Bayesian Networks with Prior Knowledge for Malware Phylogenetics

Diane Oyen
Los Alamos National Laboratory
Los Alamos, NM, USA

Blake Anderson
Cisco Systems, Inc
San Jose, CA, USA

Christine Anderson-Cook
Los Alamos National Laboratory
Los Alamos, NM, USA

Abstract

Malware phylogenetics help cybersecurity experts to quickly understand a new malware sample by placing the new sample in the context of similar samples that have been previously reverse engineered. Recently, researchers have begun using malware code as data to infer directed acyclic graphs (DAG) that model the evolutionary relationships among samples of malware. A DAG is the ideal model for a phylogenetic graph because it includes the merges and branches that are often present in malware evolution. We present a novel Bayesian network discovery algorithm for learning a DAG via statistical inference of conditional dependencies from observed data with an informative prior on the partial ordering of variables. Our approach leverages the information on edge direction that a human can provide and the edge presence inference which data can provide. We give an efficient implementation of the partial-order prior in a Bayesian structure discovery learning algorithm, as well as a related structure prior, showing that both priors meet the local modularity requirement necessary for the efficient Bayesian discovery algorithm. We apply our algorithm to learn phylogenetic graphs on three malicious families and two benign families where the ground truth is known; and show that compared to competing algorithms, our algorithm more accurately identifies directed edges.

Introduction

Advanced persistent threats (APT) are becoming a serious problem for many corporations and government agencies, in addition to malware aimed at a more general audience. APT is characterized as malware that has been tailored to accomplish a specific goal against a specific target. During the life cycle of APT malware, the authors generally follow standard software engineering principles which naturally leads to a phylogenetic graph, a graph demonstrating the evolutionary relationships among the different software versions.

When beginning the process of understanding a new, previously unseen sample of malware, it is advantageous to leverage the information gained from reverse engineering of previously seen members of that instance’s family because techniques learned from related instances can often be applied to a new program instance. A malware family is a group of related malware instances which share a common codebase and exhibit similar functionality (e.g. different branches in a software repository). We focus on extracting a more detailed picture about the relationships among the malware instances within a given family.

Fig. 1 exhibits the proposed output of the ideal phylogenetic algorithm. This figure clearly shows the evolution of the mytob worm as a directed graph. The information present in Fig. 1 is invaluable for a reverse engineer tasked with understanding specific instances within a malware family as well as the general evolution of the family. In this paper, Bayesian network structure learning is used to model the evolution of malware for a given family.

Our previous work built phylogenies using undirected graphs and then added directionality to edges in a post-hoc heuristic (Anderson, Lane, and Hash 2014). In this work, the Bayesian networks used are directed acyclic graphs (DAG), a more natural fit for phylogenies. Directionality is inferred by the learning process, but in many cases it is difficult to infer, therefore prior information is included about the edge
Figure 2: Bayesian network discovery estimates the posterior expectation of edge features over structure space for a toy example. The bar graph shows the posterior probabilities of all configurations of a 3-variable DAG. On the top-right, the expectation for each edge is the sum of posteriors over all DAGs containing that edge. For example, the $B \rightarrow A$ edge appears in many of the high-posterior DAGs and therefore has a relatively high expectation. The final result (bottom-right) arranges the estimated expectations into a real-valued adjacency matrix where entries represent the posterior expectation of each directed edge.

directions, either from human experts or a simple heuristic. This paper introduces a novel approach to combining human knowledge about the ordering of variables into a statistical learning algorithm for Bayesian structure discovery. The learning algorithm with our prior combines the complementary benefits of using statistical data to infer dependencies while leveraging human knowledge about the direction of dependencies.

