Leveraging Compression-based Graph Mining for Behavior-based Malware Detection

Tobias Wüchner, *Student Member, IEEE*, Aleksander Cisłak, Martín Ochoa, and Alexander Pretschner, *Member, IEEE*

Abstract—Behavior-based detection approaches commonly address the threat of statically obfuscated malware. Such approaches often use graphs to represent process or system behavior and typically employ frequency-based graph mining techniques to extract characteristic patterns from collections of malware graphs. Recent studies in the molecule mining domain suggest that frequency-based graph mining algorithms often perform sub-optimally in finding highly discriminating patterns. We propose a novel malware detection approach that uses so-called *compression-based* mining on *quantitative data flow graphs* to derive highly accurate detection models. Our evaluation on a large and diverse malware set shows that our approach outperforms frequency-based detection models in terms of detection effectiveness by more than 600%.

Index Terms—malware detection, quantitative data flow analysis, data mining, graph mining, machine learning

1 Introduction

Malware remains one of the largest IT security threats with thousands of new variants appearing on a daily basis, causing yearly losses of billions of dollars. As malware development has become a lucrative business model [4], [16], today's malware landscape is highly sophisticated and makes use of a diverse portfolio of advanced obfuscation and anti-analysis techniques [2], [41]. This challenges traditional signature-based detection because polymorphic malware often autonomously creates obfuscated siblings with completely differently looking binaries.

As a countermeasure, behavior-based malware detection has gained considerable momentum in the past decade. Unlike static detection, behavior-based detection approaches do not use the binary of a malware sample for profiling and detection but rather detect malware by learning and later detecting typical malicious behavior. One prevalent type of behavior models uses graphs to represent system calls [6], [7], [12], [18], [20], [24], [31], or resource dependencies [23], [36]. The most common way of leveraging such graph-based behavior models for malware detection is to scan unknown graphs for characteristic malware behavior patterns (i.e. sub-graphs[1]). Repositories of such patterns are either manually specified or extracted from sets of known malware graphs by graph mining.

The core idea of graph mining is to determine discriminating patterns shared by many graphs in a training set: patterns that are useful to accurately separate graphs of known malicious and benign samples. Most malware detection approaches that use graph mining determine the utility of a pattern from a frequency point of view [5]–[7], [18], [20], [24], [26], [31]. This means that the utility of a pattern is determined depending on how often it appears in the analyzed malware samples, irrespective of other properties of the pattern. As a consequence, graph mining for behavior-based malware detection is usually done using popular frequency-based algorithms like AGM [17], gFSG [22], or GSpan [40].

Recent results from the molecule (graph) mining domain indicate that frequency-based mining often yields significantly less interesting and thus less discriminating patterns than so-called compression-based mining approaches [21]. Unlike frequency-based mining approaches, compression-based mining approaches do consider the structural complexity of a pattern candidate to determine its utility. They do so by accounting for the capability of a pattern to "shrink" the graphs of the mining set. That is, a pattern that compresses, i.e. covers, a large portion of most graphs in the mining set, but that occurs less frequently than another less complex pattern with more limited compression capabilities, might still be more discriminating than a less complex but more frequent pattern. (To avoid confusion, please note that "compression" is to be understood in an intuitive rather than an information-theoretic way: the "compressions" we consider usually are lossy.)

To our knowledge, the utility of compression-based graph mining for malware detection has not yet been investigated. We see good reason to believe that the insights gained in the molecule mining domain carry over to malware detection. This assumption is substantiated by the results we obtained from a preliminary study where we applied a state-of-the-art frequency-based mining approach [40] on data flow graphs obtained from a large body of malware samples. The resulting patterns, although in principle discriminating and useful for malware detection, almost entirely referred to rather simplistic behavior like reading certain system libraries or writing specific registry keys. Using such simple patterns for malware detection is problematic because they likely are a) very sensitive to changes in the behavior of the profiled malware families, b) for the same reason comparably easy to circumvent [1], and finally c) might miss important and more complex malware-specific behavioral patterns like e.g. self-replication.

1. For the sake of brevity we will use the terms *pattern* and *sub-graph*, as well as the terms *system call* and *Windows API call* interchangeably.

We hence propose to use a *compression-based* graph mining approach for behavior pattern extraction. Adopting a behavior-based detection model from the literature [36] that represents malware behavior as Quantitative Data Flow Graphs (QDFGs), we show that patterns mined with the compression-based algorithm outperform patterns that were mined with the purely frequency-based approach in terms of malware detection accuracy. We further show that considering *quantitative data flows*, encoded in QDFGs, for determining graph compression levels yields better results than using graph structure properties for computing compression factors.

Problem: We address the problem of finding interesting patterns in malware behavior graphs that are sufficiently discriminating to provide high detection accuracy at reasonable mining costs. In particular, we aim at a notion of pattern utility that leads to better detection results than simply considering pattern frequency as utility metric, as usually done in related work.

Solution: To mine discriminating malware behavior patterns we adapt and modify a well-known compression-based graph mining algorithm for QDFGs. QDFGs model malware behavior as aggregations of quantified data flows between system entities and are induced by executed system calls. Matching the obtained patterns on QDFGs of known malware and benign software we train a supervised machine learning classifier that we use for classifying unknown malware samples.

Contribution: To our best knowledge we are i) the first to use *compression-based graph mining* for behavior-based malware detection using *quantitative data flow information* and ii) we show that patterns obtained using compression-based mining lead to *600% more accurate* detection results than patterns obtained with a commonly-used frequency-based mining algorithm.

Organization: After recapping the concept of graph mining and introducing a quantitative data flow model that underlies our approach in §2, we introduce its steps in §3. We discuss its evaluation in §4, put our work into context in §5, and conclude in §6.

2 PRELIMINARIES

In the following we recap an abstract system model from the literature that represents low-level behavior as QDFGs, which we use to model malware behavior. We also briefly recap the concept of graph mining.

2.1 Quantitative Data Flow Graphs

Approaches that directly leverage raw system calls for detection have been shown to be sensitive to behavior obfuscation [1]. We therefore use a more abstract model to capture system call traces as QDFGs.

QDFGs represent a system's behavior during a defined period of time as aggregated (quantifiable) data flows between system entities like processes, files, registry entries, or network sockets [36]. QDFGs are generated by interpreting traces of intercepted system calls as quantitative data flows between pairs of system entities. Executing a *ReadFile* Windows API call, for instance, yields a flow of a certain quantity of bytes from the read file to the process that issued the call. Conversely, a *WriteFile* Windows API call yields a flow of data from the calling process to the file to be written.

Nodes in a QDFG are system entities. Edges model the data exchange that happened between entities (nodes) as a result of system call executions. Formally, a QDFG is a graph $G = (N, E, A, \lambda) \in \mathcal{G} = \overline{N} \times \overline{E} \times \overline{A} \times ((\overline{N} \cup \overline{E}) \times \overline{A} \rightarrow \mathit{Value}^{\overline{A}})$, where $\overline{N}$ models the set of all possible QDFG nodes; $\overline{E} \subseteq \overline{N} \times \overline{N}$ the set of possible edges between nodes; and a set of labeling functions $((\overline{N} \cup \overline{E}) \times \overline{A}) \rightarrow \mathit{Value}^{\overline{A}}$ maps attributes ($\overline{A}$) of nodes or edges to their respective values ($\mathit{Value}^{\overline{A}}$). Labeling functions annotate nodes and edges with additional information like node type ($type \in \{\texttt{PROCESS}, \texttt{FILE}, \texttt{SOCKET}, \texttt{REGISTRY}, \texttt{URL}\}$), or aggregated amounts of transferred data ($size \in \mathbb{N}$).

System calls with obvious data flow semantics, i.e., when executed lead to a flow between system entities, are modeled by the set $\mathcal{E} \subseteq \overline{E}$. A data flow event $(src, dst, size, time, \lambda) \in \mathcal{E}$ represents the transfer of $size \in \mathbb{N}$ units of data at time $time$ from a source $src \in \overline{N}$ to a destination node $dst \in \overline{N}$ with a labeling function λ for timestamping and associating additional information on the corresponding flows. As we do not need to reconstruct individual flows later but rather want lean and aggregated models, we simplify our model by summing up the $size$ attribute of all data flows between pairs of nodes instead of creating one distinct edge per event.

QDFGs are generated and continuously updated by intercepting relevant system events, i.e. system or API calls, by reference monitors [38] and interpreting them according to their data flow semantics. We need some auxiliary notation before formally defining the graph update function that creates or updates graph nodes or edges in correspondence to detected data flow events: For $(x, a) \in (N \cup E) \times \overline{A}$, we define the attribute update function $\lambda[(x, a) \leftarrow v] = \lambda'$ as

$$\lambda'(y) = \begin{cases} v & if\ y = (x, a) \\ \lambda(y) & otherwise. \end{cases} \quad (1)$$

For updating multiple attributes, we use the notation $\lambda[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_k] = (\ldots(\lambda[(x_1, a_1) \leftarrow v_1])\ldots)[(x_k, a_k) \leftarrow v_k]$. Additionally, the composition of two labeling functions is defined by:
$\lambda_1 \circ \lambda_2 = \lambda_1[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_k]$ where $v_i = \lambda_2(x_i, a_i)$ and $(x_i, a_i) \in \mathsf{dom}(\lambda_2)$.

