edges.



(a) Histogram of dissimilarity scores (b) Histogram of relaxation iterations

Figure 4: 1000 executions of SA on another pair of highly similar graphs, ∼1084 nodes, ∼2191 edges.

close to the ground truth (Figure 3(a)). There exist a few outliers, though. Such cases are caused by the randomness factor that is involved. The same applies for the distribution of relaxation iterations which exhibits high resemblance to the Gaussian distribution (Figure 3(b)). The same experiment of 1000 comparisons is repeated with another pair of highly similar, but not isomorphic, graphs (GED: 4 nodes, 16 edges, 1 relabel, Dissimilarity: 0.003). The results are depicted in Figure 4. These graphs are of larger order (1087 vertices) and more outliers can be observed. Taking into consideration the size of the graphs and the total execution time, such deviations are acceptable. Outliers can be statistically reduced by repeating the SA procedure; the data structures have already been constructed and and it is only required to start from the same or a different initial solution. SA is also versatile in the sense that if seen as an *anytime algorithm* [24], the user may opt to interrupt the method when a specific score threshold has been achieved, in contrast to the bipartite matching in which results are yielded only when the algorithm has finished executing.

When an AV company receives a suspicious binary executable, it will be compared against the library of known malware to determine if it has been encountered in the past. To simulate this we try to match unlabelled and unclassified samples against a predefined library. All the samples of D_2 and D_3 are compared against our labelled library (D_1'). Pairs with dissimilarity lower than 0.2 (value determined empirically) are treated as belonging to the same family or group. In case of multiple matches the most similar is chosen (nearest-neighbour approach). Taking into consideration the relatively limited number of families in our library

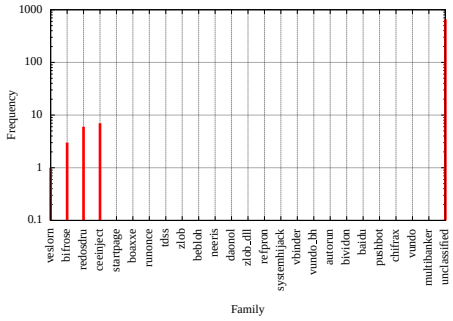


Figure 5: Matches from D_2 against D_1 (log-scale)

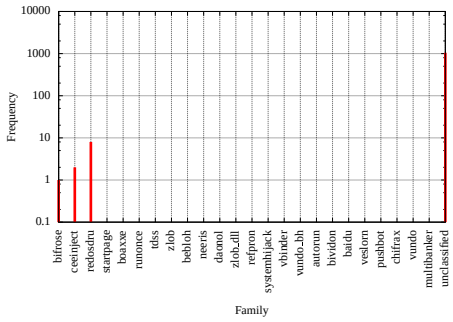


Figure 6: Matches from D_3 against D_1 (log-scale)

and the age difference between D_1 and D_2, D_3 the anticipated number of hits must be very low. Furthermore, the call graphs of D_2 and D_3 have at most 500 vertices but those in D_1' range up to 1087, meaning that for some classes (families) appearing in D_1' no matches can be expected. The results are presented in Figures 5 and 6.

When a batch of new suspicious graphs arrive in queue to be compared against candidate characteristic graphs in the database (DB), it would be sensible to check whether there exist duplicates in the queue. Performing a single comparison query in the DB for each set of duplicates would, in many cases, greatly reduce the total time required to decide for all the samples of the batch. The reason is that for each DB query multiple graph comparisons must be performed. For this scenario, the sets D_2, D_3, D_4, D_5 and $(D_3 \cup D_4)$ were tested for isomorphic function-level graphs using SA. D_2 can be reduced from 675 to 513 unique graphs, which would result to 23.85% fewer queries. For D_3 the reduction is from 1060 to 465 (56.13% fewer queries) and for D_4 from 1807 to 644 (64.36% fewer queries). When combining two daily sets (D_3 and D_4), from a total of (1060+1807=) 2867 samples we managed to reduce them to 1028 unique graphs. This is less than the sum of the unique graphs for those sets individually. This shows that that across different days it is possible to witness binary executables with isomorphic function-level call graphs but which produce different hash signatures and go undetected by current methods.