The informative prior is incorporated into a Bayesian network structure discovery algorithm, which calculates the posterior expectation of structural features. Rather than providing a single network as a solution, the goal is to identify the likelihood of each directed edge, representing a possible conditional dependency present in the data, as shown in Fig. 2. Algorithms for estimating the posterior expectation of Bayesian network features are called network discovery and generally fall into two categories: those that sum over structure space and those that sum over order space. Calculating the expectation over structure-space exactly is infeasible, so approximations and heuristics are used but can be slow and may get stuck in local maxima; yet existing algorithms that include prior knowledge use structure-space (Madigan, York, and Allard 1995; Grzegorczyk and Husmeier 2008). In contrast, we take advantage of more efficient exact and approximate solutions in order-space (Koivisto and Sood 2004), showing that our informative priors can be calculated efficiently. Our primary contribution is a partial-order prior that is specified as a collection of orderings on pairs of variables. We show that our prior has the property of local modularity and therefore can be incorporated into a Bayesian structure discovery algorithm efficiently.

Empirical results show potential improvement in network reconstruction when the informative prior information is included. When applied to discovering malware phylogenetics, we demonstrate that using expert knowledge about the partial-order of variables improves learning of true edges in the Bayesian network.

**Preliminaries**

**Phylogenetic Graphs of Malware**

Most existing malware phylogeny techniques rely on bifurcating tree-based approaches (Darmetko, Jilcott, and Everett 2013; Wagener, Dulaunoy, and others 2008). These methods do not give ancestral relationships to the data. We are interested in the problem of phylogenetic graph reconstruction where the goal is to find a graph, $G = (V, E)$. The vertices of the graph, $V$, are the instances of malware and the edges of the graph, $E$ represent phylogenetic relationships between the data such as “child of” or “parent of”.

The main advantage of the ancestral relationship approach is that the graph explicitly states the phylogenetic relationships between instances of malware. For a reverse engineer, knowing that an instance of malware is similar to known malware, from the same family for example, accelerates the reverse engineering process as the analyst will have a general idea of the mechanisms used by the sample. However, it is more informative to know the set of parents from which a given sample’s functionality is derived. Having malware which is a composite of several other samples is becoming more widespread as malware authors are beginning to use standard software engineering practices, thus making the reuse of code more prevalent.

Prior work using the graphical lasso, an undirected graphical model, showed promising results (Anderson, Lane, and Hash 2014). This paper uses a Bayesian network learning approach because it naturally models the parent-child relationships that we are interested in as a directed acyclic graph.

**Bayesian Network Discovery with Prior Knowledge**

Bayesian networks have long been used as models for combining data with human knowledge. (Heckerman, Geiger, and Chickering 1995) suggest combining human expertise and statistical data in the context of learning Bayesian networks by starting with a human-engineered Bayesian network model as a prior in the maximum a posteriori structure learning algorithm that finds the best model to fit observed data. However, this method first requires the labor intensive process of knowledge engineering to produce a prior model. More recently, (Werhli and Husmeier 2007) propose a more flexible prior on individual edges rather than a complete network, but this has the undesirable effect of setting a prior on the statistical dependencies even though these are relatively easy to estimate from data. The well-known candidate parents algorithm (Friedman, Nachman, and Peér 1999) could be used to incorporate knowledge about partial orders, but it imposes a hard constraint on potential parents rather than a Bayesian prior. Instead, we propose a partial-order prior incorporated into a Bayesian network discovery algorithm.

**Bayesian Network Discovery**

We provide necessary background on key model assumptions in Bayesian network structure discovery algorithms.
Bayesian networks compactly describe joint probability distributions by encoding conditional independencies in multivariate data. A Bayesian network \( B = \{G, \theta \} \) describes the joint probability distribution over \( p \) random variables \( X = \{X_1, X_2, \ldots, X_p\} \), where \( G \) is a directed acyclic graph (DAG) and the conditional probability distributions are parameterized by \( \theta \) (Heckerman, Geiger, and Chickering 1995). An edge \( (X_i, X_j) \) in \( G \) means that the child \( X_j \) is conditionally independent of all non-descendants given its parents \( \pi(X_j) \) where \( X_i \in \pi(X_j) \). We often use the shorthand notation \( \pi_j \) to mean \( \pi(X_j) \), the set of variables that are parents of variable \( j \). The structure of the network, \( G \), is of particular interest in many domains as it is easy to interpret and gives valuable information about the interaction of variables.

