Large-Scale Malware Indexing Using Function-Call Graphs[*] [†]

Xin Hu
University of Michigan
Michigan, USA
huxin@eecs.umich.edu

Tzi-cker Chiueh
Stony Brook University
New York, USA
chiueh@cs.sunysb.edu

Kang G. Shin
University of Michigan
Michigan, USA
kgshin@eecs.umich.edu

ABSTRACT

A major challenge of the anti-virus (AV) industry is how to effectively process the huge influx of malware samples they receive every day. One possible solution to this problem is to quickly determine if a new malware sample is similar to any previously-seen malware program. In this paper, we design, implement and evaluate a malware database management system called SMIT (*Symantec Malware Indexing Tree*) that can efficiently make such determination based on malware's function-call graphs, which is a structural representation known to be less susceptible to instruction-level obfuscations commonly employed by malware writers to evade detection of AV software. Because each malware program is represented as a graph, the problem of searching for the most similar malware program in a database to a given malware sample is cast into a nearest-neighbor search problem in a graph database. To speed up this search, we have developed an efficient method to compute graph similarity that exploits structural and instruction-level information in the underlying malware programs, and a multi-resolution indexing scheme that uses a computationally economical feature vector for early pruning and resorts to a more accurate but computationally more expensive graph similarity function only when it needs to pinpoint the most similar neighbors. Results of a comprehensive performance study of the SMIT prototype using a database of more than 100,000 malware demonstrate the effective pruning power and scalability of its nearest neighbor search mechanisms.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Invasive software*

General Terms

Design, Security

Keywords

Malware Indexing, Graph Similarity, Multi-resolution Indexing

[*]A large portion of this work was done while Xin Hu and Tzi-cker Chiueh were at Symantec Research Labs

[†]The work was supported in part by the NSF under Grant No. CNS-0523932, by the ONR under Grant No. N000140911042, and by the AFOSR under Grant No. FA9550-07-1-0423.

CCS'09, November 9–13, 2009, Chicago, Illinois, USA.

1. INTRODUCTION

With the advent of automated malware development toolkits, creating new variants of existing malware programs to evade the detection of anti-virus (AV) software has become relatively easy even for un-skilled aggressors. This has led to a huge surge in the number of new malware threats in recent years. According to Symantec's latest Internet Threat Report [30], the company received 499,811 new malware samples in the second half of 2007 alone. The first step to process any received malware sample is to determine if the sample is indeed malicious. Currently, this step is largely done manually and thus is a major bottleneck of the malware processing workflow. Because most new malware samples are variants of previously-known samples through mutation of their source or binary code, one way to ascertain the maliciousness of a sample is to check if the sample is sufficiently similar to any previously-seen malware program. We describe the design, implementation and evaluation of a graph-based malware database management system, called SMIT (Symantec Malware Indexing Tree) that is developed specifically to perform such checks efficiently.

Most existing malware-detection methods treat malware programs as sequences of bytes, and ignore their high-level internal structures, such as basic blocks and function calls. These methods are generally ineffective against recent malware threats for the following reasons. First, since most modern malware programs are written in high-level programming languages, a minor modification in source codes can lead to a significant change in binary codes. Second, the availability of automated obfuscation tools that implement techniques such as instruction reordering, equivalent instruction sequence substitution, and branch inversion, allows malware writers to easily generate new malware versions that are syntactically different from, but semantically equivalent to, the original version.

One way to overcome the difficulties of recognizing syntactically different and semantically identical variants of a malware program is to base the recognition algorithm on a high-level structure that is less susceptible to minor or local modifications. One example of such high-level structure is a program's function-call graph, which abstracts away byte- or instruction-level details and is thus more resillient to byte- or instruction-level obfuscations commonly employed by malware writers or malware development tools. Moreover, because a program's functionality is mostly determined by the library or system calls it invokes, its function-call graph provides a reasonable approximation to the program's run-time behavior. Therefore, the function-call graphs of the malware variants that are derived from the same source or binary code are often similar to one another. By representing each malware program in terms of its function-call graph, we translate the problem of finding a malware sample's closest kin in a malware database into one that searches for a graph's nearest neighbor in a graph database.

Our work is unique and different from the previous work on graph database query processing for the following three reasons. First, most previous graph database research focused on exact graph

or subgraph matching, which requires a solution to the graph or subgraph isomorphism problems (both are well-known NP problems). However, since malware variants are rarely subgraphs of one another, exact graph or subgraph matching is too restricted to be useful for identifying malware variants. Instead, SMIT supports graph-similarity search, which, given a query graph, pinpoints graphs in a database that are most similar to the query graph. Second, because the cost of computing a graph-similarity score, for example, the graph-edit distance, is exponential in the number of nodes/edges, most existing graph-similarity query methods assume that the number of nodes in the graphs is on the order of 10s. They are not directly applicable to SMIT because the number of nodes in a malware's function-call graph ranges from 100s to 1000s. For example, a variant of the Agobot family has 2,759 nodes and 5,851 edges in its function-call graph. Third, many existing graph-similarity query processing methods cannot scale to a large graph database; their applicable size are mostly on the order of 1000s. Considering the enormous number of malware samples that the AV industry receives every year, the main goal of SMIT is to support efficient similarity queries for databases of the size that is at least 100,000 and up to a million.

SMIT features a unique combination of techniques to address the scalability challenge associated with graph-similarity search. First, SMIT incorporates a polynomial-time graph-similarity computation algorithm whose result closely approximates the inter-graph edit distance. This algorithm exploits the structural and instruction-level information associated with the malware programs underlying the input graphs. Second, SMIT applies an optimistic vantage point tree [9] to index a graph database to speed up nearest-neighbor graph-similarity search. Third, SMIT employs multi-resolution indexing that uses a computationally economical feature vector for early pruning and resorts to a more accurate but computationally more expensive graph similarity function only when it needs to pinpoint the most similar neighbors. We have successfully built a SMIT prototype and tested its performance using a test database containing more than 100,000 distinct malware programs. Our evaluation results demonstrate that SMIT exhibits effective pruning power and scales to large graph databases in that the query service time grows slowly with the number of graphs in the database.

The remainder of this paper is organized as follows. Section 2 reviews previous related work. Section 3 and Section 4 present SMIT's graph-similarity algorithms. Section 5 describes the multi-resolution indexing scheme used in SMIT. Evaluation results for the current SMIT prototype are presented in Section 6. Section 7 discusses SMIT's limitations and Section 8 concludes this paper.

2. RELATED WORK

Most existing work detects or classifies malware based on either byte-level signature or malware run-time behavior. For example, Kolter and Maloof used n-gram of byte codes as features to train the classifier [19]. Rieck *et al.* [27] monitored the malware behavior (e.g., changes to file system and registry) in a sandbox and used supervised learning to predict malware families. Lee and Mody [32] collected sequences of system-call events and applied clustering algorithms to group malware families. Bailey *et al.* [4] defined malware behavior as non-transient state changes on the system and applied hierarchical clustering algorithms for malware grouping. More recently, Bayer *et al.* [5] applied Locality Sensitive Hashing on the behavior profiles to achieve efficient and scalable malware clustering. Both signature- and behavior-based approaches have their own limitations. The former is vulnerable to obfuscation and ineffective in identifying new malware samples. The latter, on the other hand, incurs expensive runtime overhead and tends to generate many false positives. SMIT differs from both in that it builds a large malware database based on their function-call graphs and supports efficient indexing techniques that allow malware analysts to quickly determine whether a new binary file is malicious or not, based on a nearest-neighbor search through the database.

