2010 IEEE Symposium on Security and Privacy

Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors

Matt Fredrikson[1] and Somesh Jha
Department of Computer Sciences
University of Wisconsin
Madison, WI, USA
{*mfredrik,jha*}*@cs.wisc.edu*

Mihai Christodorescu and Reiner Sailer
IBM T.J. Watson Research Center
Hawthorne, NY, USA
{*mihai,sailer*}*@us.ibm.com*

Xifeng Yan
Department of Computer Science
University of California
Santa Barbara, CA, USA
xyan@cs.ucsb.edu

***Abstract*—Fueled by an emerging underground economy, malware authors are exploiting vulnerabilities at an alarming rate. To make matters worse, obfuscation tools are commonly available, and much of the malware is open source, leading to a huge number of variants. Behavior-based detection techniques are a promising solution to this growing problem. However, these detectors require precise specifications of malicious behavior that do not result in an excessive number of false alarms.**

In this paper, we present an automatic technique for extracting *optimally discriminative specifications*, which uniquely identify a class of programs. Such a discriminative specification can be used by a behavior-based malware detector. Our technique, based on graph mining and concept analysis, scales to large classes of programs due to probabilistic sampling of the specification space. Our implementation, called HOLMES, can synthesize discriminative specifications that accurately distinguish between programs, sustaining an 86% detection rate on new, unknown malware, with 0 false positives, in contrast with 55% for commercial signature-based antivirus (AV) and 62-64% for behavior-based AV (commercial or research).

I. INTRODUCTION

The problem posed by malware is real, ubiquitous, and continues to grow steadily [1]. The past decade has seen a fundamental shift in incentive for creators and distributors of malware. Whereas the traditional motivation for hackers was primarily based on reputation, there now exists a substantial and continually growing economy that trades in bank account information, credit card numbers, and other sensitive information of real monetary value on the Internet. Malware plays an integral role in this economy as a tool for criminals to harvest sensitive information, as well as a platform for launching other illegal money making-schemes.

The predominant technique for detecting malware remains signature-based scanning, where the syntactic characteristics of a malware instance form a signature used to identify that specific instance. Since malware writers have an economic incentive to evade detection, they have perfected the art of creating malware variants such that each requires a distinct signature. Consequently, the number of distinct malware is growing at an alarming rate. David Perry from Trend Micro reported that some antivirus (AV) vendors are seeing 5,000 new malware samples per day [2]. The ability to create a large number of variants gives hackers an upper hand, as they can automatically generate new malware much more quickly than analysts can develop signatures for them. Recently, researchers have proposed a number of techniques that examine the underlying behavior of suspected malware [3], [4], [5]. There are also a few products on the market (such as Threatfire and Sana Security) that use behavior-based detection. These techniques are very promising because it is harder for an attacker to radically change the behavior of a malware than to morph its syntactic structure. Essentially, a single syntactic signature maps to a single instance of malware, whereas a single *behavioral specification* maps to multiple instances.

Constructing behavioral specifications that have a low false positive rate and at the same time are general enough to detect variants of malware is a major challenge. The most common technique to date for generating behavioral specifications relies almost entirely on manual analysis and human ingenuity – an expensive, time-consuming, and error-prone process that provides no guarantees regarding the quality of the resultant specifications. An automatic technique for building such specifications is desirable, both to reduce the AV vendors' response time to new threats and to guarantee precise behavioral specifications. If the behavioral specification used for detection is not specific enough, then there is a risk that benign applications will be flagged as malware. Similarly, if it is too specific then it may fail to detect minor variants of previously observed malware. In this paper we address the challenge of *automatically* creating behavioral specifications that strike a suitable balance in this regard, thus removing the dependence on human expertise.

We make the observation that the behavioral specifications used in malware detection are a form of *discriminative specification*. A discriminative specification describes the unique properties for a set of programs, in contrast to another set of programs. This paper gives an automatic technique that combines graph mining and concept analysis to synthesize discriminative specifications, and explores an application of the resulting specifications to malware detection. Given a set of *behavior graphs* that describe the semantics exhibited by malicious and benign applications, the graph mining operation extracts *significant behaviors* that can be used to

[1] This work was done over the course of an internship at IBM Research.

1081-6011/10 $26.00 © 2010 IEEE
DOI 10.1109/SP.2010.11

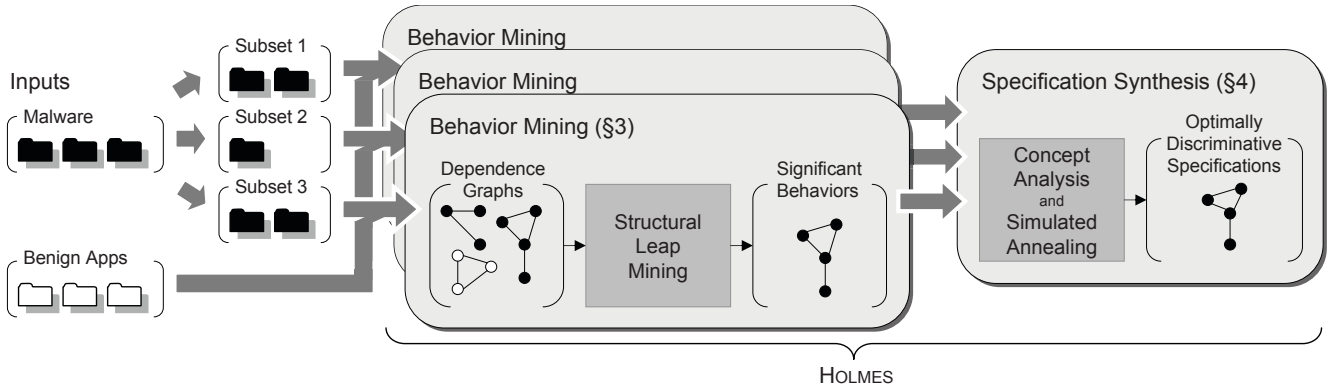


Figure 1. HOLMES combines program analysis, structural leap mining, and concept analysis to create optimal specifications.

distinguish the malware from benign applications. As these behaviors are not necessarily shared by all programs in the same set (as, for example, there are many ways in which a malicious program can attack a system), we use them as building blocks for constructing discriminative specifications that are general across variants, and thus robust to many obfuscations. Furthermore, because our graph mining and specification construction algorithms are indifferent to the details of the underlying graph representation, our technique complements and benefits from recent advances in binary analysis [6], [7] and behavior graph construction [5].

Our paper makes the following contributions:

- We divide the problem of constructing a specification into two tasks, (1) mining significant behaviors from a set of samples and (2) synthesizing an optimally discriminative specification from multiple sets of mined behaviors. For each task we introduce novel algorithms, derived from leap mining [8] and concept analysis [9], respectively. The two tasks naturally mirror a human analyst's workflow, where new malware samples are first analyzed for unique behaviors and then merged with behaviors from existing malware. (Sections III and IV)
- We develop a tool called HOLMES[1] that takes a set of malicious and benign binaries, extracts significant malicious behaviors, and creates an optimally discriminative specification. Our experience with HOLMES shows that it automatically identifies both behaviors documented by AV-industry analysts as well as previously undocumented behaviors, and that it scales with the degree of behavioral diversity in the malware and benign sets. (V)
- We perform evaluation on unknown malware, showing that a specification produced by HOLMES detects it with high accuracy (86% detection rate with 0 false alarms), even when we mine behaviors from one malware family and test synthesized specifications on another. (Sections VI and VII)

[1]Our tool, just like Sherlock Holmes, combs through mountains of seemingly unrelated information to identify signs of malicious activities.

II. OVERVIEW

The behavior of a program can be thought of as its effect on the state and environment of the system on which it executes. Most malware relies on system calls to deliver a malicious payload, so reasoning about behavior in terms of the system calls made by a program allows us to succinctly and precisely capture the intent of the malware author, while ignoring many implementation-specific artifacts. Using this representation, we wish to derive a behavioral specification that is descriptive of a given set of programs (the *positive set of programs*) but does not describe any program in a second set (the *negative set of programs*). In the malware detection case, the positive set consists of malicious programs and the negative set consists of benign programs, and the goal is to construct a specification that is characteristic of the malicious programs but not of the benign programs.

For an arbitrarily chosen positive set of programs one is unlikely to find a single behavior common to all of them; if there is one such common behavior, it is likely also present in the programs from the negative set. Thus, we need to partition the positive set into subsets of similar programs, such that programs in the same subset share many behaviors. This leads us to the high-level workflow of our technique, which is presented in 1 and proceeds as follows:

I. The positive set of programs is divided into disjoint subsets of behaviorally similar programs. This can be performed manually, or using existing malware clustering techniques [10], [11].

II. Using existing techniques for dependence-graph construction [12], a graph is constructed for each malware and benign application to represent its behavior.

III. The significant behaviors specific to each positive subset are mined. (III)

A significant behavior is a sequence of operations that distinguishes the programs in a positive subset from

all of the programs in the negative subset. We use *structural leap mining* [8] to identify multiple distinct graphs that are present in the dependence graphs of the positive subset and absent from the dependence graphs of the negative set.