D_5 is a set of 3 families of malware (*Vedio, Turkojan, Mydoom.A*) that are still active in the wild and get flagged by AV-products. For Vedio the set reduces from 139 graphs to 63 unique. For Turkojan, 68 reduce to 46 unique but for Mydoom.A 107 graphs of different hash signatures correspond to only 2 unique graphs. This result clearly demonstrates the inefficiency of hash signatures and the great advantage of call graphs on that matter.

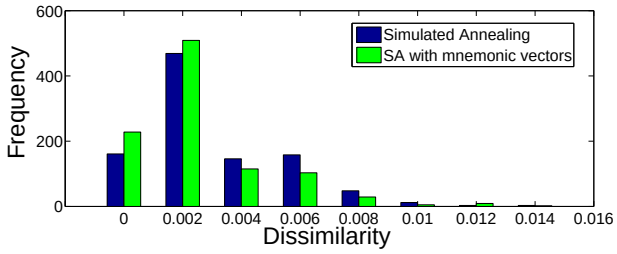
5.3 Optimizations: Opcode mnemonic vectors

As described earlier, when attempting to find a mapping between the vertices of two graphs, equivalence between external functions is straight-forward to determine by comparing their function names. That allows to reduce the actual size of the problem by having to find a mapping only among local functions and non-common external functions. Examining the internals of local functions, i.e basic block graphs, would allow to determine equivalence among them and thereby further reduce the search space of the problem. Such hypothesis is supported by the fact that malware which are characterised by an evolutionary relationship have high probability of sharing large portions of common assembly code since in such cases source code is often reused.

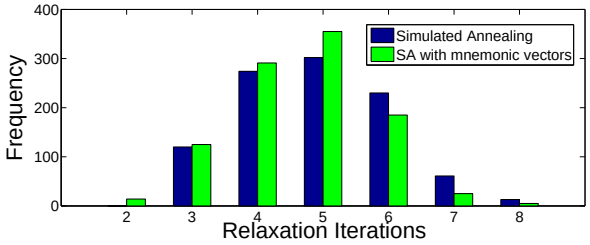
We aim at finding highly similar local functions among pairs of graphs by calculating the mnemonic vectors for each function and its pairwise similarity with functions of the other graph. If pairs of functions are found to be similar, they are considered a permanent match and their vertices will not be considered for the swapping process of SA. In this stage it is very important not to create false-positives. The error of matching functions that are not identical or equivalent propagates until the final solution and cannot be corrected since the function pairings are 'anchored' in advance. According to [2] in both malware and goodware the top 14 most frequent opcodes account for more than 90% of total opcodes that appear and for the 5 most frequent the amount is ∼65%. Consequently, we must be sure that functions do not appear to be similar only because they are of the same size and their body consists of the highly common opcodes. Since the decision for the matches must be made with high confidence, we impose two criterion parameters, one regarding the size of the functions (number of n-grams),t_n, and the other regarding the similarity threshold by which to determine function similarity (Jaccard index value),t_s.

Large functions usually have high degree values in the graph. Pairs of large functions, when considered for a match, yield more matched edges in the objective function so they tend to get matched by SA eventually. So in order for SA to benefit, the set of pre-matched nodes would have to include functions of smaller size as well. Setting $t_n = 75, t_s = 0.95$ and using 3-grams hardly any matches are observed when comparing non-identical samples of the same family. For $t_n = 50$ more hits are observed but still not enough to have any impact on the SA procedure. Lowering also t_s to 0.9 gives a rise of matched functions by ∼15% but only helps in a limited number of cases. Lowering the values of t_n, t_s even further would match functions with lower confidence. On the other hand, if vertices have different out-degree values, then the source code should be different at the parts where the function calls are placed; which makes the use of 3-grams seem very strict. Using 2-grams yields more matches among local functions. For example, in graphs of order 200 containing 80 local functions, up to 20 functions are matched and thus the problem size is reduced significantly.

We demonstrate an example of two graphs of order 352, representing malware from the *Boaxxe* family, which are isomorphic but have different hash signatures. 1000 executions of both SA and SA preceded by mnemonic vector matching are performed. The results regarding the dissimilarity scores achieved and relaxation iterations required can be seen in Figure 7. Out of 173 local functions, 26 get matched.