Given a limited amount of data, the posterior probability of any network may be quite small. However, summary statistics regarding structural features of networks may have high posterior probability even with limited data (Friedman and Koller 2003). Structural features (such as an edge) can be described by an indicator function \( f \) such that for \( f(G) = 1 \) the feature exists in graph \( G \), otherwise \( f(G) = 0 \). The posterior probability of the feature is equivalent to the expectation of its indicator (see Fig 2 for an illustrative example):

\[
E_{\text{DAG}}(f|D) = \frac{1}{P(D)} \sum_G P(G)P(D|G)f(G). \tag{1}
\]

However, this sum can be intractable, as the number of DAGs is super-exponential in the number of variables.

An important insight to making the expectation tractable is that we could fix the order of the variables. An order, \( \prec \), is a permutation on the indices of the variables \( X_{\prec(1)}, X_{\prec(2)}, \ldots, X_{\prec(p)} \) such that parents must precede children in the order, i.e., \( X_j \) cannot be a parent of \( X_i \) if \( \prec(j) \geq \prec(i) \). Given an order, learning optimal parents for each child factors into local calculations, and summing over DAGs consistent with the order is tractable (Buntine 1991; Cooper and Herskovits 1992). Extending this, we can condition on a node order, and then obtain the unconditional posterior by summing over orders (Friedman and Koller 2003):

\[
E_{\text{ord}}(f|D) = \frac{1}{P(D)} \sum_{\prec} \sum_{G \subseteq \prec} P(\prec) \sum_{G \subseteq \prec} P(G)P(D|G)f(G). \tag{2}
\]

Importantly, these two formulations for \( E(f|D) \) are not the same, as most DAGs, \( G \), will be consistent with multiple orders, \( \prec \). Typically, this formulation produces an acceptable bias in favor of simpler structures.

(Koivisto and Sood 2004) give an efficient dynamic programming method for calculating Eq 2. The approach is rather involved, so we summarize only the key points here. We describe the modularity assumptions which are critical for determining which priors we can use:

1. **Parameter modularity:** Modularity of the Bayesian network parameters must hold, \( P(\theta|G) = \prod_{i=1}^{p} P(\theta_{i,\pi_i} | \pi_i) \) and \( P(X=x|G) = \prod_{i=1}^{p} P(x_i|\pi_i, \theta_{i,\pi_i}) \).

2. **Structure prior modularity:** The network model prior must be modular so that \( P(G,\prec) = c \prod_{i=1}^{m} q_i(U_i) q_i'(\pi_i) \), where \( U_i \) is the set of variables preceding \( X_i \) in the order \( \prec \) (potential parents of \( X_i \)) and \( c \) is a normalization constant.

3. **Feature modularity:** The features must be modular, \( f(G) = \prod_{i=1}^{p} f_i(\pi(X_i)) \) where \( \pi(X_i) \) is the parent set of variable \( i \).

The most common feature to look for is a directed edge \( u \rightarrow v \) s.t. \( f = 1 \) if \( X_u \in \pi_v \), which is clearly modular. If these modularity assumptions hold, then the likelihood over order space factors into local calculations as shown in Eq 3 (Koivisto and Sood 2004).

\[
E(f|D) = \frac{1}{Z} \sum_{\prec} \prod_{i=1}^{p} q_i(U_i) \left[ \sum_{\pi_i \subseteq U_i} q_i'(\pi_i) P(x_i|\pi_i) f_i(\pi_i) \right] \tag{3}
\]

where \( \pi_i = \pi(X_i) \) is the parent set of variable \( i \).

The unconditional posterior for the features is obtained by summing over orders, using the following steps:

1. Calculate family scores: \( \beta(\pi_i) = P(\pi_i) P(x_i|\pi_i) f_i(\pi_i) \) for each node \( i \) and potential set of parents \( \pi_i \).

2. Calculate local contribution of each subset \( U \subseteq V \setminus \{i\} \) of potential parents of \( i \): \( \alpha_i(U) = \sum_{\pi_i \subseteq U} P(\pi_i) P(x_i|\pi_i) f_i(\pi_i) \).