IV. The significant behaviors mined from each positive subset are combined to obtain a discriminative specification for the whole positive set. (IV)

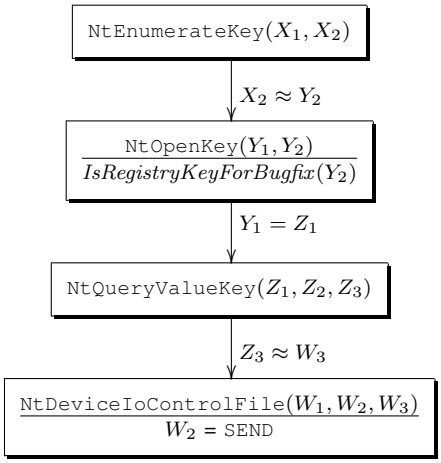
Two significant behaviors can be combined either by merging them into one significant behavior that is more general, or by taking the union of the two behaviors. We use *concept analysis* to identify the behaviors that can be combined with little or no increase in the coverage rate of the negative set, while maintaining the coverage rate of the positive set. As there are exponentially many ways of combining behaviors, we use probabilistic sampling to approximate the optimal solution in an efficient manner.

A. Behavior Mining

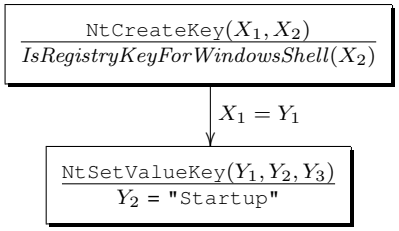
We will walk through the application of our behavior extraction algorithm and specification construction on the spyware family Ldpinch. According to Symantec security response [13], members of the Ldpinch family install themselves to persist on a system, and then attempt to steal sensitive system information, such as passwords and email addresses, and send it back to the malware author.

The first step of the behavior-mining algorithm extracts portions of the dependence graphs of programs from the positive set that correspond to behavior that is *significant* to the programs' intent. At a high level, this step can be thought of as contrasting the graphs of positive programs against the graphs of negative programs, and extracting the subgraphs that provide the best contrast. Note that we do not use any *a priori* notion of significance, but rather rely on the mining algorithm to identify significant behaviors.

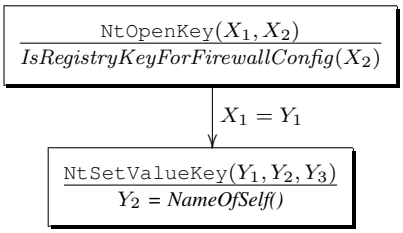
2 presents a small portion of the output produced by applying our algorithm using the Ldpinch spyware family (the positive set) and some benign programs (the negative set, listed in VI-B) as an example. These three graphs correspond respectively to leaking bugfix information, setting the system to execute the malware each time the system is restarted, and adding the malware to the list of applications that can bypass the system firewall. The first two behaviors were previously reported by Symantec analysts, while the third behavior (bypassing the firewall) was produced by HOLMES, along with the others. To the best of our knowledge, *human analysts have not previously reported this behavior for the Ldpinch family*.



(a) Significant behavior: Leaking bugfix information over the network.



(b) Significant behavior: Adding a new entry to the system autostart list.



(c) Significant behavior: Bypassing firewall to allow malicious traffic.

Figure 2. Three significant behaviors extracted from the Ldpinch family.

B. Specification Synthesis

The information produced by our behavior-mining algorithm can be thought of as a collection of high-level behavioral primitives that characterize programs from the positive set. Considering the behaviors listed in 2, we could create a detector for the Ldpinch family by treating each mined behavior as a component in a specification of malicious behavior. With this, we would treat as malicious any program that installs itself to persist on restart *and* disables the firewall *and* sends system bugfix information over the network. However, this would fail to detect other families of spyware that do not exhibit these exact behaviors. For example, a piece of spyware that only installs itself to persist on restart and leaks bugfix information, but does not disable the system firewall, would not be detected. In short, a detector created using information extracted from one specific family of spyware is too specific to capture the wide range of behavior observed across other families.

To generalize such narrowly-defined specifications, we have developed an algorithm for generating discriminative specifications composed only of the significant behaviors

descriptive of a given corpus of programs as a whole. Our algorithm can guarantee that these specifications are optimal with respect to the true and false positive rates over a target sample distribution. However, as constructing an optimal specification can take a prohibitively large amount of time, our algorithm is also capable of generating good specifications efficiently. To accomplish this, we frame the problem of constructing optimal characteristic specifications as a clustering problem, and apply concept analysis [9] to enumerate a set of possible clusters over the sample distribution. We then use *simulated annealing* [14], a probabilistic sampling technique, to efficiently search for a subset of clusters that can be used to construct a close approximation to the optimal specification. This is done in such a way as to allow a direct tradeoff between efficiency and the optimality of the solution, making our technique useful in a number of settings.

III. Mining Significant Behaviors

Our technique for mining behaviors by contrasting two sets of programs is dependent on a precise notion of software behavior. At a high level, a behavior consists of a set of operations invoked by a program and the relationships between them. We use *data dependence graphs* over a set of operations Σ to describe behaviors. in

Definition 1 (Behavior): A behavior is a data dependence graph $G = (V, E, \alpha, \beta)$ where:

- the set of vertices V corresponds to operations from Σ,
- the set of edges $E \subseteq V \times V$ corresponds to *dependencies* between operations,
- the labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the name of their corresponding operations, and
- the labeling function $\beta : V \cup E \rightarrow \mathcal{L}_{dep}$ associates vertices and edges with formulas in some logic $\mathcal{L}_{dep}$ capable of expressing constraints on operations and the dependencies between their arguments.

In the context of malware detection we equate, without loss of generality, the set of operations Σ with the set of system calls on our target platform. Thus, we will use the terms "events", "operations", and "system calls" interchangeably.

We characterize operations as functions. An operation $\sigma \in \Sigma$ is a function of N typed and named variables, $\sigma : a_1 : \tau_1 \times a_2 : \tau_2 \times \cdots \times a_N : \tau_N \rightarrow R : \tau_R$, where τ_i is the type of the i-th argument a_i and τ_R is the type of the return value R. The result of a program execution E is a sequence of operation invocations (or *trace*) $T_E = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$.

A program P exhibits a behavior $G = (V, E, \alpha, \beta)$ if it can produce an execution trace $T = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ with the following properties:

1) Every operation in the behavior corresponds to an operation invocation in the trace and its arguments satisfy the constraints:

$$\forall v_i \in V \ . \ \exists \sigma_i \in T \ . \ \alpha(v_i) = \sigma_i \wedge \beta(v_i)[\sigma_i]$$

Table I
Some of the security labels (and corresponding logical constraints) that allow us to capture information flows.

Security Label	*Description*
$NameOfSelf$	The name of the currently executing program.
$IsRegistryKeyForBootList$	A Windows registry key listing software set to start on boot.
$IsRegistryKeyForWindows$	A registry key that contains configuration settings for the operating system.
$IsSystemDirectory$	The Windows system directory.
$IsRegistryKeyForBugfix$	The Windows registry key containing list of installed bugfixes and patches.
$IsRegistryKeyForWindowsShell$	The Windows registry key controlling the shell.
$IsDevice$	A named kernel device.
$IsExecutableFile$	Executable file.

In our graphical representation of behaviors, the security labels appear as arguments constraints inside nodes. For example, in 2(a), the `NtOpenKey` node has its argument Y_2 constrained by the security label $IsRegistryKeyForBugfix$.

2) The logic formulas on edges connecting behavior operations is satisfied by the corresponding pair of operation invocations in the trace:

$$\forall (v_i, v_j) \in E \ . \ \exists \sigma_i, \sigma_j \in T \ . \ \beta(v_i, v_j)[\sigma_i, \sigma_j]$$

Capturing information flow in dependence graphs: Existing techniques for constructing dependence graphs from programs provide only data-flow (and sometimes control-flow) dependencies between operations [15], [16], [5]. Information flows are characterized not only by the path taken by the data in the program but also by the security labels assigned to the data source and the data sink of the information flow.

We enhance the dependence graphs obtained from the existing tools by assigning labels to particular files, directories, registry keys, and devices based on their significance to the system (e.g., system startup list, firewall settings, system executables). The Microsoft Windows documentation [17] lists a large number of files, directories, registry keys, and devices that are relevant to system security, stability, and performance. We list in I a few of the labels we apply to the nodes in the constructed dependency graphs. These labels, although operating system-specific, are not tied to any particular class of malware or benign programs. The labels are represented in our behavior formalism as logical formulas expressing constraints on arguments. Unlabeled arguments that are also not part of a dependency are fully unconstrained, meaning that any program (malicious or benign) can set them arbitrarily.

Approaches to mining behavior: Behavior mining analyzes the dependence graphs of programs from a positive subset and the negative set and identifies the subgraphs that are most useful in uniquely characterizing the programs in the positive subset. We associate a quality metric to each

behavior to evaluate each candidate graph resulting from mining. Roughly speaking, the quality metric of a behavior is proportional to the number of programs in the positive set that exhibit that behavior and inversely proportional to the number of programs in the negative set that exhibit it. The goal of behavior mining is then to find a set of behaviors that maximize this quantity over the given program sets. We capture this notion of behavior, which we term *significant behavior*, using statistical principles borrowed from information theory to construct a precise quality metric (III-A).