Use of graphs is becoming prevalent in depicting structural information. There exist several methods in the database field for indexing and querying graph databases. Most of them focused on exact graph or subgraph matching, i.e., graph or subgraph isomorphism. However, because both graph and subgraph isomorphism are NP problems [11], existing algorithms are prohibitively expensive to use for querying a large graph database. To reduce the search space, several indexing techniques have been proposed, including GraphGrep [29], Tree+Δ [39] and TALE [31], which use paths, trees and important nodes, respectively, as the main frequent feature to remove graphs that do not match the query. Subgraph isomorphism is then used to prune false positives from the answer set. Several disadvantages of these approaches make them unsuitable for a malware database that contains hundreds of thousands large graphs. First, some of them rely on expensive isomorphism algorithms and thus are only applicable to small graphs. Second, these approaches require all the indexing features to be matched exactly with the query and thus, cannot effectively capture the similarity among malware variants. For example, malware writers often create malware variants by adding new features (e.g., logging) or some cosmetic changes without affecting the essence of the original malware. However, a new variant created this way will not be isomorphic to the original one even though they are similar.

In this paper, we take an approximate graph-matching approach and index the malware graph database using graph similarity. Recently, several indexing methods for similarity queries have also been proposed [14, 34]. Most of them are still built upon exact subgraph isomorphism and therefore, only apply to relatively small graphs, allowing limited approximation. Another widely-used graph similarity metric is the graph-edit distance, which has shown to be suitable for many error-tolerant graph-matching applications [24]. However, because computing graph-edit distance is NP-hard [38], using exact graph-edit distance is feasible only for small graphs. To reduce the computational cost, several methods have been proposed to calculate approximate edit distance [18, 23, 28]. Riesen *et al.* [28] developed a polynomial-time algorithm to compute approximate graph-edit distance using Bipartite Graph Matching. SMIT adopts this approach and tailors it to measure distances between malware call graphs. To support similarity queries (e.g., K Nearest Neighbor query), several techniques for metric space search have also been developed. Yianilos [35] proposed the original Vantage Point Tree (VPT) structure for multi-dimensional nearest-neighbor search. Later, several extensions to VPT have been made to improve its efficiency, such as Multi-way VPT [6], Optimistic VPT [9], and M-tree [37]. They have been successfully applied to various applications, for example, content-based retrieval on multimedia data repositories.

Function-call and control-flow graphs have also been used frequently for malware analysis. Carrera and Erdélyi [8] applied graph theory to function-call graphs for clustering existing malware files. Kruegel *et al.* [20] constructed control-flow graphs from network streams and detected polymorphic worms by identifying structural similarities. Briones and Gomez [7] combined function-call graphs, control-flow graphs and entropy of data blocks to automatically classify malware samples. SMIT differs from others in that it proposes a function-call graph indexing approach towards the important problem of malware classification. It focuses on developing an efficient indexing structure to organize and query large malware databases. In addition, SMIT utilizes a graph similarity metric

based on an optimal bipartite matching algorithm which can better capture the internal structure of the call graphs.

3. FUNCTION-CALL GRAPH EXTRACTION

A binary program's function-call graph is a directed graph consisting of a set of vertices (corresponding to functions), a set of directed edges (corresponding to caller–callee relationships) and a set of labels, one for each vertex (containing the attributes of the associated function). To facilitate matching between two function-call graphs, we classify a program's functions into three categories: (1) **Local functions** are functions written by malware writers and usually shared only by malware variants within the same family; (2) **Statically-linked library functions** are library functions that are statically linked into the final distributed binary, such as Libc, MFC, etc.; (3) **Dynamically-imported functions** are DLL functions that are linked at run- or load-time, e.g., functions in Kernel32.dll, User32.dll, etc. Since these functions are dynamically linked, their bodies do not appear in malware binaries. Both library and imported functions tend to be shared across malware families.

Given an incoming malware sample, SMIT extracts its function-call graph as follows. First, SMIT uses PEiD [2] and TrID [3] to check if the malware file is packed. If so, SMIT applies SymPack (an unpacker developed inside Symantec) to unpack or decrypt the malware file. To handle multi-layer packing, SMIT applies this step recursively until the file is completely unpacked. Then, SMIT uses the popular disassembler IDA Pro [15] to disassemble the malware and identify the function boundaries. It then labels each identified function with a symbolic name. For dynamically-imported functions, their names can be found by parsing the IAT (Import Address Table) in the PE header [25] of the malware file. For statically-linked library functions, e.g., strcmp and iota, SMIT utilizes IDA Pro's FLIRT (Fast Library Identification and Recognition Technology) [17] to recognize their original names. Because the import and library functions are standard routines, their names are consistent throughout all the programs. However, for local functions, since most malware samples do not come with their symbol tables, their names are in general unavailable. As a result, we assign all local functions with the same name (`sub_`) whenever their true symbolic names are unavailable in the input binary. These local functions will later be matched based on their mnemonic sequences or call-graph structures.

To facilitate matching of local functions, SMIT extracts from each local function the sequence of call instructions it contains, and a mnemonic or opcode sequence from instructions in its body. For example, "mov" is the mnemonic for the instruction "mov eax, [0x403FBB]". Such mnemonic sequences are more robust than instruction sequences because they ignore offsets that may change due to code relocation. They are used in the graph-similarity computation as a coarse-grained filter to identify functions from two programs that are likely to be matched. That is, if two functions have similar mnemonic sequences, they are likely to be the same function. SMIT also computes the CRC of mnemonic sequences to speed up the exact matching between sequences. More formally, SMIT defines a program's function-call graph as follows.

Definition (Function-Call Graph): A function-call graph g is a directed graph defined by 4 tuples $g = (V_g, E_g, \mathcal{L}g, L_g)$, where V_g is the finite set of vertices, each corresponding to a function; $E_g \subseteq V_g \times V_g$ is the set of directed edges where an edge from f_1 to f_2 implies that f_1 contains a function call to f_2, but not vice versa; $\mathcal{L}_g$ is the set of labels each of which is comprised of 3 elements: symbolic function name, mnemonic sequence and CRC value of the mnemonic sequence; $L_g : V_g \rightarrow \mathcal{L}_g$ is the labeling function that assigns labels to vertices.

4. GRAPH-SIMILARITY METRIC

The central component of SMIT is a graph database engine that finds the nearest neighbors of a given query graph in a graph database. For this, SMIT uses a graph-similarity metric that aims to capture the similarity among variants within the same malware family, and that can be computed at low cost. Here we give details of this metric: an approximate graph edit distance.

4.1 Graph Edit Distance

The edit distance between two graphs measures their similarity in terms of the number of edit operations required to transform one graph to the other. For the purpose of identifying malware variants, the graph-edit distance effectively captures the amount of effort needed to convert one program to another at the function-call graph level, and thus forms an intuitively appealing metric. Given any two graphs, we define the following two elementary operations to transform one graph to another. **Vertex-edit operations** including: σ_R, relabel a vertex; σ_{IV}, insert an isolated vertex; and σ_{RV}, remove an isolated vertex. **Edge-edit operations** including: σ_{IE}, insert an edge and σ_{RE}, remove an edge.

An edit path $P_{g,h}$ between graphs g and h is defined as a sequence $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ of elementary operations such that $h = \sigma_n(\sigma_{n-1}(\ldots \sigma_1(g) \ldots))$. To quantify this similarity, a cost is assigned to each edit operation: $c : \sigma_R, \sigma_{IV}, \sigma_{RV}, \sigma_{IE}, \sigma_{RE} \rightarrow \mathbb{R}$. Then, the cost of an edit path is the sum of the costs of all the constituent edit operations, i.e., $P = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ as $c(P) = \Sigma_{i=1}^{n} c(\sigma_i)$. The *edit distance* between two graphs is defined as the minimum cost of all edit paths between them, i.e., $ed(g, h) = min\ c(P_{g,h})$. In SMIT, we assign a *unit cost* to each edit operation.