3. Sum over the subset lattice of the various \( U_i \) to obtain the sum over orders \( \prec \).

The maximum number of parents allowed for any child is typically fixed to a small natural number, \( r \). The computational complexity for calculating the exact expectations of all features is \( O(p2^p + p^{r+1}C(n)) \), where \( C(n) \) is the cost of calculating each family score from \( n \) observed samples. For large networks, roughly \( p > 30 \), the exponential term is intractable. In these cases, MCMC simulations give an approximation to the posterior expectations, so that \( E(f|D) \approx \frac{1}{T} \sum_{t=1}^{T} P(\pi_t) P(D|\pi_t) \) for sampled from order space (Friedman and Koller 2003) or partial orders (Niinimaki, Parviainen, and Koivisto 2011).

**Bayesian Network Discovery with Informative Priors**

We propose two different methods for incorporating prior knowledge into order-space Bayesian discovery algorithms. Our first method is to place a prior on individual child-parent relationships (or edges), as has been explored in structure-space algorithms previously. The second method is to place a prior on pairwise order relationships (partial orders). Partial order priors have not been previously studied, yet there are two advantages. Partial order priors are easy to implement in the efficient order-space discovery algorithm and such a prior can include the kind of prior knowledge that is not discoverable directly from the data.

**Structure Priors**

Following (Werhli and Husmeier 2007) we consider a prior on structures based on expert knowledge about the parents...
of a particular variable, so that the structure prior \( q_i'(\pi_i) \) in Eq 3 is defined to be:

\[
q_i'(\pi_i) = \lambda e^{-\lambda(\sum_{j=1}^{p} I(j \in \pi_i) - P(f_{ij}))},
\]

where \( \lambda \geq 0 \) is a hyper-parameter, often called the temperature, that determines the strength of the prior; \( P(f_{ij}) \in [0, 1] \) is the prior belief of the expectation of edge \( X_i \rightarrow X_j \) as given by human knowledge or some other source of information. For \( \lambda = 0 \), the prior would be uninformative giving equal probability to all sets of parents. This prior was previously used within a structure-space Bayesian discovery algorithm (Werhli and Husmeier 2007).

Incorporating this structure prior into the order-space dynamic programming algorithm, happens during the calculation of the family scores:

\[
\beta_i(\pi_i) = q_i'(\pi_i)p(x_i|x_{\pi_i})f_i(\pi_i) = p(x_i|x_{\pi_i})f_i(\pi_i)e^{-\lambda(\sum_{j=1}^{p} I(j \in \pi_i) - P(f_{ij}))} = p(x_i|x_{\pi_i})f_i(\pi_i)e^{-\lambda(C'_i + |\pi_i| - 2 \sum_{j \in \pi_i} P(f_{ij}))},
\]

where \( C'_i = \sum_{j=1}^{p} P(f_{ij}) \) is a pre-computed sum and \( |\pi_i| \) is the number of elements in set \( \pi_i \). There are \( p^{r+1} \) of these \( \beta \) family scores to calculate, where \( r \) is a user-specified maximum number of parents allowed per child, typically a small integer like 4. Each of these scores takes some constant time to calculate that depends on the number of observed samples of data, which we call \( K_n \). Therefore, the overall computational complexity of these \( \beta \) family scores is \( O((r + K_n)p^{r+1}) \), as opposed to \( O(K_n p^{r+1}) \) without the structure prior, representing a minor increase in computational complexity.

**Partial Order Priors**

Instead of placing a structure prior over parents themselves, we prefer to place a prior on pairwise order relationships. This translates into a prior over partial orders, that is, which variables are likely to precede a particular child in the order. Instead of specifying prior probabilities on the feature \( P(f_{ij}) \), priors on pairwise orderings are given which we call \( a_{ij} \) for the prior belief that \( X_i \prec X_j \), where \( 0 \leq a_{ij} \leq 1 \). In Eq 3, \( q_i(U_i) \) is the prior on the set \( U_i \) of variables that precede variable \( X_i \) in a given order. We would like this prior probability to: **increase** when \( U_i \) contains variables that are believed to precede \( X_i \); **decrease** when \( U_i \) contains variables that \( X_i \) is believed to precede; and **remain uninformative** otherwise. Using the same style of energy potential function as used previously, we assign this partial-order prior:

\[
q_i(U_i) = \lambda e^{-\lambda(|\pi_i| - a_{ij})} = \lambda e^{-\lambda(C_i + |U_i| - 2 \sum_{j \in U_i} a_{ij})},
\]

where \( \lambda \geq 0 \) is a hyper-parameter which determines the strength of the prior; \( C_i = \sum_{j=1}^{p} a_{ij} \) is a pre-computed sum; and \( |U_i| \) is the number of elements in set \( U_i \). This prior is modular and therefore can be readily incorporated into the dynamic programming method. The overall computational complexity including this partial order prior is \( O(p^2 2^p + K_n p^{r+1}) \).

**Application to Malware Phylogenetics**

Our goal is to understand the behavior of malware by identifying relationships among programs within a malware family. Typically, malware is developed in much the same way as benign software; that is, by modifying, extending and merging previous versions of similar code. By understanding the relationships among malware, we can more quickly analyze the intended behavior of the code to determine the nature of the security threat. Experts use several sources of information to come up with an informed graph depicting the evolution of malware, such as the time the malware was first seen in the wild, the compile-time timestamp, the functionality of the sample, and the obfuscation methods employed by the sample. We investigate the combination of the expert knowledge about the order in which programs appeared, along with using the programs themselves as data to infer the relatedness of the programs.

**Data Views**

This section describes the three types of data (called data views) extracted from a program used in this paper and how we represent these views. It is important to note that the views described are not an exhaustive list, and the framework presented here can be easily extended to incorporate other views of a program.

We take advantage of three different types of data with the aim of covering the most popular data views that have been used for malware analysis in the literature. We use a static, meaning the executable is not run, data view: the control flow graph of the disassembled binary (Kruegel et al. 2006); and we use two dynamic, meaning the data is collected while the program is being executed, data views: dynamic instruction trace (Anderson et al. 2011) and the dynamic system call trace (Hofmeyr, Forrest, and Somayaji 1998). We give a brief explanation of how these views are translated into data, but a more in-depth description of the views used can be found in (Anderson, Storlie, and Lane 2012).

**Markov Chain Data Representation**

The sequence of a program is important and so we transform each data view into a Markov chain representing a summary of the sequential transitions in the program. As an illustrative example, we focus on the dynamic trace data, although this representation is suitable for any sequence-based data view. The dynamic trace data are the instructions the program executes, typically in a virtual machine to reduce the risk of contamination. Given an instruction trace \( P \), we are interested in finding a new representation, \( P' \), such that we can make unified comparisons in graph space while still capturing the sequential nature of the data. We achieved this by transforming the dynamic trace data into a Markov chain which is represented as a weighted, directed graph. A graph, \( g = (V, E) \), is composed of two sets, \( V \) and \( E \). The elements of \( V \) are called vertices and the elements of \( E \) are
called edges. In this representation, the edge weight, \( w_{ij} \), between vertices \( i \) and \( j \) corresponds to the transition probability from state \( i \) to state \( j \) in the Markov chain, hence, the edge weights for edges originating at \( v_i \) are required to sum to \( 1, \sum_{j=1}^m w_{ij} = 1 \). We use an \( m \times m \) adjacency matrix, \( A \), to represent the graph (where \( m = |V| \)), where each entry in the matrix, \( a_{ij} = w_{ij} \) (Kolbitsch et al. 2009).

The nodes of the graph are the instructions the program executes. To find the edges of the graph, we first scan the instruction trace, keeping counts for each pair of successive instructions. After filling in the adjacency matrix with these values, we normalize the matrix such that all of the non-zero rows sum to one. This process of estimating the transition probabilities ensures a well-formed Markov chain. The constructed graphs approximate the pathways of execution of the program, which we exploit as data samples.

The Markov chain graph can be summarized as \( g = (V, E) \), where

- \( V \) is the vertex set composed of unique instructions,
- \( E \) is the weighted edge set where the weights correspond to the transition probabilities and are estimated from the data.