There are a number of techniques that have been developed to perform this type of *discriminative subgraph mining*, such as minimal contrast subgraph mining [18] and structural leap mining [8]. Minimal contrast subgraph mining returns complete results, and has exponential runtime complexity. On the other hand, leap mining returns approximate results, but has lower complexity. Because our experience indicates that malware dependence graphs can be quite large, we characterize our mining task as an optimization problem over the information gain function, and use leap mining [8] to find approximately optimal solutions. Furthermore, to better characterize the set of programs, we modify the basic leap mining algorithm to return multiple results that are structurally dissimilar from each other (III-B).

A. Behavior Significance

To formally characterize the notion of behavior significance used by our technique, we consider the dependence-graph sets $\mathcal{G}_+$ and $\mathcal{G}_-$ constructed from the members of a positive subset and the negative set, respectively. From these sets, we wish to produce a set of graphs $\mathcal{G}_\Delta$ such that when given an arbitrary graph $G \in \mathcal{G}_+ \cup \mathcal{G}_-$, we can deduce with high certainty whether $G \in \mathcal{G}_+$ by performing subgraph isomorphism tests with the elements in $\mathcal{G}_\Delta$. To this end, we turn to information gain as a way of comparing candidate graphs for suitability with respect to this criterion.

Information gain is defined in terms of Shannon entropy. In our setting, the Shannon entropy of $\mathcal{G}_+ \cup \mathcal{G}_-$ is defined with respect to the membership of elements in either $\mathcal{G}_+$ or $\mathcal{G}_-$. Let E_+ refer to the event that a graph G randomly selected from $\mathcal{G}_+ \cup \mathcal{G}_-$ is in $\mathcal{G}_+$, and define E_- similarly for $\mathcal{G}_-$. We define the *entropy* of $\mathcal{G}_+ \cup \mathcal{G}_-$, written $H(\mathcal{G}_+ \cup \mathcal{G}_-)$,

$$H(\mathcal{G}_+ \cup \mathcal{G}_-) = -P\left[E_+\right]\log(P\left[E_+\right]) - P\left[E_-\right]\log([E_-])$$

Intuitively, $H(\mathcal{G}_+ \cup \mathcal{G}_-)$ corresponds to the uncertainty regarding whether some $G \in \mathcal{G}_+ \cup \mathcal{G}_-$ belongs to $\mathcal{G}_+$ or to $\mathcal{G}_-$. Reducing this uncertainty means that one can tell with increased assurance whether some graph G is in $\mathcal{G}_+$ rather than in $\mathcal{G}_-$.

In terms of entropy, our problem is that of finding additional information that allows us to reduce the entropy of $\mathcal{G}_+ \cup \mathcal{G}_-$. This is precisely what information gain models; information gain is the expected reduction in entropy caused by partitioning the original set, $\mathcal{G}_+ \cup \mathcal{G}_-$, into two smaller sets according to some criterion. In our case, this criterion is the presence of a candidate subgraph via subgraph isomorphism in the elements of the original set. To simplify notation, we define information gain in terms of an arbitrary set of graphs S rather than $\mathcal{G}_+ \cup \mathcal{G}_-$. For some graph G and set of graphs S, we refer to the set $S_G = \{G_I \in S | G \subseteq G_I\}$, where $\subseteq$ denotes the subgraph isomorphism relation between to graphs. Information gain is defined over a set of graphs S and a candidate graph G:

$$Gain(S, G) = H(S) - \left(\frac{|S_G|}{|S|} H(S_G) + \frac{|S \setminus S_G|}{|S|} H(S \setminus S_G) \right)$$

where $S \setminus S_G$ is the set difference between S and S_G. The second part of this equation sums the weighted entropy of S after it is partitioned according to membership of G. $Gain(\mathcal{G}_+ \cup \mathcal{G}_-, g)$ is a measure of how well a graph g will allow us to determine whether some graph $G \in \mathcal{G}_+ \cup \mathcal{G}_-$ belongs to $\mathcal{G}_+$ by performing subgraph isomorphism tests between g and G, which is the stated goal of significant-behavior mining.

Definition 2 (Significant Behavior): Given two sets of dependence graphs, $\mathcal{G}_+$ and $\mathcal{G}_-$, a significant behavior g is a subgraph of some dependence graph $g \in \mathcal{G}_+$ such that the information gain $Gain(\mathcal{G}_+ \cup \mathcal{G}_-, g)$ is maximized.

Information gain is a powerful statistical tool that allows us to select significant behaviors from the positive set. We use it as the quality measure to guide the behavior mining process.

B. Structural Leap Mining

Using 2 we frame behavior mining as an optimization problem over $Gain(\mathcal{G}_+ \cup \mathcal{G}_-, g)$, to find g such that $Gain$ is maximized. To search the space of possible solutions for g efficiently, we use a technique called *structural leap mining* [8] to find approximate solutions to this optimization problem. Like most discriminative objective functions, information gain is not anti-monotonic [8], which could invalidate all of the frequency-centric graph mining algorithms that adopt this property as a major pruning heuristic. Instead, structural leap mining exploits the correlation between structural similarity and similarity in objective score for functions like information gain to quickly prune the candidate search space. Specifically, candidate subgraphs are enumerated from small to large size in a search tree and their score is checked against the best subgraph discovered so far. Whenever the upper bound of information gain in a search branch is smaller than the score of the best subgraph discovered, the whole search branch can be safely discarded. In addition to this *vertical* pruning of the search space, siblings of previously enumerated candidates are "leaped over" since it is likely they will share similar information gain. This *horizontal* pruning is effective in pruning subgraphs that represent similar behaviors. While this might result in an approximation of the optimal solution, it can be shown that

structural leap mining can guarantee the optimal solution at the expense of additional runtime. For further details, we refer the reader to the original paper [8].

Based on the observation that a set of programs is not likely to be sufficiently characterized by a single significant behavior, we modified the original leap mining algorithm to return the top k significant behaviors. Furthermore, to avoid redundancy among these behaviors we use a numerical measure of *behavior similarity* (3), and require that each pair of behaviors in our top-k list has a behavior similarity that falls below a predefined threshold.

Definition 3 (Behavior Similarity): Given two graphs g and g', their similarity is $|\frac{2E(g'')}{E(g)+E(g')}|$, where g'' is the maximal common subgraph of g and g' and $E(g)$ denotes the number of edges in g.

This definition of similarity ensures that the graphical encodings of the behaviors in our list share little similarity in structure; this is usually sufficient to eliminate redundancy in significant behaviors. We call such subgraphs *similarity-aware top-k significant behaviors*, and iteratively call structural leap mining to extract these behavior patterns.

IV. Synthesis of Discriminative Specifications

A specification is a collection of behaviors and a characteristic function that defines one or more subsets of the collection. A program matches a specification if it matches all of the behaviors in at least one characteristic subset. If we interpret each characteristic subset as corresponding to a *class* or *family* of programs, then a single specification can require that a program match at least one of many families that differ in arbitrary ways. A specification is *discriminative* if it matches malicious programs but does not match benign programs. The goal of specification synthesis is to construct an optimally discriminative specification from a given set of behaviors by identifying the behaviors and the characteristic function that define the specification.

It is possible to use the behaviors produced by mining to construct a discriminative specification. For example, one such specification is naïvely constructed by listing each mined behavior that is exhibitied by each sample in the positive set, and asserting that a program matches the specification if it exhibits all of the behaviors on any of the samples' lists. This specification is discriminative because it makes use of all of the significant behaviors mined from the positive set, each of which is known to discriminate from the negative set. However, it is likely to fit the input samples too closely. This solution is not likely to fulfill our goal of covering a large number of variants and malware from other families, as the samples in the malicious set may exhibit behaviors that are not essential to their respective families. We must find a set of behaviors that captures only the essential characteristics of the positive set, without over-generalizing to the negative set.

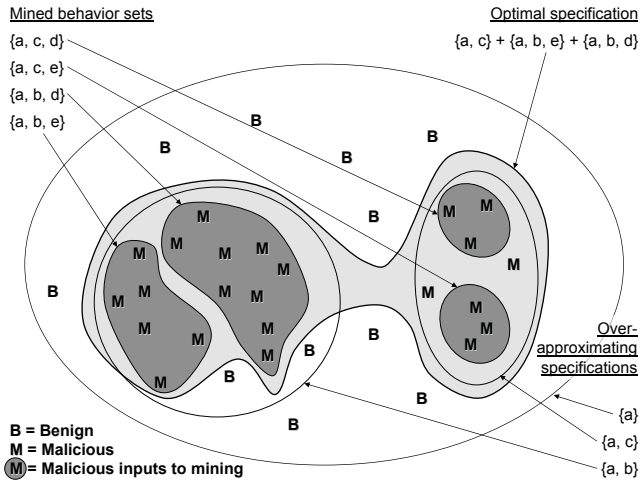


Figure 3. An example clustering of malware and benign applications over behavior set $\{a, b, c, d, e\}$.

To relate the target sample set to specification construction, we observe that each set of behaviors induces a cluster of samples, and frame the problem of finding an optimal specification as a clustering problem. In this setting, a specification corresponds to a set of clusters, and the problem of constructing an optimal specification becomes one of finding a set of clusters that satisies our notion of optimality over samples.