To simplify access to QDFG components, i.e. nodes, edges, attributes, and labeling functions, we furthermore introduce a tuple selector notation: $G.N$, for instance, yields the set of nodes N of the QDFG $G = (N, E, A, \lambda)$.

The graph update function $update : \mathcal{G} \times \mathcal{E} \rightarrow \mathcal{G}$ is then formalized by equation (2) defined below.

To operationalize the generic QDFG model in a Windows malware detection context, we map abstract QDFG nodes to concrete Windows resources like processes, files, sockets, or registry entries. Furthermore, we instantiate the set of data flow *events* by modeling all Windows API functions that, upon execution, potentially lead to a data flow between system entities. This includes file system access functions like *ReadFile* and *WriteFile*, networking functions like the Winsock *recv* and *send* functions, memory access functions like *ReadProcessMemory* and *WriteProcessMemory*, or registry manipulations like *RegQueryValue* and *RegSetValue*.

$$update(G, (src, dst, s, t, \lambda')) = \tag{2}$$
$$\begin{cases} \begin{pmatrix} N, \\ E, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow \lambda(e, size) + s; \\ (e, time) \leftarrow (\lambda(e, time) \cup \{t\}) \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{if} \quad e \in E \\ \begin{pmatrix} N \cup \{src, dst\}, \\ E \cup \{e\}, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow s; \\ (e, time) \leftarrow \{t\} \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{otherwise} \end{cases}$$
where $e = (src, dst)$ and $G = (N, E, A, \lambda)$

We give an intuition of how functions are formalized in this model for two sample functions only. A more comprehensive list of event formalizations is described elsewhere [36]:

- Using function **ReadFile**, a process reads a specified amount of bytes from a file to its memory. *Relevant Parameters*: Calling Process (P_C), Source File (F_S), ToReadBytes (S_R), time t. *Mapping*: $update(G, (F_S, P_C, S_R, t, \emptyset))$.
- Using function **WriteFile**, a process writes a specific number of bytes to a file. *Relevant Parameters*: Calling Process (P_C), Destination File (F_D), ToWriteBytes (S_W), time t. *Mapping*: $update(G, (P_C, F_D, S_W, t, \emptyset))$.

2.2 Graph Mining

Graph mining specializes the more general concept of data mining. Traditionally, data mining focused on extracting rules and patterns from unstructured or semi-structured data [35]. Structuring data in the form of graphs is very common in computer science and very naturally applies to problems in chemistry, biology, and medicine. Extending the concept of data mining to extract patterns from structured graph data recently received considerable attention.

Graph mining algorithms can be either *lazy* or *exhaustive*, and *frequency-* or *compression-based*. Exhaustive graph mining algorithms evaluate all possible sub-graphs that can be built from a set of graphs to isolate the most descriptive ones. This usually involves computing isomorphisms between each pattern candidate and each graph of the set. Because this boils down to the NP-complete [11] sub-graph isomorphism problem, exhaustive graph mining algorithms suffer from bad scalability and are expensive to run on huge data sets. On the upside, by construction they yield optimally discriminating patterns.

Lazy graph mining algorithms do not evaluate all possible sub-graphs. Instead, they incorporate domain knowledge or structural heuristics into the search process to quickly prune those parts of the search space that are unlikely to yield interesting patterns. Lazy algorithms usually are faster than exhaustive ones but miss potentially discriminating patterns from parts of the pruned search space.

Frequency-based graph mining algorithms like GSpan [40] or AGM [17] define the level of utility of a pattern exclusively by how often it occurs within the training data, also called the the *frequency*, or *support*, of a pattern. The complexity or other properties of the pattern are usually not considered.

Compression-based mining approaches like Subdue [21] or GBI [27] typically also consider pattern frequency. In addition, they account for the structural complexity of a pattern. This is usually done by evaluating the degree of *graph compression* that can be achieved when compressing ("shrinking") all sub-graphs in the training set that are isomorphic to the pattern to a single node. The shrinking factor, defined as the ratio between the complexity of the uncompressed and the compressed graphs, then determines its utility. As a consequence, a compression-based graph mining algorithm might sometimes favor a less-frequent but highly compressing pattern over a more frequent one. Recent work in the domain of molecule mining indicates that patterns obtained via compression can perform better than those extracted with purely frequency-based methods [21].

We consider our approach to be *compression-based* as we employ a lazy compression-based graph mining algorithm [21]. We expect it to provide a good trade-off between effectiveness and efficiency by incorporating domain knowledge and advanced candidate selection techniques to aggressively prune the pattern search space.

3 Approach

Our approach follows a classical data mining rationale. As for most behavior-based malware detection approaches, our core assumption is that we can detect new malware based on its behavioral similarities with already known malware samples. We aim at extracting characteristic behavior patterns from known malware samples to define detection models that discriminate malware from benign software with high accuracy. As we represent system behavior as QDFGs, a natural choice for extracting such behavioral patterns is a graph mining algorithm.

Our approach consists of the following steps:

3.1) *Data Retrieval*: To generate the raw data input to the mining step we employ a dynamic malware analysis infrastructure from the literature [36]. Using this infrastructure we generate system call traces for each sample that we obtain when executing it in the infrastructure's malware sandbox. We generate QDFGs for those traces by translating each system call into the induced (quantitative) data flow, as discussed in §2.1. This yields the training set for subsequent learning steps.

3.2) *Pattern Mining*: Using a compression-based graph miner, we extract characteristic patterns, i.e. sub-graphs, from the generated malicious QDFGs. This yields a repository of graph patterns that capture the essence of the behavior of known malware. Following our baseline assumption, we expect these patterns to also appear in unknown malware with high likelihood.

3.3) *Pattern Matching (3.3.1) and Classifier Training (3.3.2)*: Individual patterns in themselves likely are not sufficiently discriminating to accurately differentiate between benign software and malware. This leads us to introduce a second learning step. We again match the mined patterns on the training set and record which patterns matched which malware and which benign

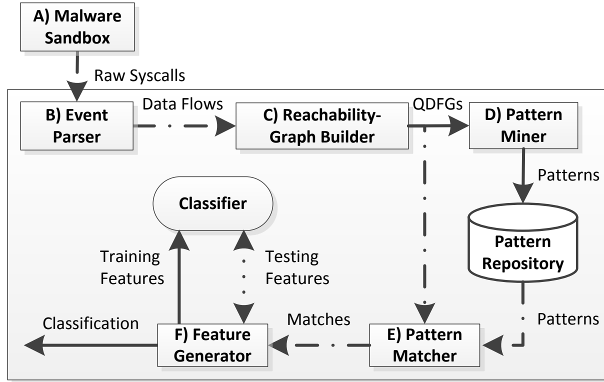

Fig. 1: High-level architecture.

software. Using this information we train a supervised classifier to learn complex relationships between different combinations of mined patterns and their occurrence in known malware and benign software graphs.

3.4) *Detection*: The classification of unknown samples is done by matching the mined patterns against the unknown sample's graph and then passing the obtained matching information to the trained classifier. Based on the similarity of the unknown sample's matching profile with matching profiles of the known benign software and malware samples from the training set, the classifier suggests the unknown sample to more likely be malicious or benign.

Figure 1 depicts a high-level overview of our approach with the conceptual components that realize the aforementioned training and detection steps. The solid arrows in the figure mark activities that exclusively relate to training. Dotted arrows refer to activities those that are only relevant for detection. Finally, the semi-dotted lines denote activities that are relevant for training and detection. In the following we elaborate on the above steps.

3.1 Data Retrieval

To obtain the raw data needed for subsequent mining and detection activities we used a dynamic malware analysis infrastructure from the literature [36]. This infrastructure is an extension of a popular *Malware Analysis Sandbox* (*Component A*) [30] that outputs QDFGs that capture the behavior of samples executed in the sandbox and monitored for a defined period of time.

After submitting a sample to the infrastructure it is executed in one of the sandbox's virtual machines. From that moment onward, a user-mode Windows API monitor [38], that is deployed within each virtual machine, records all Windows API calls issued by all processes running in the system, including the process the executed sample was loaded into. This monitor records name and parameters of each issued Windows API call, along with additional information such as the size of buffers referred to by certain call parameters.

After a pre-defined period of time the monitoring is stopped; recorded API calls are written to a log; and the log is sent to the *Event Parser* (*Component B*). The *Event Parser* interprets each received API call according to its data flow semantics (see Section 2.1), yielding sequences of data flow events that are then fed into the *QDFG Builder* (*Component C*) to finally generate the corresponding QDFGs.