The transition matrices, \( A \), are typically quite sparse, meaning that many of the values are zero. Therefore, to ease the Bayesian network learning, we binarize the matrices, so that any non-zero probabilities are treated as a possible path, and represented with 1. The matrices are then vectorized to treat each value as an observed sample of the data.

**Learning Graphs with Order Priors**

We apply our Bayesian network discovery to families of malware programs along with expert information about the likely order in which these programs were developed. We investigate three malicious program families: Bagle\(^1\), Koobface\(^2\), and Mytob\(^3\), using the data features described above. Because the ground truth for malware development is not available, we also investigate two families of benign (non-malware) program families: NetworkMiner\(^4\) and Mineserver\(^5\). For the benign programs, we have the ground truth phylogenetic graph as reported as extensions, branches and merges in their respective open-source repositories. For the malware programs, as a gold-standard best network, we use phylogenetic graphs constructed by domain experts.

The variables of the Bayesian network represent programs and the observed samples are the discretized transition matrix values. We incorporate domain knowledge about the order in which programs may have been developed. For a family with \( p \) programs, we construct a \( p \times p \) prior matrix, \( A \), such that if \( v_i \) is believed to precede \( v_j \), then \( a_{ij} = 1 \) and \( a_{ji} = 0 \); otherwise \( a_{ij} = 0.5 \) (an uninformative prior).

With informative priors, network discovery should be able to better identify true edges and non-edges than is possible without this knowledge. We compare informative priors on partial orders, and uninformative priors where \( a_{ij} = 1 \forall i, j \) (in this paper, sometimes referred to as “no prior”). We use the BeanDisco C++ implementation (Niinimaki, Parviainen, and Koivisto 2011)\(^6\). We modify the code to implement the informative priors as described in this paper. We impose a limit on the size of parent sets to be at most 4. In all presented results, we use \( \lambda = 1000 \) because it produced reasonable results, but other values could be chosen through cross-validation model selection strategies. We perform a 10-fold cross validation to get posterior estimates of the likelihoods of edges with a standard error (Kohavi 1995).

**Results**

We investigate whether prior knowledge about pairwise-ordering of some vertices improves the edge expectation calculated by a Bayesian network discovery algorithm. Then we quantitatively compare our algorithm against competing algorithms on benign and malware families of programs.

**Effect of Prior Knowledge**

We first look at results comparing the edge expectation \( E(e) \) learned with prior information and without prior information. We split the results into three different categories: learning the true edges, reverse edges and non-edges. Ideally, the expectations of true edges should be close to 1. Non-edges are edges that do not exist in the ground-truth network, and so the expectations of non-edges should ideally be close to 0. Reverse edges are non-edges that have the same end points as a true edge, but in the wrong direction, and so the expectations of reverse edges should again be close to 0. We separate out the reverse edges because they can be particularly difficult to infer from data alone.

Fig. 3 shows the posterior likelihoods (averaged over 10 cross-folds) estimated with the partial-order prior versus those estimated with an uninformative prior (labeled ”no prior”), for each of the three families of malware. In the top row, we see that the estimates for the true edges are generally higher (above the dashed diagonal line) using the prior information than without, indicating that the prior information makes it easier to learn true-positive dependencies. Reverse edges should have low posteriors, and the bottom of the figure indicates that the order prior usually produces lower posteriors for reverse edges (black exes). For the non-edges (red circles in the bottom row), the edge expectations are sometimes improved (below the dashed diagonal line) and sometimes they are worse. It is difficult to see in these plots, but overall there are more non-edges with improved estimation than vice-versa. We quantify these accuracy gains in the next section.