For example, consider the set of malware clusters presented in 3. The samples labeled M correspond to the labeled malicious samples used in the mining step, and those labeled B are benign. The behaviors produced by mining are labeled a, b, c, d, and e; we are not concerned with the details of each behavior, but only their inclusion in the given samples. The specification created using the naïve algorithm described at the beginning of this section corresponds to selecting clusters $\{a, b, e\}$, $\{a, b, d\}$, $\{a, c, d\}$, and $\{a, c, e\}$. Notice that this specification covers many of the malicious samples while excluding the benign applications, but fails to cover all of the malicious samples. Alternatively, we can construct a specification out of the single cluster $\{a\}$ that is sure to cover all of the malicious samples, but this is too general, as it also covers all of the benign applications. In fact, the best specification in this example corresponds to the selection of clusters $\{a, c\}$, $\{a, b, e\}$, and $\{a, b, d\}$. While this example is trivial, it resembles a simplified version of the scenarios we encountered in our evaluation (VI).

A. Formal Concept Analysis

Formal concept analysis [9] provides a convenient set of formalisms and tools to reason about specification synthesis in terms of clusters and sample coverage. In this framework, a *concept* is a pair (O, A) containing a set of objects O and a set of attributes A. O is referred to as the *extent* of the concept, and A the *intent*. In our setting, O is a cluster of

samples, and A is the set of mined behaviors exhibited by all of the samples in O. For our purposes, specifications are formalized in terms of concepts over behaviors.

Definition 4 (Behavioral Specification and Matching): Given a set of concepts $\{C_1, C_2, \ldots, C_n\}$, we define a *behavioral specification* $S(C_1, C_2, \ldots, C_n)$ to be a disjunction of the conjunction of all behaviors in $A_1, A_2, \ldots, A_n$, respectively. A program *matches* a specification $S(C_1, C_2, \ldots, C_n)$ if it exhibits all of the behaviors in *any* of the intents of its concepts $C_1, C_2, \ldots, C_n$.

A single malware specification of this type covers as many samples as possible from all families present in the malicious set.

Given a listing of samples and the behaviors that they exhibit, concept analysis provides a mechanism to construct all of the possible clusters of samples in terms of their attribute behaviors. We use an algorithm due to Nourine and Raynaud [19] for its simplicity and performance on our data. The algorithm begins by constructing all concepts with singleton extents, and computes the pairwise intersection of the intent sets of these concepts. This process is repeated until a fixpoint is reached, and no new concepts can be constructed. When this algorithm terminates, we are left with an explicit listing of all of the sample clusters that can be specified in terms of one or more mined behaviors.

Having enumerated each possible cluster, the problem of synthesizing an optimal specification is reduced to the task of selecting a set of clusters that are optimally discriminative over the samples. We formalize this notion by defining specification optimality in terms of coverage of the positive and negative sets.

Definition 5 (Optimal Specification): Given a specification S and a set of positive and negative samples C, let $tp_C(S)$ be the number of positive samples in C covered by S, and $fp_C(S)$ be the number of negative samples in C covered by S. We say that S is optimal if for a given threshold t, $tp_C(S) \geq t$ and $fp_C(S) < fp_C(R),\ \forall R.tp_C(R) \geq t$.

Given 5, our goal is to find concepts $C_1, C_2, \ldots, C_n$ such that $S(C_1, C_2, \ldots, C_n)$ satisfies this definition of optimality for a given threshold t. This definition allows us to specify minimum requirements for the desired true positive rate, and pick the specification with the lowest false positive rate. By varying the threshold t, we can define specifications that range from 100% true positives, to 0% false positives.

B. Simulated Annealing

Because specifications with redundant concepts do not introduce any changes to the sample coverage of a specification, the solution space for the specification optimization problem consists of all possible combinations of concepts produced by the enumeration algorithm. As is typical of combinatorial optimization problems, the size of this space is prohibitively large for an exact exhaustive search for the best solution, so approximate methods are needed.

Simulated Annealing is a general probabilistic technique for finding approximate solutions to global optimization problems [14]. Specifically, if E is a finite set and $U : E \rightarrow \mathbf{R}$ is some *cost function* defined over the elements of E, then simulated annealing finds the global minimum i_0 of U, $i_0 \in E$ such that $U(i_0) < U(i)$ for all $i \in E$. The technique proceeds iteratively as follows:

1) At each step, a candidate solution i is examined, and one of its *neighbors* $j \in N(i)$ is selected for comparison.
2) The algorithm moves to j with some probability that is positively correlated with $U(j) < U(i)$ and a *cooling parameter* T.
3) T is decremented according to a *cooling schedule*. When T reaches a specified minimum, the algorithm terminates and the current solution is returned.

The *neighbor function* $N : E \rightarrow 2^E$ used in step 1 is defined for each particular application.

Simulated annealing resembles a randomized version of the greedy algorithm; the random component allows the search to escape from local minima, and eventually converge on the global minimum. Notice from steps 2 and 3 that the tendency of the algorithm to move to a less-optimal solution decreases as the cooling parameter approaches its minimum. This causes the search to cover wider bands of the candidate space early on, and make smaller movements towards local minima later. It has been shown that the use of an appropriate cooling schedule guarantees eventual convergence to the global minimum of the cost function [14], although this convergence may be quite slow. In practice it is often acceptable to terminate the algorithm before the cooling parameter reaches its minimum, and still end up with a good solution. Our evaluation results corroborate this claim (VI).

1) Cost Function: To map the cost function to our setting, we must account for two independent variables: the true and false positive rates over the set samples C. For a given threshold t, the algorithm should discard all solutions S with $tp_C(S) < t$, and select the remaining solution that is minimal with respect to fp_C. The most straightforward choice is:

$$U(S) = \begin{cases} fp_C(S) & \text{if } tp_C(S) \geq t \\ M & \textit{otherwise} \end{cases}$$

where M is larger than $fp_C(S)$ for all possible S. Provided there exists a specification that meets the coverage requirement t, finding S that minimizes this function is equivalent to finding the optimial specification for a given t.

2) Neighbor Function: Selecting an appropriate neighbor function requires care to ensure that the search is able to efficiently enumerate the candidate solution space. For example, it is not productive to consider specifications that

contain duplicate concepts. The structural relationships of concepts to each other may also have implications for coverage. Recall that a sample matches a specification if it exhibits all of the behaviors of any of the concepts in the specification. If the specification contains two concepts C_i, C_j such that $C_i \subset C_j$, then C_i will match any sample that C_j does, and C_j can be removed from the specification.

With these considerations in mind, we construct a neighbor function that allows the algorithm to search the candidate space efficiently, without encountering obvious redundancies. For a candidate solution $S(C_1, \ldots, C_n)$, define $\mathcal{B}_{S(C_1,\ldots,C_n)}$ to be the set of specifications obtained by removing one specification in $\{C_1, \ldots, C_n\}$ from S. Similarly, define $\mathcal{F}_{S(C_1,\ldots,C_n)}$ as the set of specifications obtained by adding a single specification C_k to $\{C_1, \ldots, C_n\}$ such that $C_k \not\subseteq C_i$ for all $C_i \in \{C_1, \ldots, C_n\}$. Then for a specification S, $N(S) = \mathcal{B}_S \cup \mathcal{F}_S$. This definition of N allows the algorithm to move backwards or forwards without including concepts that contribute nothing to the coverage of the specification.

3) Sampling and Cooling: To define the acceptance probabilities for a transition from specification S_i to S_j, we use the Metropolis sampler [14]:

$$\alpha_{ij}(T) = \begin{cases} e^{\frac{U(j)-U(i)}{T}} & \text{if } U(j) > U(i) \\ 1 & \text{if } U(j) \leq U(i) \end{cases}$$

$\alpha_{ij}(T)$ is the acceptance probability for this transition in terms of the cooling parameter T. We selected this sampler for its simplicity, as well as the fact that it is straightforward to define a cooling schedule for this sampler that is guaranteed to converge to a global minimum [14]. The cooling schedule we use is indexed by the current iteration k,

$$T_k = \frac{\max\{U(i) - U(j), \forall i, j \in N(i)\}}{\log(k)}$$

which satisfies this convergence property. While $\{T_k\}_{k\geq 0}$ converges slowly, in practice this schedule produces a near-optimal solution quickly, thus giving an acceptable answer efficiently while leaving the possibility of finding the optimal specification open in exchange for additional runtime cost.

C. Constructing Optimal Specifications

Using concept analysis and the sampling techniques discussed in the previous subsection, it is possible to devise a practical algorithm that takes a threshold t, a set containing labeled positive and negative samples, and a set of behaviors produced using the behavior mining techniques described in III, and produces a behavioral specification satisfying 5. This algorithm, SPECSYNTH, is presented in 4.

SPECSYNTH begins by constructing the full set of concepts describing the input data using the algorithm of Nourine and Raynaud [19]. We perform an additional optimization to reduce the cost of constructing the full set, by removing the redundant concepts discussed in IV-B2 before running NOURINE-RAYNAUD. After computing the concept set, simulated annealing is run until $\{T_k\}_{k\geq 0}$ converges, and the best solution encountered during the search is returned.

```
function SPECSYNTH(D, A, t)
    C ← ∅
    S_opt ← ∅
    k = 2
    for all positive d ∈ D do
        a_d = {a ∈ A | a is exhibited by d}
        C ← C ∪ (d, a_d)
    end for
    remove redundant concepts from C
    C ← NOURINE-RAYNAUD(C)
    S_i ← S(c) for some c ∈ C
    while T_k > 0 do
        S_j ← SIMANNEAL(S_i)
        if fp_D(S_j) < fp_D(S_opt) or
           (|tp_D(S_j) − t| < |tp_D(S_opt) − t|
            and fp_D(S_j) = fp_D(S_opt) then
            S_opt ← S_i
        end if
        k ← k + 1
    end while
    return S_opt
end function
```

Figure 4. Specification Synthesis Algorithm. D is the input set containing labeled positive and negative samples, A is a set of behaviors produced via mining, and t is the threshold.