These QDFGs build the data basis for all subsequent mining, training, and detection steps.

3.2 Pattern Mining

Learning is done on a large set of QDFGs that capture the behavior of known malicious and benign samples. An overview of the complete learning procedure is depicted in Figure 2 and consists of a *Pattern Mining*, a *Pattern Matching* or *Feature Generation*, and a *Classifier Training* phase.

The first learning phase extracts interesting patterns from the generated malware QDFGs. As we want to use these patterns to later detect unknown malware, "interesting" here refers to how malware-specific a pattern is in the sense of more likely capturing characteristic malware behavior than benign activities. This "being interesting" will be related to pattern utility. We will later concretize this notion of pattern utility when we introduce our mining approach. As we abstract from low-level behavior using QDFGs, a natural way of obtaining highly characteristic patterns is to employ some sort of supervised graph mining algorithm on the labeled training data that we obtained in the previous step.

Most related malware detection and classification approaches that leverage graph mining on behavior models emphasize pattern frequency [5]–[7], [18], [20], [24], [26], [31]. The idea is to determine the utility of a pattern by how frequently it appears in the training malware set. Properties of the pattern itself, e.g. its structural complexity, in most cases are either ignored or only play a subordinate role. While occurrence frequency is a useful property to determine the utility of a pattern, we argue that considering the structural aspects of the patterns as well might lead to more strongly discriminating patterns. This is backed by results of experiments conducted in the context of molecule mining [21] that indicate that a few slightly less frequent but more complex patterns might be more interesting than many very frequently occurring but less complex ones.

We hypothesize that this also holds for graph patterns in the context of malware detection. If we recall the basic operation principle of frequency-based mining algorithms and consider that most malware for instance loads similar libraries or manipulates the same registry keys to e.g. ensure persistent execution, it becomes likely that employing a frequency-based mining approach on malware behavior graphs probably yields very simple behavior patterns.

Favoring small numbers of complex patterns over larger collections of simple patterns has two advantages. First, as we will later see, there is some correlation between pattern complexity and discriminating capabilities. This is because a complex pattern carries more information than a less complex one. However, this also poses the problem of finding a trade-off between being too specific and thus not generalizing well, and being too generic and thus often accidentally also matching benign graphs.

Second, the number and complexity of patterns directly impact the overall efficiency of the detection phase. As we

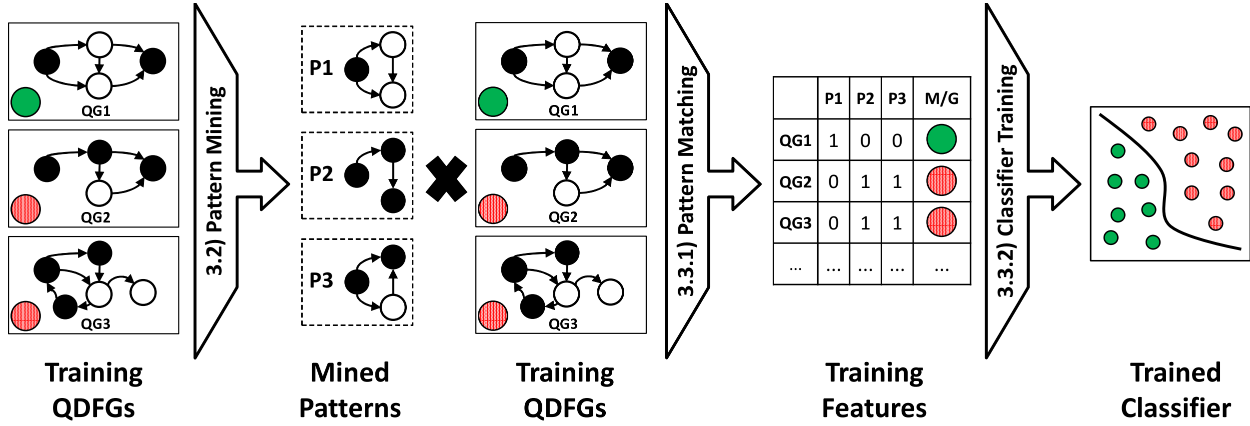

Fig. 2: High-level overview on complete training procedure.

will later see, for classifying a new sample we need to test all patterns on its behavior graph. In the worst case we need to find all isomorphisms between each pattern and all possible sub-graphs of the corresponding sample QDFG. The overall computational effort of evaluating the containment of all patterns in a given graph is thus linear in the amount of the patterns in the worst case.

Furthermore, pattern complexity intuitively impacts the computational effort needed to perform all possible isomorphism checks on QDFG sub-graphs. Consider the example of Figure 3. The pattern *P2* consists of one black and two white nodes, connected by three edges. Looking at all possible sub-graphs of *QG4* we see that there exists only one 3-node sub-graph that has the same number of nodes of the required type as the pattern *P2*, which is a necessary pre-condition for node-induced colored sub-graph isomorphism. In this case we thus only need one isomorphism check to be sure whether and how often *P2* is contained in *QG4*. If *P2* were less complex and for instance consisted of one black and one white node connected by one edge, there would be at least 6 sub-graphs of *QG4* with the same node count and type as *P2* that thus potentially could match *P2*. In this case we would thus need to perform six isomorphism checks instead of one.

Although this example does not generalize to all possible matching scenarios, it shows that compression-based mining algorithms that consider frequency **and** complexity for determining the utility of a pattern might be useful for mining malware behavior. The *Pattern Miner* (*Component D*) therefore implements a compression-based instead of a frequency-based graph mining algorithm. More precisely, we implemented a variant of the Subdue graph mining

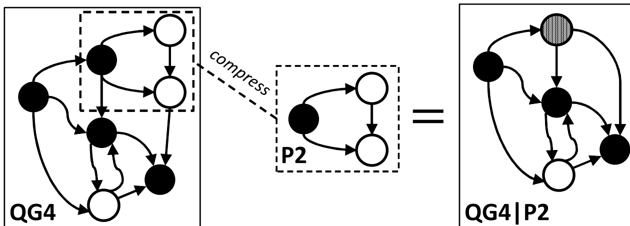

Fig. 3: Example: Graph *QG4* compressed by pattern *P2*.

algorithm proposed in [10] which we customized to our needs and which we will describe in the following.

Subdue is a lazy algorithm that only considers those parts of the pattern search space that are likely to yield interesting patterns. By construction, lazy algorithms usually are faster than exhaustive ones but might miss some patterns. Choosing a lazy rather than an exhaustive algorithm thus imposes a certain effectiveness-efficiency tradeoff. However, as we will later see in the evaluation section (§4.1), this choice is justified as it yields superior efficiency with very competitive effectiveness.

For brevity's sake we will use the terms *positive examples* (T^+) to refer to the malware QDFGs in our training data set, and *negative examples* (T^-) to refer to training QDFGs obtained from benign software. In the following we describe the details of the mining process as well as the scoring functions used for assessing pattern utility, following the algorithm sketched in Listing 1.

3.2.1 Pattern Identification and Application

To determine discriminating patterns in our training set we first define the root nodes of the prospective sub-graph candidates. Recall that in QDFGs process nodes refer to the monitored processes of a system, including the processes that loaded the executed malicious binaries together with their descendants. As these process nodes refer to the only active entities in a system, it is reasonable to initialize sub-graph candidates with process nodes as root nodes. Furthermore, we consider the direct proximity of the process nodes that loaded the malicious binaries. In other words, we

Algorithm 1 Abstracted Subdue Mining Algorithm

procedure MINEPATTERNS
  $PC_0 \leftarrow \bigcup_{g \in T^+}\{((\{n \in g.N : g.\lambda(n, init)\}, \emptyset, g.A, g.\lambda), 0)\}$
  $i \leftarrow 0$
  while $i \leq k$ **do**
    $PC_{i+1} \leftarrow PC_i$
    for each p in PC_i **do**
      $p' \leftarrow extend(p)$
      $PC_{i+1} \leftarrow PC_{i+1} \cup \{(p', S(p'))\}$
    $PC_{i+1} \leftarrow bestscoring(PC_{i+1})$
    $i \leftarrow i + 1$
  return P_k

restrict the set of initial pattern nodes to the set of all initial processes in the entire training set, i.e. all nodes that correspond to the main processes of the executed and analyzed binary samples. In our training data set we respectively mark all process nodes n that loaded the analyzed malicious binaries with a special $\lambda(n, init)$ property.

3.2.1.1 Pattern initialization: We determine the initial set of pattern candidates $PC_0 \subseteq \mathcal{G} \times \mathbb{R}$ by creating one singleton pattern containing an initial process node in the set of all malicious training QDFGs T^+. We index the pattern candidate sets PC_x to denote the iteration of the algorithm that generated it. As each pattern in the candidate set will later be assessed regarding its discriminating utility, each element of a set PC_x is a pair of a pattern and its utility score. The initial one-node patterns contain too little information to be useful for graph discrimination. Their initial utility is therefore estimated by summing the data flows of all neighboring edges (set to 0 in the listing for simplicity's sake). Moreover, since the expansion process (see below) starts with the initial nodes, we discard all nodes from the graph that are not reachable from these initial nodes.