4.2 Approximating Graph-Edit Distance Using Graph Matching

The main drawback of graph-edit distance is its computational complexity, which is exponential in the number of nodes of the graphs. Thus, application of graph-edit distance is feasible only for relatively small graphs, say those with fewer than 50 nodes. Because the number of nodes in malware graphs is significantly larger, we develop heuristic algorithms that can closely approximate the ideal graph-edit distance using graph matching techniques.

To match two unequal-size graphs g and h, we extend the vertex set of each graph as: $V_g^* = V_g \bigcup \epsilon_g$ and $V_h^* = V_h \bigcup \epsilon_h$, where ϵ_g and ϵ_h are sets of dummy nodes created to account for insertions and deletions. In other words, a match from $u \in V_g$ to a dummy node implies the deletion of u from graph g and vice versa. We set $|\epsilon_g| = |V_h|$ and $|\epsilon_h| = |V_g|$ so that the extended graph has the same number of nodes. We denote the extended graph for g as $g^* = (V_g^*, E_g, \mathcal{L}_g, L_g \cup \{\epsilon_g\})$ and define the graph matching as:

Definition (Graph Matching) A matching between two graphs g and h is a bijective function $\phi()$ between two vertex sets, $\phi : V_g^* \rightarrow h_g^*$ such that $\forall v \in V_g^*, \phi(v) \in V_h^*$.

Given a graph matching ϕ between two graphs g and h, the distance (edit cost) between them can be computed by considering mismatched nodes and edges with the following algorithm.

1. Let C_E represent the number of edges that are mapped from one graph to the other. Specifically, for any edge $(i, j) \in E_g$, if $(\phi(i), \phi(j)) \in E_h$, then the matching preserves the edge (i, j) and the counter C_E is incremented by 1.
2. $EdgeCost = (|E_g| - C_E) \times c(\sigma_{RE}) + (|E_h| - C_E) \times c(\sigma_{IE})$. Since we assign unit cost to each edit operation, $EdgeCost = |E_g| + |E_h| - 2 \times C_E$.
3. For any node in graph g that is matched to a dummy node in h, we add $c(\sigma_{RV})$ to the $NodeCost$ to penalize for deleting

the node. Similarly, when a node in graph h is matched with a dummy node in g, we add $c(\sigma_{IV})$ to the $NodeCost$.

4. For any two matched nodes, we add $c(\sigma_R)$ to the $NodeCost$ if they have different labels, i.e., the relabeling cost.
5. Edit distance under ϕ is: $ed_\phi(g,h) = NodeCost + EdgeCost$.

Because graph-edit distance is defined as the minimum edit cost between two graphs, the above algorithm casts the problem of computing graph-edit distance into finding a function ϕ that minimizes the total matching cost, i.e., a minimum-cost bipartite matching problem, where each of the two sides of the bipartite graph corresponds to nodes from one of the two input graphs. An optimal (minimum-cost) bipartite matching can be found in polynomial time ($O(n^3)$) by using the well-known *Hungarian algorithm* [21]. To further reduce the performance overhead of the Hungarian algorithm, SMIT employs various optimizations that exploit properties of the malware programs underlying their function-call graphs.

4.3 Optimizations

4.3.1 Exploiting Instruction-Level Information

Since the complexity of the Hungarian algorithm depends on the number of nodes in the input graphs, the first optimization aims to reduce the number of nodes in the two input graphs that need to be matched by removing those nodes that can be matched through other cheaper means. Specifically, SMIT uses each function's mnemonic sequence, CRC value of its mnemonic sequence and symbolic name to quickly determine if a function in one input graph matches another function in the other input graph, and compute a common function set $C = \{v : v \in V_g \bigcap V_h\}$ containing:

- Static library functions or dynamically-imported functions and that share the same symbolic names in two input graphs;
- Functions that have the same mnemonic sequence and thus the same CRC value of their mnemonic sequence; and
- Functions that have similar mnemonic sequences. We compute the edit distance between the mnemonic sequences of two functions, and consider them a match when the distance is below 15% of the length of the shorter mnemonic sequence of the two, where the threshold 15% is chosen empirically.

To further decrease the number of nodes to which the Hungarian algorithm needs to be applied, we apply a neighborhood-driven algorithm [8] that exploits the matched neighbor information associated with functions. Let's call the functions in $C = \{v : v \in V_g \bigcap V_h\}$ *atomic functions* and let $V_g^r = V_g - C$ and $V_h^r = V_h - C$ denote the sets of the remaining functions in g and h that are not yet matched. A *call-sequence signature* for each remaining function is a sequence of calls to atomic functions in this function. If the call-sequence signatures of two functions $f_1 \in V_g^r$ and $f_2 \in V_h^r$ are identical, we generate a match between f_1 and f_2 because they are likely very similar or the same. Whenever a new match between two local functions is found, we move them from V_g^r and V_h^r to the common function set C, and repeat the algorithm until it yields no additional matches. At the end of the process we apply the Hungarian algorithm to the remaining V_g^r and V_h^r. For malware variants from the same family, this optimization can match over 90% of functions. On the other hand, the number of matched functions for malware from different families is often below 20, most of which are shared library functions.

4.3.2 Bipartite Graph Matching

The problem of finding a min-cost bipartite graph matching can be solved in polynomial-time using the Hungarian algorithm [21]. Once the lowest-cost match is found, it can be used to create an edit path and compute an estimate of the true edit distance (Section 4.2). Note that, although the Hungarian algorithm is optimal, the edit-distance result returned by the match function ϕ that the algorithm finds is only suboptimal [21], because the *cost matrix* used to search for the optimal node assignment is computed without global knowledge (to be elaborated). To mitigate this problem, we develop an optimized Hungarian algorithm that biases the matching process towards the neighboring functions of already-matched functions.

The algorithm first constructs a complete bipartite graph with vertex classes $X = V_g^r \bigcup \epsilon_g$ and $Y = V_h^r \bigcup \epsilon_h$, where ϵ_g and ϵ_h are sets of dummy nodes with $|\epsilon_g| = |V_h^r|$ and $|\epsilon_h| = |V_g^r|$. In this bipartite graph, each edge is assigned a weight corresponding to an estimate of the cost of mapping a vertex $x \in X$ to a vertex $y \in Y$. The choice of weights for the edges of the bipartite graph is a vital component of the algorithm, as well-assigned weights that are closer to the real cost will result in a near-optimal edit path, and thus, the Hungarian estimate will more closely approximate the true edit distance. Assume the first graph g_r has size n, and the second graph h_r has size m, we form an $(m+n) \times (m+n)$ **cost matrix**. In the top left we have an $n \times m$ sub-matrix giving the costs of matching a real node in g to a real node in h. In the bottom right is an $m \times n$ zero sub-matrix, representing the costs of associating a dummy node with another dummy node. Finally, the off-diagonal square sub-matrices give the cost of pairing a real node from a graph to a dummy node from the other graph (thereby deleting it). On the diagonal, these matrices store the cost of deleting a node and all its incident edges (both In and Out). We set all non-diagonal components of these matrices to ∞ to ensure that each real node is associated with a unique dummy node.

In [28], the cost of matching any two real nodes was taken simply as the relabeling cost. To find a better estimate of the true edit cost, we improve the algorithm by considering the edges as well. Specifically, the cost estimate, $C_{i,j}$, of matching node i to node j, is the sum of the **Relabeling Cost** and the **Neighborhood Cost**, where the latter is calculated from the difference between i and j's adjacent nodes. This introduces structural information by giving a lower-bound for the edit cost of matching the neighbors of i and j.