From the learned edge expectations, we produce a phylogenetic graph by selecting a threshold value, \( t \), such that if \( E(e_{ij}) \geq t \) then \( e_{ij} \) is an edge in the graph. Fig. 4 shows the

---

4\(^\text{http://sourceforge.net/projects/networkminer}\)
5\(^\text{https://github.com/fador/mineserver}\)
6\(^\text{http://www.cs.helsinki.fi/u/tniinim/BEANDisco}\)
Comparison with Competing Algorithms

We compare against other algorithms that have been used for learning malware phylogeny, including our previous work. The algorithms that we compare against are: MKLGC (Anderson, Lane, and Hash 2014) which uses a graphical lasso with multiple kernel learning plus clustering of malware programs; Gupta (Gupta et al. 2009), a graph pruning algorithm; and the minimum spanning tree as a naive baseline. These three algorithms each produce a single network as output, rather than an expectation on each edge as our Bayesian network discovery algorithm produces. To produce a single network for comparison from our edge expectations, we threshold edges and only consider those with \( E(c) \geq 0.5 \) to be learned positive edges in the learned network. We calculate the precision and recall of the number of true edges identified by each algorithm. Precision is the number of true positive edges learned divided by the total number of positive edges identified. Recall is the number of true positive edges learned divided by the total number of true edges in the ground truth network. Precision and recall are good metrics to use when the true events of interest (in this case, true edges) are rare in comparison to the total number of events (in this case, all possible edges, the number of which is quadratically larger than the number of true edges) (Davis and Goadrich 2006). For an overall measure of error on identified edges, we compute the Frobenius norm, or F-norm, of the distance between the learned (binary) adjacency matrix and the ground truth adjacency matrix. The F-norm is

\[
\| A - B \|_F = \sqrt{ \sum_{i} \sum_{j} (A_{ij} - B_{ij})^2 }.
\]

Table 1 gives the results comparing precision, recall and F-norm for all of the algorithms including our Bayesian network discovery with prior information, BNPrior; and without prior information, BN. As the table shows in bold, our BNPrior algorithm outperforms the other algorithms for most metrics of the program families. BNPrior outperforms the other algorithms in overall accuracy, as represented by the F-norm score, except in the case of Mytob where BNPrior is a close second-place to Gupta.

Precision and recall should be evaluated as a pair. We see that BNPrior always has either the best precision or recall, and sometimes both. Note that with BN and BNPrior, we can trade-off precision and recall by adjusting the threshold value, \( t \), whereas Table 1 shows only the results at \( t = 0.5 \). For Mineserver and Koobface, BNPrior gives much better recall than any algorithm, but at a somewhat lower precision.
than one or two algorithms. For Mytob, BNPrior gives much better precision than any other algorithm but at a lower recall than what MKLGC gives. Mytob is a difficult learning problem for all of the algorithms. BNPrior demonstrates the most impressive performance improvement on Bagle, while the performance gain on NetworkMiner is also substantial.

Overall, results indicate that software, including malware, phylogeny can be inferred from the program views and the accuracy of the ancestry relationships is improved with prior knowledge about the ordering of the programs. Bayesian networks prove to be a good model for software phylogeny.

Conclusions and Future Directions

Malware phylogenetics can help analysts to more quickly determine the behavior of a new piece of malware code. However, building these graphs from data is difficult and can benefit from including expert knowledge. We investigate the use of expert knowledge about the order in which malware within a family was developed. We provide this information as a novel prior in a Bayesian network discovery algorithm. Bayesian network learning benefits from combining expert knowledge and statistical data. Our partial-order prior leverages the type of information most accessible to humans, while leaving the strength of statistical dependencies to be calculated directly from data. The pairwise order prior is efficient to implement.

Improving the structure of learned Bayesian networks is important in analysis domains that have both data and expert knowledge. As future work, we will look into combining human knowledge with observed data into an interactive approach of generating networks based on expert knowledge, refining the network with observed data, and further modifying the network by experts to produce systems that both match the data and represent human beliefs. The efficient algorithm presented in this paper is just the first step in a knowledge-rich and data-responsive Bayesian network structure learning system. An interactive graph learning system can also help an analyst to explore “what-if” scenarios to evaluate alternative phylogenetic graphs that could have produced the family of malware.