Now that we have established an initial population, with each isomorphic instance of the candidate containing exactly one node and no edges yet, we are set to enter the pattern evaluation and evolution loop.

As the time required to extend and evaluate all possible initial substructures turned out to be prohibitive (especially two-node structures can have hundreds of isomorphic subgraphs), we decided to only consider a fraction of the best-performing instances in $\mathcal{PC}$, for further processing steps. We have empirically determined that the optimal efficiency vs. effectiveness tradeoff is achieved when considering 1% of the initial pattern instances for further extension.

3.2.1.2 Pattern extension: For pattern extension (function $extend$), the n_i best isomorphic pattern instances are extended in each possible direction by adding an additional node and an additional edge from its neighborhood in the QDFG. This yields a new set of pattern candidates that together form the scored pattern candidate set PC_i of iteration i. At this point duplicates are filtered, isomorphic instances are grouped into one pattern, and additional patterns which we dub reduced sub-graphs are added. These latter patterns are obtained by removing the initial node from each mined pattern.

At this point, we consider a pattern candidate's structure. A purely frequency-based mining algorithm would now determine the value of a pattern only based on how often it appears in the positive training graphs (and does not appear in the negative examples). In contrast, our pattern utility assessment strategy considers both, the relative frequency and structure of a candidate pattern. This is, for each pattern $p \in \mathcal{G}$ we determine the pattern's utility through a scoring function S that computes the pattern's utility as real number: $(p, S(p)) \in PC_i$. We consider two different pattern scoring functions S_{MDL} and S_{MDC} and use S where the choice does not matter. The *Minimum Description Length* (*MDL*) scoring function S_{MDL} is the standard scoring function of the original Subdue algorithm and considers both, the frequency a pattern occurs within the training data set and the complexity of the pattern. The *Maximum Data Compression* (*MDC*) scoring function S_{MDC} extends it by also considering quantitative data flow properties encoded in the QDFGs. We will discuss the details of the scoring functions in §3.2.2 and §3.2.3.

After evaluating the utility of each pattern candidate on the entire training set we sort the resulting pattern candidate set w.r.t. the pattern score in descending order and only retain the best patterns for the next iteration (denoted as function *bestscoring* in the listing). As we only consider the n best-performing patterns of a pattern set PC_i for computing the patterns set PC_{i+1} of the next iteration, we essentially perform a heuristic beam search, where n is the size of the search buffer. By construction we thus only follow the best-performing extension branches of the initial singleton-node patterns which significantly cuts down the algorithm's search space.

This process is repeated until a defined maximum pattern complexity k is reached. As a pattern is extended by an edge in each iteration, the maximum number of iterations to be conducted directly limits the maximum allowed pattern complexity. After termination, the algorithm returns the n_b globally best-performing patterns. Note that the members of the final pattern set can be of different complexity as in some situations simple patterns can outperform more complex ones and vice versa.

Finally, after termination, the *Pattern Miner* (*Component D*) writes the final pattern set to the *Pattern Repository* for use by subsequent training and detection steps.

3.2.1.3 Parallelization: To speed up pattern expansion, we can easily distribute the expansion of the different initial singleton pattern instances among different physical processes. Since expansion and evaluation of the different instances are independent of each other we can almost arbitrarily parallelize the algorithm, even among different machines in a cloud or grid setting.

To this end, the pattern miner component spawns a new process for each initial singleton pattern instance and continues to expand it. After all expansions have finished, the main component of the pattern miner combines the results of all processes and filters out the duplicates. Even though there might be an overlap between pattern candidates in different processes during the expansion, we found it less computationally expensive to remove the duplicates at the end rather than running more complex process synchronization mechanisms that avoid redundant pattern exploration (such as those suggested by the original Subdue authors [13]). Using this distribution paradigm, the parallelization of the mining approach in principal is thus only constrained by the number of initial singleton pattern instances and available computational resources. Still, independent of the employed scoring function and parallelization, determining the utility of a pattern candidate implies evaluating its occurrence within the positive and negative training graphs. Alternatively, we have decided to distribute the computation in an even simpler manner, namely we calculate each instance of the k-fold cross validation on a separate thread.

3.2.1.4 Checking subgraph isomorphisms: Checking the occurrence of a pattern $p = (N, E, A, \lambda) \in \mathcal{G}$ in a given training QDFG $G = (N', E', A', \lambda') \in \mathcal{G}$ boils down to the node-induced sub-graph isomorphism problem. A pattern p is *sub-graph isomorphic* to G, i.e. $p \cong_s G$, iff:

1) there exists a sub-graph g of G, i.e. $g \subseteq G$, with:
   - $g = (N'', E'', A'', \lambda'') \in \mathcal{G}$
   - $N'' \subseteq N'$
   - $E'' \subseteq (E' \cap (N'' \times N''))$
   - $\lambda'' \subseteq \lambda'$;
2) and p is isomorphic to g, denoted $p \cong g$:
   - $\exists f : N \rightarrow N''$ with f being bijective
   - $\forall n_1, n_2 \in N'' : (n_1, n_2) \in E \iff (f(n_1), f(n_2)) \in E''$
   - $\forall n \in N, \forall a \in A : \lambda(n, a) = \lambda(f(n), a)$

Using this definition we introduce function $sg : \mathcal{G} \times \mathcal{G} \rightarrow 2^{\mathcal{G}}$ that returns all sub-graphs g of a graph G that are isomorphic to a pattern p: $sg(p, G) = \{g \in \mathcal{G} : g \subseteq G \wedge p \cong g\}$.

Computing sub-graph isomorphisms is the most expensive step as the underlying problem is known to be NP-complete [11]. In practice, the average-case computational complexity of the employed VF2 algorithm is only quadratic in the maximum complexity of the training graph and the pattern candidate [11].

Unfortunately the pattern expansion in all possible directions, at least in theory, demands that we check the sub-graph isomorphism relation for each possible pattern-candidate training-graph pair. The theoretical worst-case computational effort for one entire mining run thus is exponential in the number of expansion iterations, i.e. is in $\mathcal{O}(v^2 \cdot (|T^+| + |T^-|) \cdot |PC_0|^k)$ with v being the number of nodes of the most complex training graph, i.e. $v = max(\{|g.N| : g \in T^+ \cup T^-\})$.

In practice, it is not possible to extend each pattern candidate to an arbitrary depth. Therefore, the exponential factor in practice is rather c^k with $c \ll |PC_0|$. Fortunately, the maximum number of expansion iterations k is constant and can usually chosen to be rather small. This further reduces the average case computational complexity. While we have little influence on the maximum training graph complexity v, we can further cut down the overall complexity of the algorithm by only considering a smaller sub-set of the entire training data set for determining the pattern scores. To this end we introduce the approximation ratio σ, which describes the fraction of training graphs that are considered for the isomorphism checks. Each time we need to evaluate the utility of a pattern, i.e. we need to calculate the isomorphic sub-graphs in the training set $T = T^+ \cup T^-$, we only consider a random sub-set of size $\sigma \cdot |T|$. In several independent experiments with differently-sized data sets we empirically determined a approximation ratio of $\sigma = 25\%$ to yield the best cost-benefit ratio (see §4.1).

By sub-sampling the training set and thus not considering all training graphs for assessing the utility of a pattern we certainly compromise generalizability of the computed pattern scores. However, as our evaluation shows (see §4), not too aggressively down-sampling the training set has barely any effect on the overall detection accuracy. In contrast, it does have a significant effect on mining efficiency. We explain this by most malware samples from the same family behaving fairly similarly. Down-sampling a sufficiently large training data set with a fairly uniform distribution of malware families thus mainly removes redundant behavior, resulting in little effect on the respectively computed pattern utility scores.

We can now turn our attention to the details of the scoring function, i.e., the pattern utility computation.

3.2.2 Minimum Description Length (MDL)

In the context of compression-based graph mining, scoring functions are used to express a pattern's utility in "describing well" a larger set of graphs.

The standard scoring function of Subdue is *Minimum Description Length* (MDL). The idea behind MDL goes back to the work of Rissanen [34] who, in essence, postulates that the optimal description for a set of data items is the one that encompasses as many and complex commonalities within the data set as possible. In this sense, an optimal description compresses the data set as well as possible.

Applied to graph mining this means that a pattern is interesting—it describes well a set of graphs—if by removing it from each training graph (i.e. removing all isomorphic sub-graphs), the cumulative complexity of the graph set is reduced. If we encode the graph structure in bits, a good pattern thus compresses the graph set so that describing it after compression needs significantly fewer bits than before.