1. Relabeling Cost: If the label of node i is not the same as the label of node j, we set $C_{i,j}$ to be the relabeling cost (σ_R).
2. Outgoing Neighborhood Cost: For any graph g and node $i \in V_g$, $N_{Out}^g(i) \equiv \{L_g(k) | (i,k)\} \in E_g$. Then, the outgoing neighborhood cost of matching node i to node $j \in V_h$ is $|N_{Out}^g(i)| + |N_{Out}^h(j)| - 2 \times |N_{Out}^g(i) \cap N_{Out}^h(j)|$ to $C_{i,j}$.
3. Incoming Neighborhood Cost is similarly defined with the incoming edges.

The cost computed from the above algorithm is a lower-bound of the true edit cost for the following reasons. As mentioned in Section 3, due to lack of symbolic information for all local functions written by malware writers, we assign the same label to those functions. As a result, when computing the estimated matching cost between i and j, any local functions in i's and j's neighborhood are conservatively considered matched (i.e., incurring no matching cost). However, in the final matching function ϕ (found by applying the Hungarian algorithm on the cost matrix), these two neighbor nodes can be unrelated, in which case, the true edit cost between i and j is higher than the estimate. In other words, because the cost matrix is predetermined, the algorithm will only be able to consider the local structure of the nodes without any information about the matching. This lack of global knowledge when computing the cost matrix leads to the sub-optimality of the resulting edit distance as calculated by the algorithm in Section 4.2, even though the Hungarian algorithm by itself is optimal in the sense that it finds the min-cost matching according to the pre-determined cost matrix. To

alleviate this problem, in the next subsection, we present our improved Hungarian algorithm that actively exploits the structural information of already-matched nodes as the algorithm progresses.

4.3.3 Neighbor-Biased Hungarian Algorithm

One drawback of the standard bipartite matching approach to computing the graph-edit distance is that it assumes a fixed cost of matching two function nodes. However, as observed in [14], when two nodes are matched, their neighbors are also likely matched, because if more neighbors of a node are matched with those of another node, the edge-edit cost of matching these two nodes will decrease (thus reducing the real edit cost). Based on this intuition, we develop a modified Hungarian algorithm that adaptively biases the order of matching towards those pairs of nodes whose neighboring nodes have already been matched.

Given two malware call graphs g and h, we first find the initial set of matched functions (Section 4.3.1). For each matched function f, we decrease the cost (in the cost matrix) of matching all the unmatched neighbors of function f in g with their counterparts in h by a predefined percentage. Then, the Hungarian algorithm is applied to the remaining graphs g_r and h_r with the updated cost matrix. In each iteration of the algorithm, whenever two functions, for instance (u, v), are chosen to be matched, the costs of matching their unmatched neighbors in the cost matrix are similarly lowered, thus increasing their chances of being matched later by the algorithm. The procedure repeats itself until a complete match is found in the bipartite graph. As an additional optimization, whenever (u, v) is selected to be matched, the amount of cost reduction for their unmatched neighboring functions is positively proportional to the matching quality of (u, v), defined as the percentage difference between the mnemonic sequences of (u, v). Intuitively, the extent to which the Hungarian algorithm is biased toward the neighbors of a matched node pair is proportional to the degree to which they are considered matched. Due to space limitation, we refer interested readers to our extended technical report [16] for the detailed pseudocode of the algorithm.

The above algorithm generates the cost-minimizing matching between function nodes $\phi : V_g \cup \epsilon_g \rightarrow V_h \cup \epsilon_h$, from which the edit-path cost (denoted as $ed_\phi(g, h)$ under ϕ) can be calculated, which is a close approximation to the true edit distance. Note that $ed_\phi(g, h)$ gives the cost of a particular edit path from g to h. The minimality of edit distance across all edit paths ensures that the distance from the Hungarian method is an upper bound on the edit distance. That is, for any two graphs g and h, $ed(g, h) \leq ed_\phi(g, h)$.

5. MULTI-RESOLUTION INDEXING

5.1 Overview

For the purpose of identifying malware variants, it is not necessary to pinpoint the exact nearest neighbor for a new malware file. As long as one can identify a neighbor that is close enough to the new file, one can "convict" it. For scalability to a large database, SMIT exploits this latitude and incorporates a multi-resolution indexing technique that makes a good balance between pruning efficiency and search effectiveness.

Conventional indexing methods decompose a database into partitions and organize them hierarchically, so that a search can focus on a subset of these partitions at each level of the hierarchy, thus reducing the total number of database items that it needs to touch. These indexing methods are inadequate for SMIT for two reasons. First, SMIT requires an indexing scheme that supports nearest-neighbor search, rather than exact search that conventional methods are designed for. Second, since computation of graph similarity is expensive, SMIT must minimize the number of such computations.

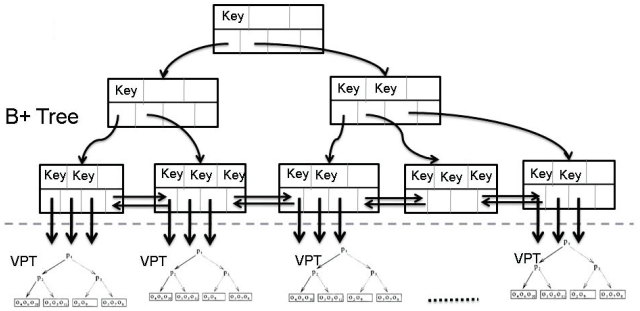

Figure 1: Multi-resolution indexing structure.

For instance, our evaluation shows that a modern desktop PC can perform an average of 20 graph-similarity computations per second for our malware set. At this performance level, even if an indexing scheme could reduce the number of graphs that a search needs to touch, to less than 10% of the database, it will still take hours to answer a single query for a database of 1,000,000 malware graphs.

To address the first problem, SMIT organizes the input malware graph database using the optimistic Vantage Point Tree (VPT), which is designed for nearest-neighbor search and can exploit the fact that sufficiently near neighbors are usually good enough. To solve the second problem, SMIT uses a two-level indexing scheme, where the first level is a standard B+-tree index based on coarse-grained malware features that can be computed inexpensively and that can effectively prune irrelevant parts of the malware database. Graphs associated within each leaf node of the B+-tree index are organized with a second-level index, i.e., the VPT Tree, which uses a more accurate but computationally more expensive graph-similarity function to pinpoint the most similar neighbors. The two-level indexing (Figure 1) in SMIT is an instance of multi-resolution indexing because similarity functions with different accuracy and computational requirements are used in the different levels.

5.2 B+-tree Index Based on Malware Features

The feature vector used in SMIT's first-level index must satisfy two requirements. First, its computation cost must be low. Second, it must be able to identify parts of the malware database that are not relevant to a given malware query. That is, the feature vector needs to be able to pinpoint the obviously irrelevant, but not necessarily the most relevant. Specifically, SMIT uses the following feature vector $v = (N_i, N_f, N_x, N_m)$ derived from the assembly code of each malware program, where N_i is the total number of instructions; N_f the total number of functions; N_x the total number of control transfer instructions (e.g., jumps and calls), which is a good approximation of a program's complexity because it indicates the degree to which a program deviates from a straight-line code; and N_m the median number of instructions per function. The feature vector has the following property: if two malware programs are similar to each other, so are their feature vectors. However, if two malware are dissimilar, their feature vectors may or may not be similar. Therefore, it is only useful when the feature vectors of two malware are drastically different, meaning that the underlying programs are definitely different, but not when their feature vectors are somewhat different or similar.