V. IMPLEMENTATION

To evaluate HOLMES's ability to effectively generalize behavioral specifications of malware, we implemented a proof-of-concept tool and ran it using real malware samples. The functionality of HOLMES is largely agnostic to the specific details of the behavior representation used, as well as the method used to generate execution traces. As trace collection and behavior graph construction are not the focus of this work, we save consideration of more sophisticated techniques in this area for future work. However, we note that the use of multi-path analysis tools [6], [7] to extract a more diverse set of behaviors from a sample may be fruitfully applied here, and will most likely result in higher-quality specifications, as the mining and synthesis algorithms will have access to a fuller set of malicious behaviors.

Toolkit for Dependence-Graph Construction: We perform dynamic analysis on our malicious and benign sample sets to extract a sequence of system call events with argument data for each sample. We use a version of Bindview's STrace for Windows [20], which we modified to provide more detailed argument information for the system calls. Collecting traces from malware must be performed with care, as it requires running the samples in a realistic en-

vironment, while ensuring that they cannot escape the trace collection environment. For this, we use two instances of QEMU [21]. The first instance runs Windows, and serves as a *victim* machine connected to a second instance. The second instance, called the *server*, ensures that none of the network traffic originating in the Windows victim is able to reach the real network, and also acts as a virtual "internet-inna-box" [22] to provide a realistic set of network services to the Windows victim. Our experiences indicate that this is a safe approach to extracting reasonable traces from many malware.

After collecting traces, we construct dependence graphs. The traces we parse range in size from several hundred kilobytes to several hundred megabytes, making this a demanding computation in terms of memory requirements. If the graph is constructed in memory as dependencies are inferred, the graph constructor frequently hits the 4 gigabyte limit common on 32-bit platforms. Therefore, our utility constructs the graph directly on the hard disk as the trace is parsed.

We use the dynamic dependence inference algorithm of Christodorescu *et al.* [16]. Initially, the dependence graph contains only nodes, one for each event in the trace. The algorithm inserts edges into this graph for every inferred dependence, and annotates both nodes and edges with constraints over system call arguments.

The algorithm infers two types of dependences between system calls. A *def-use dependence* expresses that a value output by one system call is used as input to another system call and is similar to the concept of def-use dependence from program analysis. A *value dependence* is a logic formula expressing the conditions placed on the argument values of one or more system calls, describing any data manipulations the program performs in between system calls. Currently, the algorithm computes an underapproximation of the dependence graph as it does not recover complex dependences.

To discover def-use dependences between events (system calls) in an execution trace, the algorithm uses argument values together with type information. Each argument of a system call has its type qualified to specify whether the argument is an *in* argument, an *out* argument, or an *inout* argument. The algorithm infers a dependence between two system calls (and creates a corresponding edge in the dependence graph) when the later system call has an *in* (or *inout*) argument with the same type and the same value as the *out* (or *inout*) argument of the earlier system call.

The algorithm infers value dependences between system call events that have string-valued arguments. Strings (and, equivalently, byte arrays) are interesting because the vast majority of system calls have at least one string argument through which a large amount of data is passed between a program and the OS. If a string-valued *out* (or *inout*) argument and a string-valued *in* (or *inout*) argument from a subsequent event share a substring of longer than a threshold (i.e., 12 in our experiments, to minimize false dependences), then a dependence edge is added from the first system call to the second. We enhanced the algorithm to annotate nodes in the dependence graph with the security labels from I, thus gaining a degree of information-flow sensitivity.

After inferring the behavior graphs, we prune them by removing nodes and edges that are not likely to correspond to malicious behavior, and merging redundant subgraphs. For example, we remove events relating to user interaction and display graphics, while we keep all events that correspond to resource acquisition, modification, transmission, and execution. To identify redundancies, we compare subgraphs and eliminate duplicated portions. Additionally, we collapse sequential reads and writes to stream resources (e.g., files and sockets) into a single "block" access. Intuitively, one read of 1000 consecutive bytes from a file is equivalent to one thousand consecutive reads of 1 byte each.

Toolkit for Behavior Mining and Specification Synthesis: After constructing the behavior graphs, we partition the malware graphs based on the behavioral similarity of their corresponding samples. Rather than relying on experimental techniques [10], [11], we utilize the manual efforts of the antivirus industry by using samples whose AV labels agree for multiple vendors. We used our own implementation of structural leap mining to mine significant behavior subgraphs from the malicious and benign graph database, because no existing leap mining implementation meets our functional needs (see III-B). Our implementation of the specification synthesis algorithm of Figure 4 is written entirely in Python. The algorithm fits cleanly within about 300 lines of code, including the Nourine-Raynaud algorithm [19] to compute the specification set.

VI. EVALUATION

We evaluated our behavior mining and specification synthesis algorithms on a corpus of real malware samples from a set of honeypots [23], and a representative corpus of benign applications. Our evaluation shows:

- The specifications synthesized by HOLMES from a known malware set allows the detector to reach an *86% detection rate over unknown malware with 0 false positives*. This is a significant improvement over the 40-60% detection rate observed from commercial behavior-based detectors, the 55% rate reported for standard commercial AV [24], and the 64% rate reported by previous research [5].
- HOLMES is efficient and automatic, constructing a specification from start to finish in under 1 hour in most cases. The longest execution time was caused by the mining algorithm, which took 12-48 hours[2] to complete for some network worms. *This is a significant*

[2]We are currently parallelizing this algorithm, and expect speedups that are almost linear in the number of processors (e.g., 6 times faster on an 8-core system.)

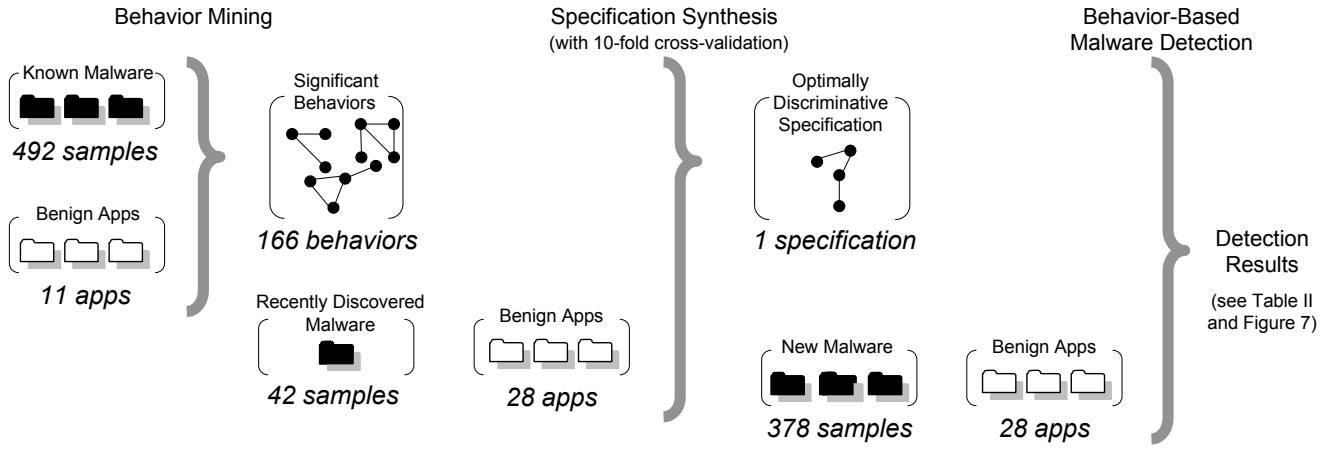


Figure 5. Our evaluation workflow and timeline, annotated with input-set sizes for a single fold.

improvement over the reported current time window of 54 days between malware release and detection by commercial products [25].

- The behavior-mining algorithm described in III finds approximately 50 malicious behaviors on average for a malware family, including some behaviors never previously documented by human analysts. The specification synthesis algorithm uses 166 malicious behaviors from 6 families to derive a specification containing 818 nodes in 19 concepts each with 17 behaviors on average, without introducing any false positives.

These results strongly support our claim that our algorithms produce near-optimally discriminative specifications.

Evaluation Input Selection: We collected 912 malware samples and 49 benign samples. The malware samples were obtained from a honeypot over a period of eight months [23], and consisted of samples from the following families: Virut, Stration, Delf, LdPinch, PoisonIvy, Parite, Bactera, Banload, Sality, DNSChanger, Zlob, Prorat, Bifrose, Hupigon, Allaple, Bagle, SDBot, and Korgo.