An optimal pattern p (w.r.t. a set of graphs G) thus minimizes the term $\sum_{g \in G} DL(p) + DL(g|p)$, where $g|p$ denotes the graph we obtain when compressing g with p, and DL is a function to encode the structure of a graph (e.g. its edges and nodes) in bits. DL is defined as $DL(g) = DL_N(g) + DL_E(g)$. $DL_N(g)$ computes the number of bits required to encode the nodes and node labels of a QDFG g; $DL_E(g)$ computes the number of bits needed to represent the interconnection of nodes via edges.

Let f be the above bijective function between nodes that defines the subgraph isomorphism between pattern p and graph G. Compressing G with pattern p replaces the image of the pattern in the graph with a new node $n' \in \overline{N} \setminus (G.N \cup p.N)$ and adds relevant edges from and to n':

$$\begin{aligned} G|p = (\ & \{n'\} \cup G.N \setminus \{f(n) : n \in p.N\}, \\ & G.E \setminus \{(n_1, n_2) : \exists n_1'.n_1 = f(n_1') \vee \exists n_2'.n_2 = f(n_2')\} \\ & \cup \{(n_1, n') : n_1 \in G.N \setminus \{f(n) : n \in p.N\} \\ & \qquad \wedge \exists n_2.(n_1, f(n_2)) \in G.E\} \\ & \cup \{(n', n_2) : n_2 \in G.N \setminus \{f(n) : n \in p.N\} \\ & \qquad \wedge \exists n_1.(f(n_1), n_2) \in G.E\}, \\ & G.A, G.\lambda\ ). \end{aligned} \tag{3}$$

The binary encoding of the nodes of a QDFG G and their connection via edges is then done as follows:

- $DL_N(G) = |G.N| \cdot \log_2 |G.N| + |G.N| \cdot (\log_2 |G.A|) + \log_2 |\mathbf{cod}(G.\lambda)|)$ encodes the set of nodes N of G and their respective labeling functions.
- $DL_E(G) = \sum_{e \in G.E} (2 \cdot \log_2 |G.N|)$ encodes the edges of a QDFG as list of tuples of node references. Since we do not use edge labels for the node-induced sub-graph isomorphism check, we do not need to encode them.

Graph compression is then done by replacing all instances of a pattern p in g, i.e. all sub-graphs in g isomorphic to p, with a single node while retaining the original edges. A simple compression example is shown in Fig. 3.

Finally, we do not need only to consider a pattern's compression capabilities on the positive graph samples but also its compression effects on the negative examples. A good pattern in this sense should strongly compress the

positive examples while barely compressing the negative ones. Our MDL scoring function is defined by:

$$S_{MDL}(p) = \frac{\sum\limits_{g^+ \in T^+} DL(g^+) + \sum\limits_{g^- \in T^-} DL(g^-)}{DL(p) + \sum\limits_{g^+ \in T^+} DL(g^+|p) + \sum\limits_{g^- \in T^-} (DL(g^-) - DL(g^-|p))} \quad (4)$$

3.2.3 Maximum Data Compression (MDC)

The MDL scoring function allows us to consider more information than a purely frequency-based mining approach. Considering the structural complexity of pattern candidates for assessing their utility potentially generates patterns of higher utility. Recent work on malware detection with quantitative data flow analysis has shown that explicitly considering quantitative data flow aspects for training substantially improves detection accuracy [36], [37]. The main reason is that there exists a correlation between edges that relate to relatively high data flows and the malicious behavior encoded in a QDFG.

We therefore incorporate information about the data flows encoded in the training QDFGs into a more advanced pattern scoring function using the *Maximum Data Compression* (MDC) pattern scoring function. MDC, like MDL, considers both pattern frequency and pattern complexity. However, unlike MDL that only considers the *structural complexity* of a pattern to assess its utility, MDC also considers the cumulative *data flow complexity* of a pattern candidate.

Instead of counting how many edges are removed when compressing training graphs by pattern candidates, we measure how much data is compressed when removing those edges. We calculate the fraction of the total amount of data associated with the removed edges with respect to the total amount of data of all edges of the uncompressed graph.

The MDC score of a pattern $S_{MDC}(p)$ thus calculates the relative amount of data that is encompassed by the edges removed from all training graphs when removing all subgraphs that are isomorphic to the pattern. To this end we introduce the function $QC(p, g)$ that returns the fraction of data that is removed by compressing a QDFG q with a pattern p, formally defined by:

$$QC(p, g) = \frac{\sum_{e \in g.E \setminus (g|p).E} g.\lambda(e, size)}{\sum_{e \in g.E} g.\lambda(e, size)} \quad (5)$$

Assigning a positive score to patterns that data-compress well the positive examples and penalizing the data compression of negative examples, the MDC scoring function is defined as:

$$S_{MDC}(p) = \sum_{t^+ \in T^+} QC(t^+, g) - \sum_{t^+ \in T^-} QC(t^-, g) \quad (6)$$

In sum, our approach features two distinct pattern scoring functions to assess the utility of mined pattern candidates. The S_{MDL} scoring function considers the structural complexity of a pattern candidate to evaluate its expected utility. In addition, the S_{MDC} scoring function also considers a pattern's inherent data flow complexity.

3.3 Pattern Matching and Classifier Training

Now that we have established a basis for discriminating malware behavior patterns, we could directly use them for the classification of unknown samples and, for instance, flag a sample as malicious if it contains one or more of the mined malware patterns. However, as we use a soft-computing methodology, i.e. a graph mining algorithm, we cannot be sure that the mined patterns necessarily generalize to malware different from the ones contained in the training set. This specifically means that there is a non-negligible risk of mined malware patterns appearing in unknown benign software samples. Following such a naive detection strategy likely leads to many false positives.

Furthermore, our mining algorithm is lazy as it a) does not evaluate the entire search space and b) by construction can output malware patterns that also appear in training graphs of benign software but overall are still more specific to known malware behavior in the training set. This suggests that implementing a naive pattern matching strategy that only looks for the existence of a malware pattern in unknown samples likely will yield sub-optimal effectiveness.

We therefore introduce a second learning phase that anticipates the potential presence of malware patterns in benign software. The idea is to match the mined malware patterns from the first step with the graphs of all malware and benign software from the training set, and use this information to train a classifier. This way, the classifier learns complex relations between the occurrence of different combinations of behavior patterns in a graph. This to some extent compensates the effects of our mining step potentially returning patterns that not exclusively appear in malware.

In the following we elaborate on the *pattern matching* step that yields the feature on top of which we train a supervised machine learning classifier in the *classifier training* step.

3.3.1 Pattern Matching

The mined patterns together with their utility scores provide an aggregated view of the distribution of occurrences on the different malicious and benign samples in the training set. Unfortunately, we need more precise pattern matching information in order to build more complex detection models that relate the occurrence of *different patterns* in known benign software and malware graphs for a joined final classification decision.

We hence propose to again evaluate all mined patterns on all (positive and negative) examples in the training set. This is done by the *Pattern Matcher* (*Component E*) that for each training graph searches for all sub-graphs isomorphic to the mined patterns. The pattern matcher provides us with a mapping between the mined patterns and the respective isomorphic sub-graphs in the training graphs. This mapping allows us to answer questions such es whether, or how often, a specific pattern matched a certain training graph.

For each pattern / training graph pair we record a) how many sub-graphs in the training example were isomorphic to the pattern (*Frequency Match*), and b) which fraction of the overall data of the training graph can be “compressed” by removing all matching sub-graphs (*Compression Match*).

The matching information is generated by checking subgraph isomorphism between all patterns $p \in PC_k$ and all training graphs $g \in T$ and then evaluating the functions M_F and M_C on the results:

- *Frequency Match*: $M_F(p, g) = |sg(p, g)|$
- *Compression Match*: $M_C(p, g) = QC(p, g)$

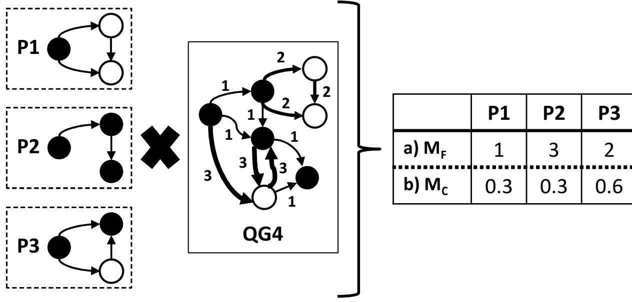

Fig. 4: Example for a matching procedure (where M_F is a frequency match and M_C is a compression match).

While the frequency match is meant to be used in conjunction with patterns that were mined using GSpan and MDL, the (quantitative data) compression matching is the natural dual to the MDC scoring function and thus is used together with patterns mined using MDC scoring. In the following, if the exact matching function does not matter, we will simply use the symbol M.