Because leaf nodes in a B+tree need to be ordered by their keys (feature vectors), we impose a total ordering among feature vectors by giving priority to more useful features ($N_i > N_f > N_x > N_m$). We also augment the B+ tree structure by adding a *backward sibling pointer* to each leaf node, which points to the previous leaf node. Together with the *forward sibling pointer* in the B+-tree, it facilitates navigation across leaf nodes and indexed search.

Given a malware query, SMIT first extracts its feature vector and uses it as a key to search the B+-tree index. Suppose the probing ends in a lead node X. SMIT then follows X's forward and backward sibling pointers to locate N leaf nodes before and after X, and further explores the second-level index trees (VPT) associated with these $2N+1$ leaf nodes. Here N is an empirically-determined parameter that is designed to reduce the probability of the feature vector pruning away sufficiently close neighbors. Because these $2N+1$ VPTs are independent of one another, they can be queried in parallel to reduce the query response time. Finally, the K nearest neighbors returned from the exploration of each of the $2N+1$ VPTs are combined to determine the final K nearest neighbors.

5.3 Optimistic Vantage Point Tree

The Vantage Point Tree (VPT) is designed for database items whose similarity to each other must be explicitly computed (e.g., graphs), and exploits the triangular inequality to prune irrelevant database items. To construct a VPT for a graph database, we first select a graph as the root pivot V, compute the distance between V and all the remaining graphs, and then divide these graphs into M approximately equal-sized partitions ($P_i, i = 1, 2, ..., m$) based on their distance to V. In addition, at the pivot V, we record the distance range associated with each partition P_i, which is represented by $low[i]$ and $high[i]$. This same procedure is repeated for each partition recursively, until all partitions fall below a certain size.

Given a query graph g, the K-nearest-neighbor (KNN) search of a VPT with a root pivot p starts with computation of the edit distance $d(p, q)$ between p and q, and then decide which partitions to explore further by exploiting the triangular inequality of the distance metric. More specifically, let δ_{now} be a parameter indicating to the search algorithm that it should ignore any database item whose distance to the query q is larger than δ_{now}. Given δ_{now}, the search only needs to explore those partitions whose distance range overlaps with the range of interest, $[d(p, q)-\delta_{now}, d(p, q)+\delta_{now}]$, as shown in Figure 2. That is, partition P_i is pruned if and only if

$$high[i] < d(p, q) - \delta_{now} \text{ or } low[i] > d(p, q) + \delta_{now}. \quad (1)$$

This search procedure is applied recursively at each visited node until all nodes are either pruned or visited.

Eq. (1) shows that at each node, the pruning power of the VPT search algorithm is dependent on the value assigned to δ_{now}. If δ_{now} is small, only a few partitions need to be traversed. However, too small a δ_{now} may lead to pruning of the partitions that actually contain the nearest neighbors. One way to keep δ_{now} as small as possible is to update it during the search. At any point in a KNN search, the algorithm remembers the K closest neighbors that it has encountered so far together with their distance to the query graph q in a priority queue, and sets δ_{now} to the largest of these distance values after accumulating K closest neighbors. Every time the search encounters a database item p whose $d(p, q)$ is smaller than δ_{now}, it adds p together with $d(p, q)$ to the priority queue and updates δ_{now} accordingly. Another way to reduce the value of δ_{now} is to traverse the partitions that are closer to the query graph earlier than those that are farther away. For example, in Figure 2, partition 3 is traversed before partition 2 or 4, because closer partitions are more likely to contain closer neighbors.

To make an optimal balance between accuracy and efficiency when initializing δ_{now}, we take an optimistic approach (OVPT) [9] by starting with a small initial δ_{now} value, and exponentially increasing it at subsequent iterations if previous iterations fail to identify K nearest neighbors. Specifically, for a VPT rooted at node p, the initial δ_{now} is chosen to be $\delta_{now} = \max_{i=1}^{m-1} \frac{low[i+1]-high[i]}{2} + 1$ where $low[i]$ and $high[i]$ are the lower and upper ends of the i-th partition's distance range. This choice of the initial δ value guarantees that for any query graph q, at least one partition will be traversed, because $d(q, p)$ will fall within at least one partition's extended distance range, $[low[i] - \delta_{now}, high[i] + \delta_{now}]$.

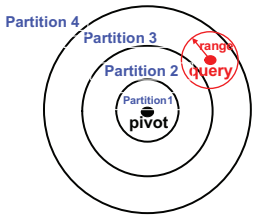

Figure 2: Pruning on a VPT based on the triangular inequality

Feature	Min	Max	Average	Median	STD
N_i	1	1807413	24233.0	7319	55390.9
N_f	1	37130	480.6	85	1077.6
N_x	1	9998	39.1	18	181.4
N_m	0	731350	4932.3	1090	10519.7

Table 1: Statistics of different features in the feature vector

When the initialized value of δ_{now} is too small, the search may not find all K nearest neighbors. In such a case, SMIT increases the initialized value δ_{now} using $\delta_{now,M} = \delta_{now,M-1} + \alpha$ or $\delta_{now,M} = \delta_{now,0} * \beta^{M-1}$where α and β are additive and multiplicative constants and M is the number of iterations that have been attempted to find the K nearest neighbors. To reduce the performance overhead of OVPT, all the distance-computation results in previous iterations are cached so that no distance computation may ever be done more than once in an OVPT search.

The performance gain of OVPT comes from two sources. First, we notice empirically that there is a big difference between the time needed to locate the K nearest neighbors and the time needed to verify that they are indeed nearest neighbors. Using a smaller initial δ_{now} value significantly reduces the verification cost because it cuts down the number of candidates considered, especially when the query graph is indeed close to its nearest neighbors. Second, the optimistic approach carries almost no additional performance overhead because all distance-computation results in previous iterations are cached and can thus be readily reused. More concretely, any partitions that are not pruned in the $(M-1)$-th iteration will never be pruned in the M-th iteration because $\delta_{now,M-1} < \delta_{now,M}$. This means that all the distance computations in previously iterations are necessary, and their caching guarantees that no distance computation will be done more than once.

6. EVALUATION

In this section, we apply SMIT to a large collection of real-world malware files and evaluate its performance in terms of *effectiveness* (whether the results produced by SMIT are meaningful and similar to those produced by human analysts), *efficiency*, and *scalability*. We focus on the K-NN search, because, given the polymorphic nature of modern malware, finding the most similar samples in the database to a given malware file is more useful than pinpointing its exact match.

6.1 Experiment Setup

The dataset used in the evaluation contains 102,391 unique malware programs recently submitted to Symantec Corporation. These malware samples range from simple trojan/virus (less than 100 instructions) to considerably larger malware (more than hundreds of thousand instructions). All the malware files had been analyzed by human experts and classified into families. Each file is labeled with a VID (Virus ID) representing the malware family to which it be-

longs. As a result, we can determine that a binary file used in a query is a variant of an existing malware file if both share the same VID. In total, these malware programs come from 1747 families. We first create a function-call graph representation for each malware file. The graphs have an average number of 504 nodes and 1074 edges, and a maximum number of 37809 nodes and 83737 edges. We implement SMIT in C++ and conduct all experiments on a Dell R905 Server with 1.90 G Quad-Core CPU running Windows Server 2003. SMIT is a CPU-bound application and has a moderate memory requirement (less than 100MB).