The benign samples used in our evaluation were selected to form a behaviorally diverse and representative dataset. It is important to provide the behavior extraction algorithm a set of benign applications that is representative of common desktop workloads, or the results that it produces may contain behaviors commonly percieved to be non-malicious. To this end, we included a number of applications that interact with the web, standard office applications, administrative tools, entertainment applications, installers, and uninstallers. We believe that the hardest task for a behavior-based malware detector is to distinguish between malicious programs and benign programs that are quite similar. For example, installers and uninstallers oftentimes perform the type of administrative routines used by malware upon initial infection (e.g. setting code to persist on restart, changing existing system configurations, etc.). Thus, we consulted an expert from the behavior-based antivirus industry for benign applications that are known to produce false positives and added them to our evaluation set.

Our benign corpus consists of the following applications: Adobe Acrobat Reader, Ad-Aware installation, Ad-Aware, AVG Antivirus, Bitcomet, Bitcomet uninstallation, Google Chrome, Google Chrome installation/configuration, Windows command prompt, FileZilla FTP server, Firefox, Windows FTP client, Google Desktop installation, Google Desktop, Internet Explorer, mIRC chat client, Outlook, Putty SSH client, Skype, Windows tasklist, Sysinternals TCPView, Thunderbird, Windows traceroute, Microsoft Visual Studio, WinRAR archive utility, Windows Media Player, Microsoft Word, YouTube video downloader, Ares Galaxy, FLV Player, PC Tools Antivirus, PowerISO, Siber Systems RoboForm, Almico Software SpeedFan, Javacool SpywareBlaster, Torrent Swapper, TrueCrypt, and Windows Essentials Media Codec Pack. We also evaluated on the installation routines for the following applications: Ares Galaxy, dBpoweramp Music Converter, Easy CD-DA Extractor, FLV Player, PowerISO, Primo PDF, Almico Software SpeedFan, Javacool Spyware blaster, System Mechanic, ClamWin Antivirus, and TrueCrypt.

Evaluation Methodology: We split the malware corpus into three sets: one for mining behaviors, one for driving the synthesis process, and one for testing the resulting specification. The set used in mining simulates the known malware at a given point in time and so it was fixed throughout the experiments. We selected six families for mining that contain behaviors which represent a wide range of known malware functionality. The malware set used in the synthesis step corresponds to malware discovered after the behavior mining, while the test set represents malware discovered after specification synthesis. Recall that our goal is to mine general family-level specifications robust to differences within variants of the same family. Thus, for the synthesis

and test sets, we divide samples from the remaining families into randomly-selected disjoint sets.

Since newly discovered malware and future, unknown malware are fungible, we perform 10-fold cross-validation to ensure that the synthesis process is not affected by any biases in our choice of malware sets. Graphically, the evaluation proceeded as shown in 5. The second and third steps shown in the figure, specification synthesis and behavior-based malware detection, respectively, correspond to a single fold of the cross-validation. Our final results are averaged over all ten folds.

For constructing the malware dependence graphs, we perform a single-path dynamic analysis of the malware samples for 120 seconds to collect a trace. This yields useful results because most malware is designed to execute at least part of its payload without additional input, and in as many environments as possible. We used executions of 120 seconds, as we found that two minutes is generally enough time for most malware to execute its immediate payload, if it has one. While some malware samples do not perform any malicious behavior in this period, we found that these samples usually wait for some external trigger to execute their payload (e.g. network or system environment), and will not perform any behaviors if left to execute without further action. Finally, we attempted to extract multiple execution traces from the samples used in our evaluation using previously-existing SAT-based concolic execution tools [6]. However, while we believe that this is a promising area with positive implications for our work, the performance of current tools does not scale to the extent required by our evaluation. Extracting between ten and twenty execution traces from a single sample can take on the order of days, whereas we need to evaluate hundreds of samples. See section VII for further discussion on this topic.

For benign samples (most of which are interactive applications), we extract a representative trace by interacting with it as a normal user would in a common environment, for up to 15 minutes. For example, if the application is a web browser, we direct it to popular websites that require more than just plain HTML (i.e. JavaScript, Flash, etc.). For installer applications, we run them to completion on the system using all of the default options. For applications reported to us as problematic by the AV industry, we try to reproduce any problematic executions that were described to us. In general, collecting a representative trace from benign application is more difficult than from malicious applications. We made a best-effort attempt to cover as many typical code paths as possible by using the benign application in a realistic desktop environment, but due to the well-known difficulty of this problem [6], [7], we can make no guarantees of total coverage.

To cluster the behavior graphs by perceived similarity, we consulted the McAfee, Kaspersky, and Microsoft malware classifications, and grouped those graphs given the same label. These particular antivirus engines were selected based on our experience that they tend to agree about labels more frequently than others. Each cluster of behavior graphs with the same label was then run through the leap-mining process.

Table II
DETECTION RATES OF COMMERCIAL BEHAVIOR-BASED DETECTORS AND HOLMES.

	Sana ActiveMDT	PC Tools ThreatFire	HOLMES
Detection rate	42.61%	61.70%	**86.56%**

A. Comparison with Commercial Products

Using the malware in our evaluation set, we evaluated the detection capabilities of two commercial behavior-based malware detectors, Sana's ActiveMDT [26] and ThreatFire by PC Tools [27]. Our goal was to determine how a detector using the specifications produced by HOLMES fares against state-of-the-art tools with behavior-based detection technology. The results shown in II indicate clear gains in detection capabilities using specifications produced with our technique.

While the technical details of ActiveMDT and Threatfire are proprietary, we suspect that the superior performance of our specifications is due to HOLMES's ability to quickly search for a disjunction of combinations of relatively simple software behaviors that describe the target malware. Due to the enormous state space of these disjunctions, and the difficulty of finding one that does not also specify benign applications, manual efforts to do so are unlikely whereas our probabilistic search finds a near-optimal candidate quickly. The difficulty of deriving a similar specification manually is highlighted by the size of our most effective specifications (~ 800 nodes).

B. Behavior Extraction Results

We ran HOLMES's behavior-mining algorithm over the known malware set, consisting of six malware families: Virut, Stration, Delf, LdPinch, PoisonIvy, and Parite. These families exhibit a wide range of malicious behaviors, including behaviors associated with network worms, file-infecting viruses, spyware, and backdoor applications. The set of benign applications used in behavior extraction are a representative subset of our full benign corpus: AVG Antivirus, Firefox, Internet Explorer, Skype, Windows task manager, Microsoft Visual Studio, Windows traceroute, WinSCP, RealVNC Server, and Sun Java 6 runtime environment installation.

The results of our behavior mining evaluation are presented in III. The second column, *# Mined*, corresponds to the number of patterns returned by the mining algorithm. The third column, *# Unique*, corresponds to the number of distinct behaviors returned by our behavior extraction algorithm. Because the semantics of system calls are not

Table III
RESULTS OF BEHAVIOR EXTRACTION ON 72 SAMPLES FROM SIX MALWARE FAMILIES.

Malware Family	*# Mined*	*# Unique*	*# Malicious*
VIRUT	67	19	14
STRATION	29	18	13
DELF	44	28	21
LDPINCH	75	14	10
POISONIVY	21	13	9
PARITE	51	17	10
Average	47.8	44.5%	70.0%

entirely independent [28], distinct sequences of system calls often have nearly identical semantic effect on the system, making this quantity relevant to our results. The fourth column, *# Malicious*, corresponds to the number of unique behaviors that are likely used to carry out malicious intent. In our experiments, some of the behaviors returned by HOLMES are not malicious in nature. For example, many of the malicious samples changed the system's random number generator seed, which none of the benign applications did. As a result, this pattern was returned for many of the families, despite the fact that by itself it is not a malicious behavior. Each average at the bottom of the table is based on the number of graphs in the column to its left; for example, 70.0% of the unique graphs mined (column 3) were malicious (column 4).

To compare the behaviors returned by HOLMES with the behaviors reported by professional analysts, we consulted the malware information databases maintained by Symantec [13], Kaspersky [29], and McAfee [30]. In almost all cases, the malicious behaviors returned by HOLMES correspond to those reported in the malware families we used. One interesting exception is the result of mining some our file-infecting viruses (Virut and Parite) and network worms (Stration and Delf). HOLMES found behaviors characteristic of spyware, such as remote process code injection and reconfiguration of browser settings. These behaviors were not reported by many of the databases we consulted, demonstrating the effectiveness of the algorithm. In a few cases, HOLMES did not return behaviors that were documented by the AV databases. For example, every database claims that Parite injects code into `explorer.exe` in order to stay memory-resident, but this behavior was not found by HOLMES in our evaluation. We suspect that many of these behaviors were not returned due to the limited coverage of the single-path dynamic analysis used to collect behavior information.

The results of our evaluation indicate that the behavior-mining algorithm described in III can be used as a powerful tool for automatically extracting useful information from malware code. Although the results leave room for improvement, we show in the following section that they are sufficient to construct near-optimal specifications.