In considering the human element of malware development, phylogenetic graphs exploit the fact that malware is developed like other software. In future work, we can investigate the potential for using these learned phylogenetic graphs to predict future versions of malware programs within a family.
Table 1: Phylogenetic graph reconstruction results in terms of F-norm distance to gold standard adjacency matrix, precision of true edges, and recall of true edges.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>F-norm</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network-Miner</td>
<td>BNPrior</td>
<td>4.3589</td>
<td>0.5128</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td>BN</td>
<td>6.1644</td>
<td>0.2632</td>
<td>0.5000</td>
</tr>
<tr>
<td></td>
<td>MKLGC</td>
<td>4.5826</td>
<td>0.4857</td>
<td>0.8500</td>
</tr>
<tr>
<td></td>
<td>Gupta</td>
<td>5.0000</td>
<td>0.3810</td>
<td>0.4000</td>
</tr>
<tr>
<td></td>
<td>MST</td>
<td>5.6569</td>
<td>0.3500</td>
<td>0.7000</td>
</tr>
<tr>
<td>Mine-server</td>
<td>BNPrior</td>
<td>2.8284</td>
<td>0.6818</td>
<td>0.9375</td>
</tr>
<tr>
<td></td>
<td>BN</td>
<td>5.3852</td>
<td>0.0667</td>
<td>0.0625</td>
</tr>
<tr>
<td></td>
<td>MKLGC</td>
<td>4.0000</td>
<td>0.7222</td>
<td>0.8125</td>
</tr>
<tr>
<td></td>
<td>Gupta</td>
<td>4.7968</td>
<td>0.8462</td>
<td>0.3438</td>
</tr>
<tr>
<td></td>
<td>MST</td>
<td>7.4833</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Bagle</td>
<td>BNPrior</td>
<td>4.6904</td>
<td>0.5263</td>
<td>0.8333</td>
</tr>
<tr>
<td></td>
<td>BN</td>
<td>7.4162</td>
<td>0.1026</td>
<td>0.1667</td>
</tr>
<tr>
<td></td>
<td>MKLGC</td>
<td>5.7446</td>
<td>0.2000</td>
<td>0.3333</td>
</tr>
<tr>
<td></td>
<td>Gupta</td>
<td>6.5574</td>
<td>0.1200</td>
<td>0.0120</td>
</tr>
<tr>
<td></td>
<td>MST</td>
<td>8.3607</td>
<td>0.0208</td>
<td>0.0417</td>
</tr>
<tr>
<td>Mytob</td>
<td>BNPrior</td>
<td>6.2450</td>
<td>0.2059</td>
<td>0.3684</td>
</tr>
<tr>
<td></td>
<td>BN</td>
<td>7.0000</td>
<td>0.0833</td>
<td>0.1579</td>
</tr>
<tr>
<td></td>
<td>MKLGC</td>
<td>7.9373</td>
<td>0.1563</td>
<td>0.5263</td>
</tr>
<tr>
<td></td>
<td>Gupta</td>
<td>6.0828</td>
<td>0.0500</td>
<td>0.0526</td>
</tr>
<tr>
<td></td>
<td>MST</td>
<td>7.2801</td>
<td>0.0526</td>
<td>0.1053</td>
</tr>
<tr>
<td>Koobface</td>
<td>BNPrior</td>
<td>4.5826</td>
<td>0.4516</td>
<td>0.7778</td>
</tr>
<tr>
<td></td>
<td>BN</td>
<td>5.8310</td>
<td>0.1923</td>
<td>0.2778</td>
</tr>
<tr>
<td></td>
<td>MKLGC</td>
<td>5.2915</td>
<td>0.5812</td>
<td>0.5000</td>
</tr>
<tr>
<td></td>
<td>Gupta</td>
<td>5.9161</td>
<td>0.3158</td>
<td>0.3333</td>
</tr>
<tr>
<td></td>
<td>MST</td>
<td>7.2111</td>
<td>0.0278</td>
<td>0.0556</td>
</tr>
</tbody>
</table>

Acknowledgments Thanks to Aaron McPhall and the Advanced Computing Services division at Los Alamos National Lab for making data available to us; as well as Curtis Storlie and Joshua Neil for their support. This work is funded by Los Alamos National Lab LDRD 20140355ER.

References