To evaluate the performance of SMIT, we use the following three metrics: 1) the percentage of index entries that are accessed to locate the K nearest neighbors of the query file; 2) the percentage of the returned K-NN malware files that are in the same family as the query file; and 3) the average runtime of K-NN search. The first metric measures the average portion of the SMIT index tree that needs to be examined to service a query. The second reflects the accuracy and effectiveness of the SMIT index tree in correctly identifying a new malware. The last one represents the total computation cost for each query. Because SMIT comprises two indexing structures (B+tree and OVPT), we first evaluate them separately and then their aggregate performance when they are combined.

6.2 Effectiveness of B+-tree Index

The first-level B+-tree index in the SMIT index tree uses a computationally economical feature vector representation to attain pruning-efficiency. The statistics of different features are summarized in Table 1, showing that the value distribution of different features varies significantly across malware samples.[1] This wide variation gives the feature vector considerable pruning power and enables SMIT to search only a small number of most relevant VPT trees.

SMIT's B+ tree index takes the following two parameters: 1) the fan-out of each B+ tree node (the maximum number of data entries in each node); 2) the number of adjacent leaf nodes (denoted as N) whose associated second-level VPT trees are further searched. As the fan-out parameter increases, more keys and pointers can be packed into a B+ tree node, fewer nodes are required to hold the index, and fewer tree nodes need to be accessed during a query search. However, larger fan-out parameters also require bigger second-level VPT trees to be explored to achieve better accuracy. This is a typical trade-off between query result accuracy and computation overhead. According to our experience, setting the fan-out parameter to between 300 to 400 achieves a good balance. By default, SMIT sets the fan-out of its B+ tree index to 400, which results in a three-level B+ tree with 209 leaf nodes. On average, each leaf node contains 273 keys (the occupancy ratio 68.3%) and 398 malware programs (some are mapped to the same key). 65% of time, malware programs that are mapped to the same key also have the same VID, i.e., belong to the same malware family.

To evaluate the effectiveness of SMIT's B+ tree index, we randomly select 426 unique malware files and use them as queries against the SMIT's malware database. For 90.8% of these queries, the returned B+tree leaf node contains at least one malware sample that belongs to the same family as the query, and for 96.2% of them, the returned leaf node or its immediate two neighboring leaf nodes contain at least one malware sample that belongs to the same family as the query. Although the end-to-end accuracy in pinpointing a query file's nearest neighbor also depends on SMIT's second-level indexing, i.e., OVPT, and is thus smaller, the high success rate of finding samples of the same malware family as the query file in the same or close-by leaf nodes, demonstrates the efficacy of SMIT's choice of feature vector as used in its B+ tree index.

[1]There are very low feature values such as 0 or 1, because some malware employ various packing or anti-disassemble techniques and cannot be successfully disassembled.

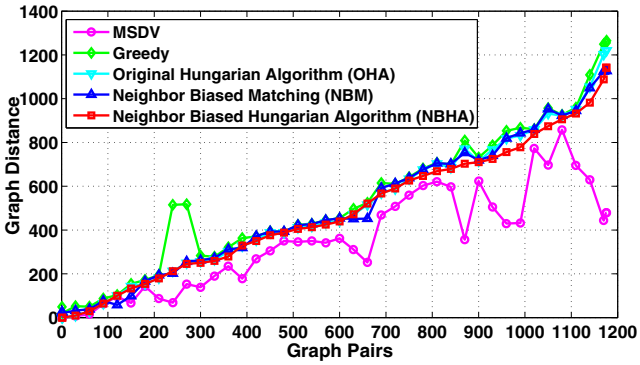

Figure 3: Quantitative comparison among graph distance metrics The X-axis corresponds to a sequence of graph pairs.

6.3 Quality of Graph-Similarity Metric

Accurate graph-distance metric is crucial for SMIT's VPT to correctly prune away irrelevant parts of its malware graph database. Therefore, we first evaluate the quality of the proposed graph distance metric—Neighbor Biased Hungarian Algorithm (NBHA). We compare NBHA with the original Hungarian Algorithm (OHA) [28], the Neighbor Biased Matching (NBM) algorithm [14] and a Greedy algorithm, which computes the distance between two graphs from an edit path formed by repeatedly matching the most similar node pairs according to the cost matrix. The results of all these algorithms, including NBHA, have been shown to be an upper bound for the Exact Graph-Edit Distance (EGED). Because EGED computation incurs an exponential cost, we cannot directly compare NBHA with EGED. Instead, we qualitatively evaluate the closeness of NBHA to EGED by computing a graph distance metric called the *multi-set degree-vector distance* (MSDV), which compares the vertices' label and in/out degree between two graphs without considering their connectivity structure. It has been shown that the MSDV distance is a lower bound for the exact edit distance [10].

We randomly select 66 malware graphs, and compute their pair-wise distance using the graph-distance metrics, NBHA, OHA, NBM, Greedy and MSDV. We order the pair-wise distance values obtained from the NBHA algorithm, and present the distance values from other algorithms according to this order. The results are shown in Figure 3, where each point on the X-axis corresponds to a particular pair of graphs. By definition, true edit distance (EGED) lies between its upper-bound metrics (NBHA, OHA, NBM, Greedy) and lower-bound metric (MSDV). Because in many cases the upper bounds and lower bound shown in Figure 3 are close to each other, these bounds empirically approximate EGED effectively. Moreover, NBHA outperforms other upper-bound metrics (OHA, NBM and Greedy algorithm) in terms of accuracy, because in most cases NBHA's results are smaller than other algorithms'. For upper-bound metrics, smaller metric values imply more accurate approximation to EGED. Specifically, NBHA results are smaller than or equal to those of OHA and NBM, about 95% and 70% of all graph-distance computations in this experiment, respectively.

Next, we evaluate the accuracy and effectiveness of NBHA in terms of the similarity of NBHA results to those produced by human analysts. Specifically, if the distance between two malware files is considered sufficiently small according to NHBA, would the human analysts classify them into the same malware family? To answer this question, we randomly selected from the test database 991 malware samples that belong to 122 malware families. In each experimental run, we first select one malware sample as a query and

K=1	K=3		K=5	
Success Rate	Success Rate	Average Hit	Success Rate	Average Hit
71.30%	78.20%	2.36	80.10%	3.11

K=7		K=9	
Success Rate	Average Hit	Success Rate	Average Hit
81.80%	3.64	82.50%	4.14

Table 2: Accuracy and effectiveness of the NBHA in terms of K-NN search results

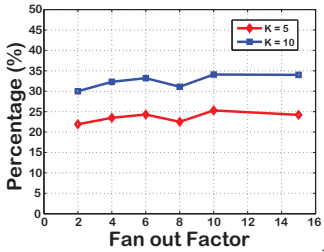

Figure 4: Percentage of index entries (PIE) accessed versus the fan-out factor of the VP tree

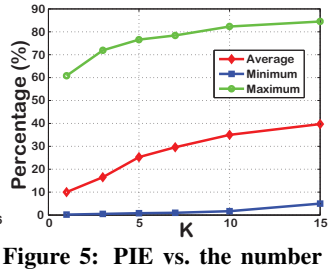

Figure 5: PIE vs. the number of nearest neighbors requested (K) (fan-out factor is 10)

build up a VP Tree for the remaining 990 malware samples. Then, we perform a K-NN search for the query to find the K malware samples that are closest to the query. We repeat the above process for each of the 991 malware samples while varying K, and summarize the results in Table 2. In this table, a K-NN query result is a *Success* if at least one out of K nearest neighbors belongs to the same malware family as the query malware file. *Average Hit* is defined as the average number of the returned K nearest neighbors that are in the same family as the query malware. Results in this table suggest that NBHA is effective in classifying unknown malware samples, because it not only achieves high success rate (over 80% for $K >= 5$) but also produces correct labeling in many cases because the most prevalent malware family among the K nearest neighbors is indeed the query malware's family. This result shows that SMIT can indeed facilitate, and even automate, the process of convicting incoming malware samples.