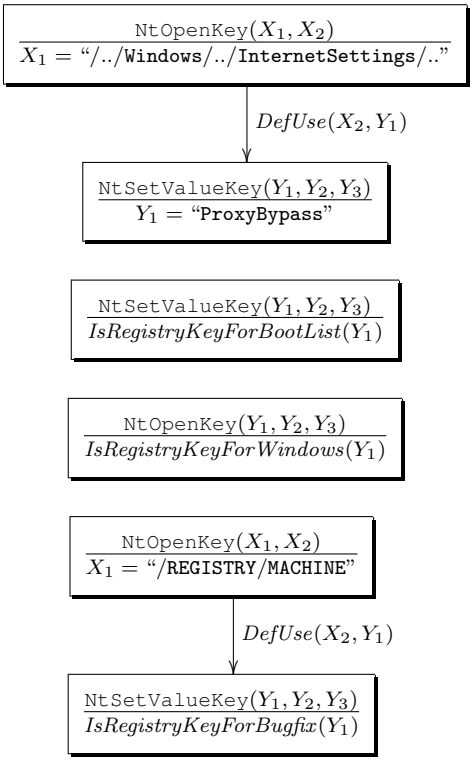


Figure 6. A representative concept derived by HOLMES, consisting of 4 significant behaviors (changing the browser proxy settings, changing the boot list, querying system information, and querying the bugfix information).

C. Specification Synthesis Results

We applied the specification synthesis algorithm given in 4 to the behaviors mined in VI-B. The synthesis algorithm relies on a training set of labeled samples (both known malicious and known benign) to search for an optimally discriminative specification. Note that this synthesis step does not add new behaviors to the set, it only combines the ones previously mined. The training set consisted of 28 common desktop applications and 42 malware samples from twelve families *not* used in the behavior-mining step: Bactera, Banload, Sality, DNSChanger, Zlob, Prorat, Bifrose, Hupigon, Allaple, Bagle, SDBot, and Korgo. To evaluate the performance of HOLMES on multiple datasets, we performed cross validation by partitioning the 420 *new* malware samples not used in mining into ten disjoint sets, running training cycles over each of them independently, and evaluating on the remaining sets not used in each round of training. We did this for multiple threshold values t to explore the tradeoff between true positive and false positive rate.

The results of our evaluation are presented in 7, with an example of a concept derived by SpecSynth shown in 6. Note that the graphs in Figure 6 do not constitute an entire specification, but merely one *disjunct* among many in a much larger specification (see Definition 4 for clari-

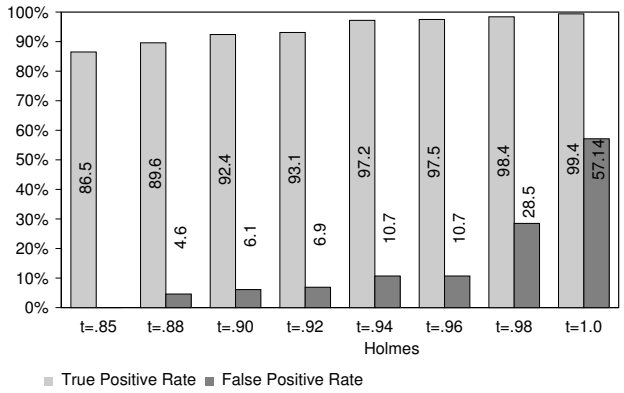


Figure 7. Detection results for multiple synthesized specifications show a clear tradeoff between true positives and false positives.

fication on this point). Each percentage given in Figure 6 is the average over the ten folds. With a threshold value of $t = 0.85$, HOLMES was able to construct a specifications for each fold that covered none of the benign samples from our set, and nearly all (86.56% avg.) of our unknown malware. The variance between results for different folds was small, in the range of tenths of a percentage point. Raising the threshold resulted in a slightly higher true positive rate, but at the expense of a much higher false positive rate. These trends underscore the inherent difficulty of constructing good discriminative specifications.

To summarize, these results indicate that our technique for mining discriminative specifications is able to effectively identify the key components required to describe a given class of software in useful, realistic settings.

D. Performance and Scalability

In our experience, the runtime performance of the behavior mining algorithm varies significantly between malware families. For some families, particularly spyware (LdPinch) and file-infecting viruses (Virut and Parite), behavior extraction required between ten and sixty minutes to complete. For families that exhibit repeated behaviors through the course of their execution, such as network worms (Stration and Delf), the dependence graphs can be quite large (10,000-20,000 nodes and edges), causing the behavior mining algorithm to run between 12 and 48 hours before completion. The worst-case complexity of this operation is exponential. However, this performance is rarely encountered, and can be mitigated by accepting a tradeoff in the quality of the result for a shorter running time [8]. In practice, it is difficult to precisely characterize the expected complexity of behavior mining. The factors that affect performance the most are the size of the input behavior graphs, and the similarity between the malicious and benign graphs. When the two sets are highly dissimilar, the algorithm can quickly find subgraphs that maximize entropy between the two sets; when this is not the case, the algorithm must search for larger subgraphs, and suitable results take a long time to produce. Currently, this component is the largest performance bottleneck in our workflow. However, it is fully unsupervised, and can thus run concurrently with other analyses, in the "background". We are currently investigating the use of probabilistic algorithms to improve the performance of behavior mining, as well as parallelizing the algorithm. We expect significant performance gains from these enhancements.

On the other hand, the performance of HOLMES is quite good. On average, the specifications presented in 7 were found in under one minute (the single exception to this took approximately 80 seconds), although the algorithm was allowed to run for 30 minutes to search for a potentially better solution. This corresponds to a tradeoff present in our technique: if allowed to run to completion, HOLMES is guaranteed to find the optimal solution. However, it will usually find a good solution quickly, so the user is ultimately left with the choice of performance or solution optimality.

One concern is the potentially high cost of generating all concepts for a given dataset using NOURINE-RAYNAUD [19]. In the worst case, this computation can take exponentially long in the size of the dataset. However, because we remove all redundant concepts in the first round of this computation (this is discussed in more detail in IV), the size of our input to NOURINE-RAYNAUD does not necessarily scale linearly with the number of samples in the training set. Rather, it scales with the *behavioral diversity*, or the number of mined behaviors shared by the samples, of the training set. For our evaluation, it took less than one second to compute all of the concepts for the samples, and we expect that most usages of HOLMES will meet with similar results. However, to address the possibility of encountering dense concept sets, we are currently evaluating the use of binary decision diagrams (BDD's) for concept enumeration [31].

VII. DISCUSSION

Our results indicate that malicious behaviors are shared across multiple families, and that our algorithms as implemented in HOLMES are successful in identifying and isolating these behaviors. HOLMES combines these discriminating behaviors to form a specification that can be used in the detection of unknown malware with a 86% true positive rate and 0 false positives. As with any empirical evaluation, there are limitations that must be considered when interpreting the results.

First, we consider threats to construct validity, and in particular to our choice of behavioral model. We combine data flows connecting system calls with security labels to allow us to characterize the information flows enabled by malicious programs. Previous work has shown that using data flows to describe malicious behavior is a powerful approach [15], [5], [3] and that the system-call interface is the right abstraction for characterizing user-space malware [11],

[32]. This of course does not cover all malware types, some of which might produce other event types (e.g., browser spyware interacts primarily with the browser API). We designed HOLMES to be orthogonal to the actual semantics of the events and constraints that make up behavior graphs. Thus, deriving specifications for other classes of malware not covered here can use our algorithms, given an appropriate behavior-graph construction mechanism.

Threats to internal validity relate to our system's ability to construct accurate and precise behavior graphs, in terms of events, dependencies. and security labels captured. Missing events, dependencies, or labels can prevent HOLMES from finding the optimally discriminating specification, even when given sufficient time. HOLMES builds on existing work for creating the behavior graph and will benefit from more powerful tools that use, for example, dynamic taint tracing [15], [5] and multipath analysis [7]. Currently, the set of security labels we use is limited to our perusing of the Microsoft Windows documentation website. Deriving a complete and accurate set of security labels is an open research problem, as is measuring the impact of such a set on the specification-synthesis process in HOLMES.

Finally, there is the question of whether the malware and benign sets are representative of real-world scenarios. We analyzed only 912 malware samples and only 49 benign programs, and cannot claim that the results generalize to other settings. However, the malware samples are real, have been collected over a period of 8 months using an Internet-connected honeypot, and have exhibited a wide variety of behaviors when we analyzed them. Similarly, the benign applications are some of the most popular applications on the Microsoft Windows platform. Furthermore, given that our synthesis algorithm is adaptive by design, improved specifications can be produced when completely new malware or new benign applications appear, without repeating the analysis of any previous samples.

VIII. RELATED WORK

Malware analysis: Our work continues a research tradition of using program analysis and statistical reasoning techniques to analyze and prevent malware infection. Several researchers have investigated the problem of clustering malware; Bailey *et al.* used behavioral clustering to resolve inconsistencies in AV labeling schemes [10], and Rieck *et al.* proposed a classification technique that uses support vector machines to produce class labels [11] for unknown malware. Our work is complementary to automated clustering efforts, as we can use the results of these techniques to create initial sample partitions for behavior extraction (see Section III).

Our work produces precise specifications of malware families from which existing behavioral detection techniques can benefit. Two detectors mentioned in the literature, those of Kolbitsch *et al.* [5] and Christodorescu *et al.* [3], use notions of software behavior that correspond very closely to our own (see Section III), and could thus make direct use of our specifications. Additionally, commercial behavior-based detectors such as Threatfire and Sana's ActiveMDT could potentially use the behavioral specifications produced by HOLMES; as we show in Section VI, doing so may reduce the amount of time needed to produce reliable specifications.