Figure 4 shows the differences when matching three different patterns with a simple training graph. The two scoring functions applied to the same matching scenario do not necessarily always concur in the distribution of weights of the matched pattern. For instance, the frequency matching function in this simple example considers the pattern P2 to be the most important one and assigns it with the highest value in the result row. In contrast, the compression matching function considers P3 to be the most important pattern. This means that in fact classifiers trained using different matching functions can come to different classifications when facing unknown samples.

3.3.2 Classifier Training

Having matched all mined patterns on the positive and negative training graphs, we can train a supervised machine learning classifier. We organize the matching information obtained in the previous step in a training feature matrix (*Feature Generator*, *Component F*). Each column of this table captures the matching information for one graph t_i from the training set T. Each element of the column is obtained by evaluating one of the previously mentioned matching functions for one specific pattern p_i from the final mined pattern set PC_k. As our training set is labeled—the training set's QDFGs represent the behavior of known malware or goodware—we label each feature vector with the *class* (benign vs. malicious) of the respective training graph.

Remember that PC_k is the result of the pattern mining step that contains n best patterns. The training data generation function is defined by:

$$gen(PC_k, T) := \begin{bmatrix} M(p_1, t_1) & \cdots & M(p_n, t_1) & class(t_1) \\ \vdots & \ddots & \vdots & \vdots \\ M(p_1, t_{|T|}) & \cdots & M(p_n, t_{|T|}) & class(t_{|T|}) \end{bmatrix}$$

On this training data we finally train a standard supervised machine learning classifier. Preliminary evaluations on training data obtained from smaller training sets with different supervised machine learning algorithms (k-nearest neighbor, support vector machine, etc.) indicated that meta learners using decision trees, i.e. RandomForest [3] or ExtraTrees [15] yielded particularly good classification results. We chose to use a standard ExtraTrees algorithm as classifier.

The final classification model thus consists of the trained classifier and a set of patterns that were used for training.

3.4 Detection

To classify an unknown sample we first execute the sample in the analysis sandbox component and follow the data retrieval process described in §3.1 to obtain a QDFG that represents the sample's captured behavior.

Subsequently, we take the set of detection patterns contained in the previously generated classification model and match them against the QDFG of the sample to be classified. As described in §3.3 we then convert the matching results into a feature vector, using the same matching function that was used for generating the training features the classifier contained in the classification model was trained on.

Finally, we pass the obtained feature vector to the classifier which returns a classification based on the similarity of the matching profile with matching profiles of known malware and benign software from the training set.

4 Evaluation

To evaluate the effectiveness of our approach we implemented a prototype of the architecture sketched in Figure 1. For the *Malware Sandbox* component we used a modified Cuckoo sandbox[2], with its guests running Windows 7 SP1.

To allow replication and comparability of our results, for our evaluations we used the Malicia malware data set from Nappa et al. [28], composed of a representative and diverse selection of malware from 12 different malware families, including samples from the Zeus, SpyEye, and Cleaman family. We further populated this set of known malicious samples with known benign samples including standard Windows executables like Paint, Wordpad, or Explorer, as diverse software samples obtained from the Internet. Executing these samples in the malware sandbox then yielded a total set of 6994 malware and 513 benign QDFGs.

The following experiments were performed on an Intel Xeon sever running Ubuntu 14.10, powered by 6 physical 3.5 GHz cores and 128 GB of RAM.

4.1 Effectiveness

For evaluating the effectiveness of our approach we are interested in investigating two research hypotheses:

- *H1*: Compression-based graph mining for malware pattern extraction yields better malware detection accuracy than frequency-based mining.
- *H2*: Quantitative data flow information for compression yields better detection accuracy than the structural complexity of patterns alone.

For investigating hypothesis *H1*, which we consider the main hypothesis that motivates this work, we aimed at comparing the detection effectiveness of our approach with other detection approaches that use frequency-based graph mining [5]–[7], [18], [20], [24], [26], [31].

2. http://www.cuckoosandbox.org/

	a) $GSpan(M_F)$	b) $MDL(M_F)$	c) $MDC(M_C)$
Avg. AUC (Std. Dev. σ)	0.953 (0.012)	0.943 (0.013)	0.993 (0.001)
Avg. BDR (Std. Dev. σ)	0.159 (0.065)	0.754 (0.067)	0.967 (0.008)
Avg. F1-Score (Std. Dev. σ)	0.983 (0.005)	0.867 (0.004)	0.997 (0.005)
Avg. Precision (Std. Dev. σ)	0.982 (0.004)	0.780 (0.000)	0.992 (0.000)
Avg. Recall (Std. Dev. σ)	0.986 (0.005)	0.978 (0.004)	0.990 (0.000)

TABLE 1: Effectiveness quality metrics.

Unfortunately, for most of these approaches implementations and data sets were either not publicly available or the contacted authors could not provide us with operational prototypes. For our experiments we thus fell back on re-implementing the core mining concept shared by most of the approaches. This is, at least for the ones that mentioned the used graph mining algorithms we could compare the detection performance by substituting our compression-based mining component with a component implementing the respective mining algorithm. A closer look at related work that uses graph mining revealed that the frequency-based GSpan [40] mining algorithm was, by far, the most commonly used [18], [20], [24], [26], [31]. Most other used algorithms were structurally very similar to GSpan.

Another major reason for us to (at least partially) re-implement related approaches instead of directly comparing the numbers presented in the respective papers is given by the well-known fact that machine learning based approaches are highly sensitive to the number and nature of used training samples. A direct comparison of numbers produced on different evaluation data sets would thus likely lead to heavily biased conclusions.

For assessing hypothesis *H1* we thus evaluated our approach on the evaluation data set: once with our compression-based pattern mining component using MDL scoring for pattern utility evaluation; and once replacing it with a component that interfaces to the original publicly available implementation of the frequency-based GSpan algorithm[3]. The individual evaluations were performed through typical 10-fold cross validation experiments. For each fold of the experiment we trained the approach on 90% of the evaluation data, i.e. mined interesting patterns; trained the final classifier on this set; and used the resulting detection model to classify the remaining 10% of the data.

We evaluated the following combinations of mining algorithms and matching functions as discussed in §3.2 and §3.3: a) frequency-based mining using GSpan and frequency matching; b) compression-based mining using Subdue with MDL scoring function and frequency matching; c) compression-based mining using Subdue with MDC scoring function and (data) compression matching. We chose these combinations of mining and matching metrics for two reasons. First, setting a) covers most of the aforementioned malware detection approaches from that literature that use frequency-based mining. Second, the combinations of mining scoring and matching functions of setting b) and c) are the the natural duals of each other.

Malware often uses randomized names for dropped files. Extracted patterns that use the full name of a file would likely be too restrictive and only match very few malware instances. We thus conducted all aforementioned experiments using only the file extensions part of the file node labels for label equivalence matching instead of the full file name.

To avoid biasing the comparison due to unequal baseline data sets we set the sub-sampling ratio δ of our mining algorithm and configured our GSpan wrapper to only use 25% of the training part of each fold for isomorphism checks. Considering that this sub-sampling was done randomly we furthermore repeated each cross validation experiment ten times to weed out noise and cut out random side-effects.

To express the aggregated effectiveness of the approaches, we computed four standard quality metrics: Area under ROC Curve (AUC), F1-Score, Precision, and Recall. True positives in this context refer to malware samples that have been correctly classified as malicious, true negatives to benign software samples that were correctly classified as benign, false positives to benign samples incorrectly classified as malicious, and false negatives to malware samples that were mistakenly labeled as benign.

To evaluate our approach's effectiveness in a more operational perspective we furthermore measured its *Best-case Detection Rate* (*BDR*), which captures the best-case true positive rate that can be achieved when fixing the maximum acceptable false positive rate to a threshold of 0.5%, which we deem reasonable in an operational context. In other words, the BDR is the value of the ROC function at 0.005. In contrast to the AUC and F1-Score that, although better capturing the overall quality of a classifier, do not have an obvious operational interpretation, the BDR gives a better idea of the operational detection accuracy. Consider a medium-to-large sized company environment with 10,000 to be classified email attachments per day. Using *BDR* would translate to the question of how many real malicious attachments we can correctly identify as malware when accepting an upper bound of 50 emails wrongly put into quarantine due to incorrect malware alerts.

The average results and the respective standard deviations of the experiment runs are depicted in Table 1. We can see that the results support our hypothesis *H1* in that, at least on our evaluation data set, the best compression-based mining approach outperformed the frequency-based mining using GSpan. Even if we factor out random effects, i.e. take the standard deviations into account, we at least perform $\frac{AUC_{MDC(M_C)}}{AUC_{GSpan(M_F)}} = \frac{0.993}{0.953} \approx 4\%$ better in terms of AUC when using data compression-based mining instead of frequency-based mining. At least concerning the overall effectiveness in terms of AUC, however, structural compression-based mining performs worse than frequency-based mining with GSpan. Looking at the individual detec-

3. https://www.cs.ucsb.edu/~xyan/software/gSpan.htm

tion rates when fixing the maximum accepted false positive rate to 0.5% (BDR), the effectiveness differences between the approaches become more apparent. Concerning the BDR, the quantitative data flow compression-based yielded more than $\frac{BDR_{MDC(M_C)}}{BDR_{GSpan(M_F)}} = \frac{0.967}{0.159} \approx 600\%$ better results than the frequency-based GSpan approach. This means, in particular when targeting low false positive classification rates, our quantitative data flow compression-based mining approach significantly outperforms frequency-based mining (*H1*).