6.4 Efficiency of Optimistic VPT

We now evaluate the efficiency of Optimistic Vantage-Point Tree (OVPT) using the percentage of index entries (PIE) that need to be accessed to locate the K nearest neighbors of a query file. Because accessing each index entry involves one graph-distance computation, PIE is a proper metric that captures OVPT's computation cost.

We first explore the performance impact of the fan-out factor of SMIT's OVPT (i.e., the number of children each tree node has) and the results are plotted in Figure 4. Although a larger fan-out factor reduces the number of levels in the tree, it also increases the number of child nodes that need to be explored at each tree level, because the coverage of each child node is smaller and more of them intersect with the current query range. As a result of these two conflicting influences, Figure 4 shows that the fan-out factor does not have a significant impact on PIE. However, the larger fan-out factor increases slightly the overall computation overhead.

Intuitively, as K decreases, less graph-distance computation is required to service each query, because smaller K allows δ_{now} to decrease faster so that fewer partitions of each intermediate OVPT visited need to be traversed. However, in practice, a K value of between 5 and 10 is required for human analysts to determine if an incoming binary file is malicious or not. Specifically, if a dominant number of returned K neighbors belong to the same family, there is a very good chance that the query binary file indeed belongs to that family. As shown in Figure 5, although PIE increases with K,

	K=5		
n	Success Rate	Dominant Family Rate	Average Hit
0	76.7%	66.7%	3.24
1	83.3%	70.0%	3.20
2	83.3%	66.7%	3.12
3	86.7%	66.7%	3.13
4	86.7%	66.7%	3.13
	K=10		
n	Success Rate	Dominant Family Rate	Average Hit
0	78.3%	65.2%	6.29
1	87.0%	69.6%	6.30
2	87.0%	69.6%	5.99
3	91.3%	69.9%	5.91
4	91.3%	69.9%	5.98

Table 3: Impact of N on the accuracy of identifying the malware family of a query binary file

SMIT's OVPT can still prune away an average of about 70% of the database even when $K = 10$, i.e., for 10-NN search queries. This result demonstrates the effectiveness of SMIT's OVPT index.

Finally, we evaluate the scalability of SMIT's OVPT with respect to the number of graphs being indexed. Because each leaf node in SMIT's first-level B+ tree corresponds to a second-level OVPT tree, this evaluation also helps shed light on the impact of the fan-out factor of the first-level B+ tree. We construct OVPT trees that contain a different number of malware samples, from 100 to 1000 in increments of 50, and for each resulting OVPT, we query it with 100 randomly-selected malware samples and measure the average number of graph distance computations required for different values of K. Figure 6 summarizes the results and suggests that the number of graph distance computations approximately increase logarithmically with the size of the OVPT tree (the time complexity of searching VP tree is $O(logn)$ [36]), demonstrating its scalability.

6.5 Evaluation of Multi-Resolution Indexing

Despite the great pruning power of the OVPT tree, it cannot be directly applied to organize the entire malware graph database, which we envision will grow to millions. For example, even if an OVPT tree can achieve an excellent PIE of 10%, pinpointing the nearest neighbors of a query in a 100,0000-malware database necessitates over 10,000 graph-distance computations, which is unacceptable for practical use. To ensure reasonable response time while maintaining good query accuracy, SMIT uses a multi-resolution indexing structure that removes irrelevant parts of the database with a B+ tree and queries multiple relevant OVPT trees *in parallel*. Next, we evaluate the accuracy and performance of SMIT's combined indexing structure using 102,391 unique malware programs.

6.5.1 *Impact of N on Query Accuracy*

The parameter N for SMIT's B+ tree determines the number of sibling leaf nodes ($2N + 1$) in the first-level index that need to be searched in the second-level index search. A larger N improves the probability of locating the true K nearest neighbors in the database of the query file, and of correctly identifying the true malware family it belongs to, if any. However, increasing N inevitably increases computational overhead because more second-level OVPT trees are searched. To evaluate the impact of N on SMIT's accuracy, we randomly select 50 malware programs and perform K-NN searches for them with different K (5 and 10) and N (0, 1, 2, 3 and 4). Table 3 summarizes the experimental results. *Success Rate* and *Average Hit* are defined as in Section 6.3 and *Dominant Family Rate* is defined as the percentage of 50 experiments where the most prevalent family among K returned nearest neighbors is also the family to which the query malware belongs. As expected, Success Rate increases with the increase in N. However, the difference in Success

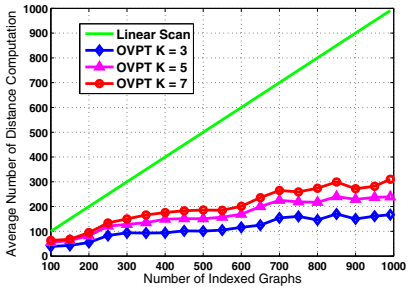

Figure 6: Scalability of the VP tree with respect to the number of indexed graphs

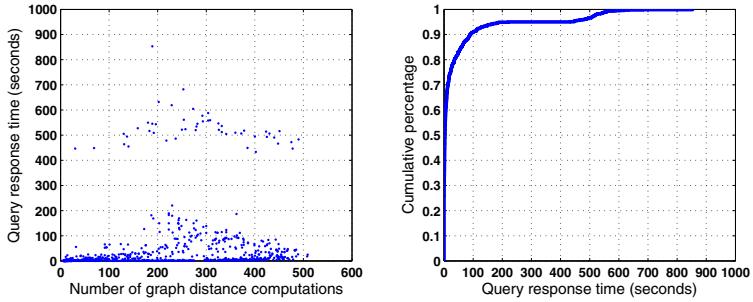

Figure 7: Query response time of 500 five-nearest-neighbor queries against a 100,000-malware database

Rate among $N = 2, 3$ and 4 does not appear significant enough to warrant the extra performance cost. This is because leaf nodes that are far away from the current leaf node usually contain malware files whose feature vectors are quite different from the query malware, indicating that they are likely not in the same family as the query malware. Hence, exploring more leaf nodes (i.e., larger N) does not significantly improve the accuracy. In our current SMIT prototype, we choose $N = 2$ as the default setting. In addition, the high values of Dominant Family Rate and Average Hit in Table 3 also demonstrate the effectiveness of SMIT in helping human analysts identify the malware family of incoming samples.

6.5.2 *Query Response Time of SMIT*

Finally, we measure the response time of SMIT for K-NN queries against the entire test database, where N is set to 2 and K is set to 5. We randomly select over 500 malware files and use them to query SMIT. The response times of these queries and their cumulative distribution function are shown in Figure 7. The X-axis of the left figure is the number of graph-distance computations required for a query and the corresponding Y-axis is the response time in seconds for that query. From the right figure, for over 95% of all queries, the response time is less than 100 seconds, although several queries (mostly for very large malware files) incur a significantly longer delay and thus skew the overall average response time. More specifically, each 5-NN query requires, on average, 112 graph-distance computations (median is 78 and maximum is 918). The query response time ranges from 0.015 second to 872 seconds with average 21 seconds and median 0.5 second. This result demonstrates that SMIT's performance is adequate for day-to-day use even for relatively large malware databases.

7. LIMITATIONS AND IMPROVEMENTS

We now discuss several limitations of the current SMIT prototype that may limit its classification effectiveness, and possible improvements to remove or alleviate them.