Recently, Kolbitsch *et al.* explored the problem of creating behavioral malware specifications from execution traces [5]. They demonstrated a technique for producing *behavior graphs* similar to those used in our work, but with a more sophisticated language for expressing constraints on event dependencies, and showed that their specifications effectively detect malware samples with no false positives. One of their key observations was the necessity of complex semantic dependencies between system call events, without which false positive rates are unacceptably high. We also observed this phenomenon in our experiments, and found the need to introduce heuristic annotations (information flows) to our graphs to effectively extract significant behaviors from our corpus. Aside from differences in the type of behavior representation graph used, their specification construction algorithm differs from ours in two ways. First, they make no attempt to *generalize* specifications to account for variants within the same family, or of the same type. Second, they do not discriminate their malicious behaviors from those demonstrated by benign programs, so their technique does not present a safeguard against false positives. Thus, their work is complementary to ours: their specifications can serve as a starting point for SPECSYNTH to automatically refine for increased accuracy and coverage of variants.

Others have taken different approaches to deriving general behavioral specifications robust to differences between malware variants of the same family. Cozzie *et al.* [33] describe a system which fingerprints programs according to the data structures that they use, and show that these fingerprints can be used to effectively detect malware on end-hosts. Our behavioral specifications differ fundamentally from theirs. The elements used to build our behavioral specifications, namely system calls and constraints on their arguments, are program actions that directly contribute to the malicious nature of the malware. In some sense, the malware cannot induce its harmful effect on the system without these elements. On the other hand, the data structures used to build the specifications of Cozzie *et al.* are not inextricably related to the malicious nature of the samples. The practical upshot of this difference is that our specifications may be more difficult to evade, as doing so would require changing the system call footprint of the malware rather than the data structure usage patterns. Stinson and Mitchell [34] identified bot-like behaviors by noting suspicious information flows between key system calls. Egele *et al.* describe a behavioral specification of browser-based spyware based on taint-tracking [4], and Panorama uses whole-system taint analysis in a similar vein to detect more general classes of spyware [15].

Specification mining: Several researchers have explored mining for creating specifications to be used for formal verification or property checking. The seminal work in this area is due to Ammons *et al.* [35], who mined automata representations of common software behavior to be used for software verification. Their technique finds commonalities between elements of a given set of programs, but unlike our work, does not discriminate from a second set of programs. Recently, Shoham *et al.* [36] and Sankaranarayanan *et al.* [37] both mined specifications of API usage patterns. Like that of Ammons *et al.*, their work was geared towards the ultimate goal of detecting accidental flaws in software, rather than contrasting different types of software behavior (e.g. malicious vs. benign). Christodorescu *et al.* [16] mined discriminitive specifications for behavior-based malware detection and suggested the use of simple security labels on arguments. However, their technique produces specifications that discriminate a single malware sample from a set of benign applications. Our work is a generalization of this setting, where a specification is produced to discriminate a malware family from a set of benign applications. Furthermore, we incorporate a tunable notion of optimality to guide a search through the full space of candidate specifications for the most desirable solution. This tunable notion of optimality, along with our use of statistical sampling techniques, makes HOLMES scalable to a larger range of realistic settings (see VI).

Concept analysis: Formal concept analysis was introduced by Rudolf Wille in the 1980's as part of an attempt to make lattice theory more accessible to emerging fields [9]. It has since been applied to a number of problems in computer science, in particular software engineering. Ganapathy *et al.* [38] used concept analysis to mine legacy source code for locations likely to need security retrofitting. Although our work is similar to theirs in the use of concept analysis to secure a software system, there are considerable differences in the settings in which our work is appropriately applied, and the output of our analysis. The work of Ganapathy *et al.* is meant to be applied to a body of non-adversarial source code to derive suggestions for likely places to place hooks into a reference monitor. In contrast, our work is applied to adversarial binary code to derive global specifications of program behavior. Others have applied it to various problems in code refactoring [39], [40] and program understanding [41]. Our work contributes to the state of the art in this area by demonstrating a novel application of concept analysis, and showing how the associated drawbacks, namely complexity explosion, can be mitigated with the appropriate use of statistical sampling techniques. This technique may be of independent interest for other uses of concept analysis that suffer from the prohibitive cost of searching the concept space.

IX. CONCLUSION

We described a technique for synthesizing *discriminative specifications* that result in behavior-based malware detectors reaching 86% detection rate on new malware with 0 false positives. Framing specification synthesis as a clustering problem, we used concept analysis to find specifications that are optimally discriminative over a given distribution of programs. We showed how probabilistic sampling techniques can be used to find near-optimal specifications quickly, while guaranteeing convergence to an optimal solution if given sufficient time. The synthesis process is efficient and constructs a specification within 1 minute. Our prototype, called HOLMES, automatically mines behaviors from malware samples, without human interaction, and generates an optimally discriminative specification within 48 hours. Preliminary experiments show that this process can be reduced to a fraction of this time by using multi-core computing environments and leveraging the parallelism of our mining algorithm. HOLMES improves considerably on the 56-day delay in signature updates of commercial AV [25].

Acknowledgments: There are a number of people we would like to thank for their invaluable contributions throughout the course of this work: Drew Davidson, Andrew Goldberg, Bill Harris, J.R. Rao, Angelos Stavrou, Hao Wang, and the anonymous reviewers for their helpful comments.

REFERENCES

[1] B. Acohido and J. Swartz, *Zero Day Threat: The Shocking Truth of How Banks and Credit Bureaus Help Cyber Crooks Steal Your Money and Identity*. Union Square Press, April 2008.

[2] D. Perry, "Here comes the flood or end of the pattern file," in *Virus Bulletin (VB2008)*, Ottawa, 2008.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'06)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–46.

[4] M. Egele, C. Kruegel, E. Kirda, H. Yin, , and D. Song, "Dynamic spyware analysis," in *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX'07)*. USENIX, 2007, pp. 233–246.

[5] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th USENIX Security Symposium (Security'09)*, August 2009.

[6] D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome, P. Poosankam, D. Song, and H. Yin, "BitScope: Automatically dissecting malicious binaries," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-07-133, Mar. 2007.

[7] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*. IEEE Computer Society, 2007, pp. 231–245.

[8] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. New York, NY, USA: ACM Press, 2008, pp. 433–444.

[9] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts," *Ordered Sets*, 1982.

[10] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*. Springer, 2007, pp. 178–197.

[11] K. Rieck, T. Holz, C. Willems, P. Dussel, and P. Laskov, "Learning and classification of malware behavior," in *Proceedings of the 5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'08)*. Springer, 2008, pp. 108–125.

[12] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 319–329.

[13] "Symantec security response," http://www.symantec.com/business/security_response/index.jsp.

[14] P. Bremaud, *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer, January 2001.

[15] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[16] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*. New York, NY, USA: ACM Press, 2007, pp. 5–14.

[17] Microsoft Corporation, "MSDN Library," http://msdn.microsoft.com/en-us/library/default.aspx.

[18] R. M. H. Ting and J. Bailey, "Mining minimal contrast subgraph patterns," in *Proceedings of the 2006 SIAM International Conference on Data Mining (SDM'06)*. SIAM, 2006.

[19] L. Nourine and O. Raynaud, "A fast algorithm for building lattices," *Inf. Process. Lett.*, vol. 71, no. 5-6, pp. 199–204, 1999.

[20] BindView, "Strace for NT," http://www.bindview.com.

[21] F. Bellard, "QEMU, a fast and portable dynamic translator," http://fabrice.bellard.free.fr/qemu/.

[22] I. Whalley, B. Arnold, D. Chess, J. Morar, A. Segal, and M. Swimmer, "An environment for controlled worm replication and analysis," in *Virus Bulletin Conference*, 2000.

[23] "SRI honeynet and malware analysis," http://www.cyber-ta.org/Honeynet.

[24] E. Larkin, "Top internet security suites: Paying for protection," *PC Magazine*, January 2009.

[25] Damballa, Inc., "3% to 5% of enterprise assets are compromised by bot-driven targeted attack malware," Mar. 2008, Press Release.

[26] "Sana security active malware defense technology center," http://www.sanasecurity.com/amdtc.

[27] "Threatfire anti-malware," http://www.threatfire.com.

[28] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM Press, 2002, pp. 255–264.

[29] "Malicious programs descriptions search," http://www.viruslist.com/en/virusesdescribed.

[30] "McAfee threat center," http://www.mcafee.com/us/threat_center.

[31] S. Yevtushenko, "Computing and visualizing concept lattices," Ph.D. dissertation, Technischen Universitat Darmstadt, Darmstadt, Germany, Oct. 2004.

[32] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P'01)*, 2001.

[33] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceedings of the 8th USENIX Symposium on OS Design and Implementation (OSDI'08)*, 2008.

[34] E. Stinson and J. C. Mitchell, "Characterizing bots' remote control behavior," in *Proceedings of the 4th GI International Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'07)*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 89–108.

[35] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. New York, NY, USA: ACM Press, 2002, pp. 4–16.

[36] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2007 (ISSTA'07)*. New York, NY, USA: ACM Press, 2007, pp. 174–184.

[37] S. Sankaranarayanan, F. Ivanči, and A. Gupta, "Mining library specifications using inductive logic programming," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. New York, NY, USA: ACM Press, 2008, pp. 131–140.

[38] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining security-sensitive operations in legacy code using concept analysis," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 458–467.

[39] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," in *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 1998, pp. 99–110.

[40] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, 2001.

[41] ——, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495–509, 2003.