(a) Histogram of dissimilarity scores



(b) Histogram of relaxation iterations

Figure 7: Comparison of plain SA vs SA enhanced with opcode vectors, 1000 executions

There is significant increase in the number of attempts that managed to find the optimal solution when the vectors were used (Figure 7(a)). The number of relaxation iterations also dropped (in average, from 4.877 to 4.66). In this case we can be sure that there were no erroneous matches that affected negatively the final result, when comparing mnemonic vector similarities, since the isomorphism of the graphs could be detected. It becomes clear that if there is a significant reduction in the problem size, with lack of erroneous permanent matchings, SA becomes more successful in finding the optimal solution within fewer steps.

When performed on-line, the benefits of using such a procedure should be worth the added time that is required. or should not significantly slow down the whole process when no local functions are matched. One of the advantages of using mnemonic vectors is that the vectors can be calculated on-the-fly and appended to the files encoding the graphs, during the call graphs extraction. Since SA only benefits when there is significant problem size reduction, mnemonic vectors should be used only when there is reason to anticipate the existance similar local functions, e.g. when trying to identify the exact malware variant within a specific family.

6. CONCLUSIONS & FUTURE WORK

We demonstrated how Simulated Annealing can be used to approximate the graph edit distance of call graphs. SA outperformed previous approaches both in execution time and solution quality. SA performed very well in various real-world scenarios. This allows us to argue that it should be the preferred method when it comes to implementing very large scale systems based on performing call graph comparisons for the detection of malware. Attempts to reduce the problem size of GED but also experiments on how SA responds to that were carried out by calculating and comparing opcode mnemonic vectors by using the Jaccard index. Simulated annealing benefits in both time and accuracy only when the reduction of the problem size is significant.

In the future, it would be interesting to investigate more

complex edit operations that resemble actual source-code editing, e.g. function merging or spliting, duplication, reordering. In order to achieve that it seems necessary to perform further experiments on the mnemonic vectors, i.e to be able to determine merged functions. We would like to examine the performance of our method implemented on a live system and the performance gain in comparison to existing systems. In the direct future we plan to study the similarities but mostly the differences on the results yielded by call graph comparison against those by dynamic analysis of malware. Furthermore, it would be interesting to examine whether it is possible to distinguish malware from benign software based on the structure of the callgraphs. Effectiveness of call graph comparison-based systems is limited by the inability to extract call graphs from all of the binary executables. Work on tackling the obfuscation techniques would directly benefit large scale call graph comparison.

Acknowledgements

The authors of this paper would like to acknowledge F-Secure Corporation for providing the data required to perform this research. Special thanks go to Pekka Orponen (Head of the ICS Department, Aalto University), Alexey Kirichenko (Research Collaboration Manager, F-Secure) and Gergely Erdélyi (Research Manager Anti-malware, F- Secure) for their valuable support and many useful comments.

This work was partly 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).

This work was partly supported by the Academy of Finland as part of its Finland Distinguished Professor programme, project 129024/Aurell

7. REFERENCES

[1] U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1871–1878. ACM, 2010.

[2] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

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

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

[5] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186, August 2003.

[6] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, 2005.

[7] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[8] H. Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions, Malware and Vulnerability Assessment (DIMVA)*, pages 161–173, 2004.

[9] M. Fossi, D. Turner, E. Johnson, T. Mack, T. Adams, J. Blackbird, S. Entwise, B. Graveland, D. McKinney, J. Mulcahy, and C. Wueest. Symantec Global Internet Security Threat Report: Trends for 2009, Volume XV, 2010.

[10] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis & Applications*, 13(1):113–129, 2010.

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

[12] Hex-rays. The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[13] X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *ACM Conference on Computer and Communications Security*, pages 611–620. ACM, 2009.

[14] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28:1200–1214, 2006.

[15] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671, 1983.

[16] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7, 2006.

[17] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[18] M. Neuhaus, K. Riesen, and H. Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Structural, Syntactic, and Statistical Pattern Recognition. LNCS*, volume 4109/2006, pages 163–172. Springer, 2006.

[19] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950 – 959, 2009.

[20] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*, pages 1–12, 2007.

[21] P. van Laarhoven and H. Aarts. Simulated annealing: theory and applications. *Mathematics and Its Applications, D. Reidel, Dordrecht*, 1987.

[22] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proceedings of BlackHat DC 2007*, 2007.

[23] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009.

[24] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.