One way for malware authors to evade SMIT's classification is to prevent SMIT from extracting useful features by applying packers/protectors to their malware files. SMIT's classification accuracy will degrade significantly if it cannot successfully unpack packed malware files. To counter the packing problem, the current SMIT prototype employs several packer detection (PEiD, TrID) and unpack tools (SymPack), but they are by no means complete. For example, PEiD can be misled by a simple modification to a PE file's entry point. Most existing unpack tools fail to handle sophisticated packers, such as Armadillo [1] and VMProtect [33]. To improve SMIT's unpacking capabilities, we plan to incorporate generic unpackers, such as OmniUnpack [22] and Justin [12], which execute malware samples, detect the end of unpacking and then dump the process image at that instant. The extra performance overhead associated with these techniques is generally acceptable, because SMIT is mainly positioned as a back-end malware classification and analysis tool.

Second, because SMIT analyzes malware samples at the level of individual instructions and function calls, it may be susceptible to advanced obfuscation techniques. For instance, attackers may circumvent SMIT's function matching by obfuscation, such as instruction reordering, equivalent instruction substitution, import table modification (to hide the symbolic names of imported functions), etc. Alternatively, they could also modify the function-call graph by, for example, inserting useless functions into the graph, breaking existing functions into several smaller functions, inlining certain functions, etc. Although SMIT cannot completely handle all types of obfuscation, it makes these attacks more difficult. For instance, SMIT uses the edit distance between mnemonic sequences to evaluate inter-function similarity, which enables SMIT to be relatively resilient to simple code obfuscation and relocation. To defeat more sophisticated obfuscation, SMIT could pre-process malware files with advanced deobfuscation techniques [26]. More importantly, because SMIT relies on *structural similarity* to match function-call graphs, changes to a few nodes in the graphs are unlikely to significantly influence the matching results.

Third, SMIT extracts function-call graphs using IDA Pro, which may occasionally fail to identify all the functions in a malware binary. IDA Pro finds function-start addresses by traversing direct function call or recognizing function prologues. As a result, if the functions are indirectly referenced or have non-standard prologues, IDA Pro may fail to identify their starting points. A more thorough approach [13] that uses a new function model based on a multi-entry control flow graph could mitigate this problem.

Finally, the dominant family metric used in SMIT may lead to false positives. Because SMIT is mainly used to help malware analysts quickly determine the maliciousness and the identity of incoming malware, it assumes that the query malware sample belongs to the same family as the majority of its nearest neighbors in the database. However, this assumption is not always valid and a false positive may occur if the distance between an input malware sample and its dominant family neighbors is too large. One way to address this problem is to apply a distance threshold in the classification process so that an input sample is classified into a malware family if and only if it is sufficiently close to the returned nearest neighbors. The optimal threshold could be chosen based on the average inter-member distance within a malware family as well as the inter-family distance between the centroids of adjacent families.

In summary, although there are ways malware writers could use to detract SMIT's overall effectiveness, SMIT is still very effective in practice against modern malware samples, as demonstrated in Section 6, and thus represents a very efficient tool available for malware analysts to handle the exponentially-growing influx of malware samples as seen in recent years.

8. CONCLUSION

In recent years, the number of malware samples seen in the field has increased exponentially, and automating the malware processing workflow is crucial to commercial anti-virus companies such as Symantec. A critical step in malware processing workflow is to determine if an incoming sample is indeed malicious or not. A common approach taken today is to apply multiple commercial Anti-Virus scanners to a sample and convict the sample as malware if a sufficient number of Anti-Virus scanners consider it malicious. Although this approach is useful, it does not completely solve the problem, because at any point in time a significant percentage of new samples are unknown to existing Anti-Virus scanners.

This paper describes the design, implementation and evaluation of a malware database management system called SMIT that implements a malware conviction approach which casts the problem of determining if a new binary sample is malicious into one of locating the sample's nearest neighbors in the malware database. SMIT converts each malware program into its function-call graph representation, and performs nearest neighbor search based on this graph representation.To efficiently capture the similarity among malware variants, SMIT supports an approximate graph-edit distance metric rather than isomorphic graph match. To efficiently support accurate and scalable nearest neighbor search, SMIT features a multi-resolution indexing scheme that combines a B+ tree based on high-level summary features and a vantage-point tree based on the graph-distance metric. With these techniques, SMIT is able to detect malware samples at a speed and accuracy level that can keep up with the current malware sample submission rate. The main contributions of this work include: (1) an efficient graph-distance computation algorithm whose result closely approximates the ideal graph-edit distance metric; (2) a multi-resolution indexing scheme that supports efficient pruning through a combination of exact indexing based on summary features and nearest-neighbor indexing based on graph-edit distance; and (3) A fully working SMIT prototype and a comprehensive performance study of this prototype that demonstrates its efficacy and scalability with a 100,000-malware database.

9. REFERENCES

[1] Armadillo. http://www.siliconrealms.com/armadillo.htm, 2008.

[2] Peid 0.95. http://www.peid.info/, 2008.

[3] Trid v2.02. http://mark0.net/soft-trid-e.html, 2008.

[4] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, pages 178–197, 2007.

[5] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security*, 2009.

[6] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *In Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

[7] I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

[8] E. Carrera and G. Erdelyi. Digital genome mapping ął advanced binary malware analysis. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

[9] T.-c. Chiueh. Content-based image indexing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 582–593, 1994.

[10] S. Das, A. Mistry, D. Negoescu, G. Reed, and S. K. Singh. A graph matching problem. Techical report, IPAM Research in Industrial Projects for Students (RIPS), 2008.

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

[12] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *RAID '08*, pages 98–115, 2008.

[13] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, 2005.

[14] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 38, 2006.

[15] Hex-rays. The IDA Pro Disassembler and Debugger. http://www.hexrays.com/idapro/, 2008.

[16] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs (extended). Technical Report, Department of Computer Sicence, University of Michigan, 2009.

[17] Ilfak Guilfanov. Fast Library Identification and Recognition Technology. http://www.hex-rays.com/idapro/flirt.htm, 1997.

[18] D. Justice. A binary linear programming formulation of the graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(8):1200–1214, 2006. Fellow-Hero,, Alfred.

[19] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, 2006.

[20] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *In RAID*, pages 207–226. Springer-Verlag, 2005.

[21] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[22] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *In Proceedings of the Annual Computer Security Applications Conference (ACSAC*, 2007.

[23] R. Myers, R. C. Wilson, and E. R. Hancock. Bayesian graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(6), 2000.

[24] M. Neuhaus and H. Bunke. An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In *SSPR/SPR*, pages 180–189, 2004.

[25] M. Pietrek. An In-Depth Look into the Win32 PE File Format. http://msdn.microsoft.com/en-us/magazine/cc301805.aspx, 2002.

[26] J. Raber and E. Laspe. Deobfuscator: An automated approach to the identification and removal of code obfuscation. *Reverse Engineering, Working Conference on*, 0:275–276, 2007.

[27] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA '08*, pages 108–125, 2008.

[28] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*, volume 4538, 2007.

[29] D. Shasha, Jason, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.

[30] Symantec Corp. Symantec Global Internet Security Threat Report. Volume XII. http://www.symantec.com/, April 2008.

[31] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[32] T.Lee and J.J.Mody. Behavioral classification. http://www.microsoft.com/downloads/details.aspx?FamilyID =7b5d8cc8-b336-4091-abb5-2cc500a6c41a&displaylang=en,2006.

[33] VMProtect. Vmprotect. http://www.vmprotect.ru/, 2008.

[34] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005.

[35] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1993.

[36] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*. Springer, 2006.

[37] P. Zezula, P. Ciaccia, and F. Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. Technical Report 7, HERMES ESPRIT LTR Project, 1996.

[38] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[39] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta $\leq$ graph. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 938–949, 2007.