If we look at the individual differences in effectiveness between purely structural compression-based mining and the experiments where we considered quantitative data flows for mining and matching, we see that quantitative data flow compression-based mining on average yields $\frac{MDC(M_C)}{MDL(M_F)} = \frac{0.993}{0.943} \approx 5\%$ better overall effectiveness, i.e. AUC. Again looking at the effectiveness for low false positive rates, data compression-based mining yields $\frac{BDR_{MDC(M_C)}}{BDR_{MDL(M_F)}} = \frac{0.967}{0.754} \approx 30\%$ better detection rates than when only considering structural compression.

These findings support our hypothesis *H2* in that considering quantitative data flow aspects for mining indeed seems to improve the quality of mined patterns and overall accuracy of respectively devised detection models.

4.2 Efficiency

For efficiency considerations, we differentiate between training and detection overheads. The training time encompasses the computational effort for mining and scoring a set of detection patterns, as well as training a supervised machine learning classifier on them. The detection phase consists of matching the patterns against the testing graphs and evaluating the obtained feature vector on the trained classifier.

GSpan makes use of graph encoding and tries to avoid expensive isomorphism checks. Our compression-based approach, in contrast, heavily relies on graph-isomorphism verification. We hence expected the frequency-based GSpan approach to outperform ours in terms of time efficiency. We thus expected that the gain in effectiveness comes at the cost of efficiency.

Table 2 summarizes the average training and mining times of the different approaches. The experiment results confirmed this assumption in that the frequency-based mining with GSpan was more than one order of magnitude faster than the best compression-based mining experiment. Frequency-based mining using GSpan was almost $\frac{T_{MDC}}{T_{GSpan}} = \frac{133.85\,s}{13.20\,s} \approx 10$ times faster than compression-based mining using MDC and more than $\frac{T_{MDL}}{T_{GSpan}} = \frac{3408.00\,s}{13.20\,s} \approx 260$ times faster than MDL. We explain the difference in performance between MDL and MDC with the same argument as we explain their effectiveness difference: MDC likely is better able to early prune useless pattern candidates from the search space and thus needs to perform significantly less isomorphism checks than MDL.

For assessing the detection overhead we measured the time for matching patterns of a detection model against unknown QDFGs of different size. From the results depicted in Figure 5 we see that the overall detection time, at least on our evaluation set, seems to linearly increase with the complexity of the to be classified QDFGs and ranges from 6ms to almost 4200 ms. On average, classifying an unknown sample took 102 ms.

4.3 Threats to Validity

While we put considerable effort to ensure the sound evaluation or our approach and comparison with related mining approaches, there remain threats to the generalizability of the gained insights.

Firstly, we make use of probabilistic soft-computing algorithms whose overall effectiveness and accuracy are well-known to depend on the amount and structure of the data they are trained on. While we used a publicly available data set for our evaluations, composed of a diverse range of malware samples from different families as well as a diverse selection of widely used benign software, we can only safely claim that our approach works well with respect to this data set. In other words, because our approach is learning-based, i.e. infers patterns and their interrelation from the training data to predict classification of unknown data, its effectivness naturally depends on the quality and diversity of the used training data. Although we used a well-established data set for training and evaluation [28], the presented insights have to be interpreted with this potential threat to generalizability in mind.

Furthermore, supervised machine learning algorithms can suffer from over-generalization or over-fitting to the training data. To compensate this threat we consistently employed repeated 10-fold cross validation experiments for evaluating our research hypothesis and used a supervised learning algorithm, i.e. ExtraTrees [15], that is known to be rather robust towards over-fitting to the training data. Moreover, further experiments with nine different classification algorithms, i.e. Extra Trees, k-NN, QDA, LDA, GaussianNB, AdaBoost, SVN, Decision Trees, and Random Forest, did not reveal a significant impact of the used classification algorithm on the absolute accuracy of the overall approach.

In addition to these more statistics-related threats to validity we also potentially suffer from the same generalization issues as most dynamic malware analysis approaches do in that we do not tackle the issue of dormant behavior or in general environment-sensitive malware. Together with

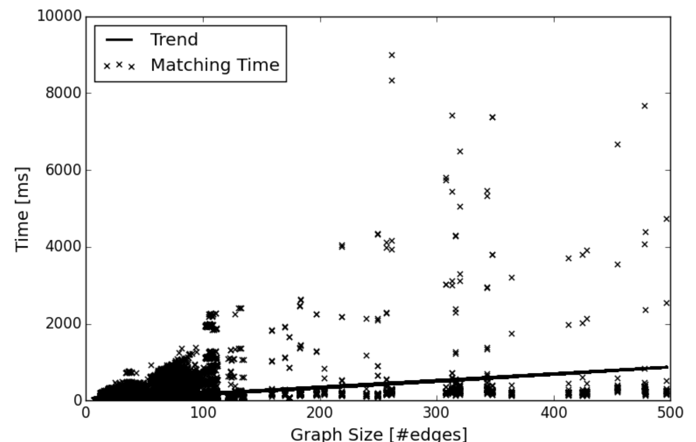

Fig. 5: Detection time vs. QDFG size.

	a) $GSpan$	b) MDL	c) MDC
Training Time (Std. Dev. σ)	13.20 s (0.10 s)	3408.0 s (555.52 s)	133.85 s (3.58 s)

TABLE 2: Average training efficiency.

the fact that our malware analysis environment uses a user-mode monitor for capturing WindowsAPI calls and thus by construction cannot profile the behavior of more advanced kernel-level rootkits this imposes the risk that our approach does not work for kernel-based malware.

Finally, we acknowledge that we introduced a certain bias in the comparison between our compression-based mining approach and frequency-based mining approaches in that we could not directly compare the approaches themselves but only indirectly compare them based on a comparison of the used mining algorithms. Considering that none of the comparison approaches offer publicly-available implementations or data sets it was impossible for us to do so and this indirect comparison on the same standardized data set was the closest we could get towards a fair comparison. Nevertheless, there is a strong conceptual similarity between those approaches and they often build on top of similar graph mining algorithms. We thus argue that our evaluation results nevertheless give important insights into the general utility of graph mining for behavior-based malware detection.

5 Related Work

Behavior-based malware detection received considerable attention in the past decade. We focus on related work that, like us, uses graphs and graph analysis to model and extract malware behavior.

Christodorescu et al. [8] were among the first to propose generating detection patterns by mining the behavior, represented as graphs, present in malware but not in benign software. They do so by looking for minimal sub-graphs of system call dependency graphs of known malware that are not contained in benign software. By this the utility of a mined pattern directly correlates with how often it appears in the malware and does not appear in the benign set. The structural complexity of the pattern itself is not considered as long as it is minimal. Therefore, this approach falls into our category of frequency-based methods.

Chen et al. [6] improve on this idea by first shrinking the pattern search space by summarizing the to-be-mined system call dependency graphs and only keeping their core behavioral properties. Through this approximation they, like us, avoid an expensive evaluation of the entire search space. They thus also make use of graph compression but only use it to reduce the search space; the pattern utility evaluation is still done entirely frequency-based and does not anticipate pattern complexity aspects. Hence, we also consider their work in the category of frequency-based methods.

The HOLMES detection system proposed by Fredrikson et al. [12] works along the same lines. It also relies on system call dependency graph mining but introduces an aggressive probabilistic sampling of the pattern search space to improve accuracy. They further use concept analysis to combine semantically redundant patterns. In our understanding, their pattern selection does not directly consider the structural complexity of a pattern and we therefore also categorize this work into the frequency-based category.

Besides these works that propose custom graph mining techniques, there exists a considerable body of work that uses standard frequency-based graph mining algorithms like GSPan [18], [20], [24], [26], [31].

We differ from all of those approaches in two main ways: i) we primarily consider the compression capabilities of a pattern to determine its utility, and not its frequency, and ii) we make use of quantitative data flow aspects inherent to system call traces, both of which we showed to improve detection effectiveness. Finally, like recent semantics-based detection approaches [29] we also abstract from raw low-level system calls and thus also suffer less from system call injection attacks than earlier system call centric approaches. Our work relies on more abstract QDFGs that have been shown to be less vulnerable to advanced behavior obfuscation attacks than system call dependency graphs [1], [37].

Park et al. [32], [33] proposed a malware classification method based on so-called HotPaths, i.e. maximum common sub-graphs on kernel object dependency graphs, to capture characteristic behavior of malware families. Their dependency graphs do not take quantitative flows between objects into account. As our evaluations showed, this can have a significant effect on detection accuracy. Even though, in addition, this approach per se does not make use of dedicated graph mining techniques to construct HotPaths and thus does not fit our categorization scheme, we wanted to compare its performance against our approach.

As we were not able to obtain an implementation of the approach from the authors we had to fall back to re-implementing it following the descriptions in the respective publications. We then tried to evaluate this implementation on the same data set that we used for evaluating our approach to assure a fair comparison. Even the most powerful systems that we had access to did not manage to get the algorithm to terminate and deliver HotPaths for our data set. This can be explained by the HotPath calculation process relying on repeated maximum common sub-graph (MCS) calculations, which are known to be NP-complete [14]. Our implementation of their approach turned out to already be computationally infeasible even for few small graphs (≤ 50 nodes). This seems to be in line with a purely complexity-theoretical perspective, and also when comparing our experiences with the practical evaluation of maximum common sub-graph algorithms by Conte et al. [9]. We probably have wrongly implemented their algorithms or did not employ optimizations that went unmentioned in the article.

Recent work proposed the use of graph metrics instead of patterns for malware detection [19], [25], [37]. This differs from our approach in that the respective metric-based detection models lack a clear semantic dimension. This

makes it very hard to human analysts to verify a detection prediction and conduct additional post-detection analysis steps to e.g. get a deeper understanding of the behavioral characteristics of a malware sample. The detection models and classification results generated by our approach, in contrast, conveniently allow for such further-reaching manual analysis steps, which already has been shown [39].

In contrast to previous work that uses patterns on QDFGs for malware detection [36] we use graph mining to extract highly characteristic behavior patterns from a large malware corpus instead of using a fixed selection of manually defined detection patterns. As we could show with our evaluation, this significantly boosts detection accuracy.

The differences with related work can thus be summarized as follows: i) we employ a compression-based mining scheme where most related mining approaches use frequency-based algorithms; and ii) we mine detection patterns instead of manually specifying them like related QDFG pattern-matching do. Both ideas yield superior detection accuracy.

6 Discussion and Conclusion

We introduced a novel approach for behavior-based malware detection that uses *compression-based graph mining*. We propose to exploit *quantitative data flow properties* to extract highly characteristic behavior patterns from collections of known malware. By combining a lazy graph mining technique with a robust machine learning scheme our approach to some extent is able to compensate noisy training and testing data, which is reflected by stable high detection effectiveness in our evaluation experiments.

Our evaluation revealed that compression-based mining can yield patterns that are up to six times more effective than patterns obtained through frequency-based mining methods, which is the common data mining technique employed by related work. This gain in effectiveness comes at the cost of a tenfold increase of the training time.

Recent studies [1], [37] indicate that QDFG-based malware detection approaches often outperform static and more simplistic dynamic malware detection in accurately detecting obfuscated malware. As our evaluations indicate an even better classification accuracy of our approach in comparison with earlier fixed-pattern QDFG-based detection approaches [36], [37], we consider our new approach a significant bar-raiser with respect to the state-of-the-art.

We plan to improve accuracy and efficiency by incorporating malware-specific domain knowledge into the pattern scoring process and e.g. early prune patterns that refer to known benign behavior. Furthermore, we plan to improve efficiency by more tightly combining the mining and matching steps of the training phase and e.g. storing information which patterns matched which training graph in order to avoid redundant isomorphism checks.

References

[1] S. Banescu, T. Wüchner, A. Salem, M. Guggenmos, M. Ochoa, and A. Pretschner. An empirical evaluation framework for malware behavior obfuscation. *MALCON*, 2015.

[2] J.-M. Borello and L. Me. Code obfuscation techniques for metamorphic viruses. *J. in Comp. Virology*, 2008.

[3] L. Breiman. Random forests. *Machine learning*, 2001.

[4] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX*, 2011.

[5] D. H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *SIAM International Conference on Data Mining*, 2011.

[6] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. *Proceedings of the VLDB Endowment*, 2009.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India Software Engineering Conference*, 2008.

[8] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, 2008.

[9] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 2007.

[10] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *J. of Artificial Intelligence Research*, 1994.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence*, 2004.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. *S&P*, 2010.

[13] G. M. Galal, D. J. Cook, and L. B. Holder. Exploiting parallelism in a structural scientific discovery system to improve scalability. *J. of the American Society for Information Science*, 1999.

[14] M. Garey and D. Johnson. Computers and intractability: a guide to the theory of np- completeness. *San Francisco, LA: Freeman*, 1979.

[15] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 2006.

[16] T. Holz, M. Engelberth, and F. Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *ESORICS*, 2009.

[17] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*. 2000.

[18] G. Jacob, R. Hund, C. Kruegel, and T. Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Sec.*, 2011.

[19] J.-w. Jang, J. Woo, J. Yun, and H. K. Kim. Mal-netminer: malware classification based on social network analysis of call graph. In *WWW*, 2014.

[20] F. Karbalaie, A. Sami, and M. Ahmadi. Semantic malware detection by deploying graph mining. *Int. J. of Computer Science Issues*, 2012.

[21] N. S. Ketkar, L. B. Holder, and D. J. Cook. Subdue: Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, 2005.

[22] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. *Information Systems*, 2007.

[23] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: Using system-centric models for malware protection. In *CCS*, 2010.

[24] H. D. Macedo and T. Touili. Mining malware specifications through static reachability analysis. In *ESORICS*. 2013.

[25] W. Mao, Z. Cai, X. Guan, and D. Towsley. Centrality metrics of importance in access behaviors and malware detections. In *ACSAC*, 2014.

[26] F. Martinelli, A. Saracino, and D. Sgandurra. Classifying android malware through subgraph mining. In *Data Privacy Management and Autonomous Spontaneous Security*. 2014.

[27] T. Matsuda, H. Motoda, T. Yoshida, and T. Washio. Mining patterns from structured data by beam-wise graph-based induction. In *Discovery Science*, 2002.

[28] A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting. In *DIMVA*, 2013.

[29] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti. Employing program semantics for malware detection. *Information Forensics and Security*, 2015.

[30] D. Oktavianto and I. Muhardianto. *Cuckoo Malware Analysis*. Packt Pbl. Ltd, 2013.
[31] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer. Extraction of statistically significant malware behaviors. In *ACSAC*, 2013.
[32] Y. Park and D. Reeves. Deriving common malware behavior through graph clustering. *CCS*, 2011.
[33] Y. Park, D. S. Reeves, and M. Stamp. Deriving common malware behavior through graph clustering. *J. on Computers & Security*, 2013.
[34] J. Rissanen. Modeling by shortest data description. *Automatica*, 1978.
[35] T. Washio and H. Motoda. State of the art of graph-based data mining. *ACM SigKDD Explorations Newsletter*, 2003.
[36] T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In *ASIACCS*, 2014.
[37] T. Wüchner, M. Ochoa, and A. Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *DIMVA*, 2015.
[38] T. Wüchner and A. Pretschner. Data loss prevention based on data-driven usage control. In *ISSRE*, 2012.
[39] T. Wüchner, A. Pretschner, and M. Ochoa. Davast: data-centric system level activity visualization. In *VizSec*, 2014.
[40] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining*, 2002.
[41] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, 2010.

Tobias Wüchner received his PhD from Technical University of Munich in 2016 and his M.Sc. and B.Sc. degrees in Computer Science from Technical University of Kaiserslautern, Germany, in 2011 and 2009, respectively. His research interests include information and system security, malware detection, and applied machine learning.

Aleksander Cisłak received a B.Sc. degree in computer science from Lodz University of Technology in 2014 and M.Sc. degree in informatics from Technical University of Munich in 2015. His research interests include string matching algorithms and applied graph theory. He worked as an assistant at TU Munich in the area of malware detection, and he currently works as an assistant at Warsaw University of Technology (Faculty of Mathematics and Information Science), focusing on fuzzy cognitive maps.

Martín Ochoa is an assistant professor at Singapore University of Technology and Design. His research interests include information flow analysis, security testing and software diversity. Before, he was a senior researcher at the Technical University of Munich and a researcher for Siemens. He holds a PhD in computer science from the Technical University of Dortmund, a M.Sc. in Mathematics from the Ludwig Maximilian University of Munich and a B.Sc. in mathematics from La Sapienza University.

Alexander Pretschner is a full professor at Technical University of Munich. Prior appointments include a full professorship at Karlsruhe Institute of Technology, an adjunct associate professorship at the Technical University of Kaiserslautern, a group management position at the Fraunhofer IESE in Kaiserslautern, and a senior researcher's position at ETH Zurich. He holds a PhD from Technical University of Munich, a diploma from RWTH Aachen University, and an M.Sc. degree from the University of Kansas. Research interests include software engineering, specifically security and testing.