Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors

Somesh Jha*, Matthew Fredrikson*, Mihai Christodoresu†, Reiner Sailer‡, Xifeng Yan§
*University of Wisconsin–Madison, †Qualcomm Research Silicon Valley, ‡IBM T.J Watson Research Center,
§University of California–Santa Barbara

***Abstract*—Behavior-based detection techniques are a promising solution to the problem of malware proliferation. However, they require precise specifications of malicious behavior that do not result in an excessive number of false alarms, while still remaining general enough to detect new variants before traditional signatures can be created and distributed. 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 stochastic optimization, scales to large classes of programs. When this work was originally published, the technique yielded favorable results on malware targeted towards workstations (~86% detection rates on new malware). We believe that it can be brought to bear on emerging malware-based threats for new platforms, and discuss several promising avenues for future work in this direction.**

I. Introduction

Although behavior-based malware detection techniques continue to grow in popularity, the predominant technique for detecting malware remains signature-based scanning. Meanwhile, malware authors continue to perfect the science of quickly generating large numbers of signature-resistant variants, giving them the upper-hand. Kaspersky's most recent annual report documented an average of roughly 200,000 new samples each day [1], *up 60% from the 125,000 daily new samples seen at the end of the previous year*. This leaves the *window of vulnerable exposure*—the period in which a new variant exists but suitable methods for detecting it do not—wide open.

Behavior-based detectors have been proposed as a remedy to this problem. It is more challenging for an adversary to mutate the behavior of a malware while still maintaining its intended payload, so in principle this approach holds promise. However, the challenge remains of constructing effective specifications of malicious behavior, i.e., those that are *general* enough to detect new variants but *specific* enough to avoid false positives. The most reliable technique to date for generating behavioral specifications treats the practice as more of an art than a science, relying almost entirely on manual analysis and human ingenuity—an expensive, time-consuming, and error-prone process that provides no quantitative guarantees of effectiveness. 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 and errors.

We make the observation that behavioral specifications are best viewed as a form of *discriminative specification*. A discriminative specification describes the unique properties of a given class, in contrast to the properties exhibited by a second mutually-exclusive class. This paper presents an automated technique that combines program analysis, graph mining, and stochastic optimization to synthesize malware behavior specifications. We represent program behaviors as graphs that are *mined* for discriminative patterns. As there are many ways in which malware can accomplish the same goal, we use these patterns as building blocks for constructing discriminative specifications that are general across variants, and thus robust to severe obfuscations. Furthermore, because our graph mining and synthesis procedures are agnostic to the details of the underlying graph representation, our technique will benefit from ongoing advances in program analysis and malware research.

This paper presents the following contributions:

- We divide the specification problem into two automatable tasks: (1) mining discriminative behavior patterns from a set of samples and (2) synthesizing a discriminative specification from multiple sets of mined behaviors. These tasks mirror a human analyst's workflow, where new malware samples are first analyzed for potentially malicious behaviors, and subsequently merged with known behaviors to construct a good specification.
- We discuss a specification synthesis algorithm that mirrors this workflow, and its evaluation on a corpus of desktop malware. Our experience shows that it automatically identifies both behaviors documented by AV-industry analysts as well as new behaviors, provides favorable detection statistics, and scales agreeably with reasonable parameters.
- We discuss several avenues for future research that are likely to yield useful results when applying our technique to the newer, emerging malware-related threats that are prevalent on the web and mobile platforms.

The remainder of the paper is organized as follows. In Section 2, we present the algorithm. In Section 3 we briefly discuss performance. In Section 4 we discuss promising future avenues for research, and in Section 5 we discuss related work.

978-1-4799-2535-3/13/$31.00

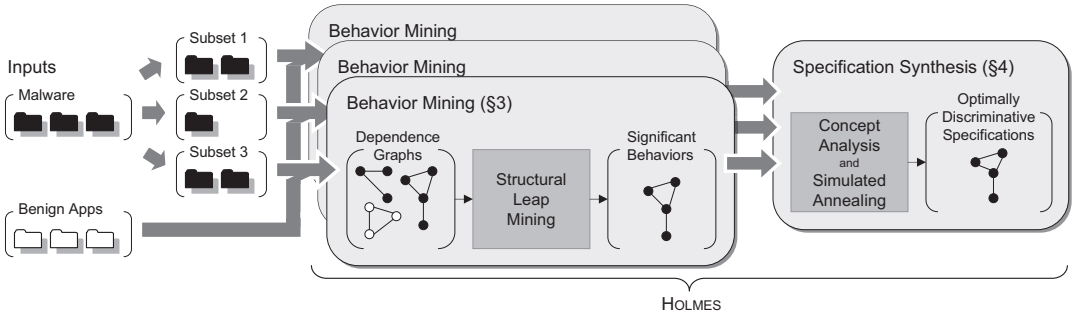


Figure 1. HOLMES combines program analysis, structural leap mining, and concept analysis to create optimal specifications.

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 $IsRegistryKeyForBugfix$.

II. THE ALGORITHM

A. Background & Overview

For the purposes of this work, we define the behavior of a program as its effect on the state of the system over which it executes. On modern desktop platforms, including the Windows operating system to which we focus our effots, the portion of the system state that is pertinent to malicious behavior is accessible only via system calls. Thus, we define a representation of software behavior in terms of the system calls, and their inter-relationships, made by a single program executing on the host.

Definition 1 (Behavior Graph): A behavior graph g is a structure (V, E, α, β) where:

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

We enhance our dependence graphs by assigning specialized labels to particular files, directories, registry keys, and devices based on their significance to the system. The Microsoft Windows documentation [2] lists a large number of files, directories, registry keys, and devices that are relevant to system security, stability, and performance. Table I lists 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. They allow our algorithm to recognize as potentially malicious otherwise benign behaviors, such as modifying the registry or reading certain files. For example, programs that modify bugfix registry settings or executable files are more likely to engage in further, more elaborate malicious behaviors than those that do not. Several example behavior graphs are given in Figure 2.

Using this representation, we wish to derive a behavioral specification that is descriptive of a given set of programs (the malicious set of programs) but does not describe any program in a second set (the benign set of programs). Our experience suggests that any behavior common to the *entirety* of a diverse set of malware samples will not be useful in the manner required of a malware specification, as it is likely to show up in benign programs as well (e.g., common file operations, memory allocation, etc.). Thus, before searching for discriminative patterns, we partition the malware set by family to "draw out" the meaningful malicious behaviors. This leads us to the high-level workflow of our technique, which is presented in Figure 1 and proceeds as follows:

I. The known, labeled malware is divided into disjoint families using existing antivirus-industry labels.

II. Using existing techniques for dependence-graph construction [3], a graph is constructed for each malware and benign application to represent its behavior.

III. The *significant* behaviors specific to each malware family are mined.

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* [4] to identify multiple distinct graphs that are present in the behavior graphs of the malware and absent from those of the benign set.

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

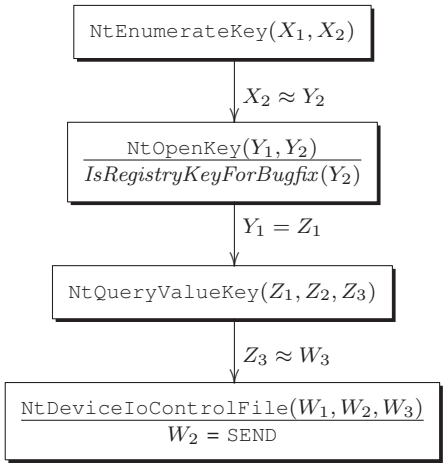
Two significant behaviors can be combined either by merging them into one significant behavior that is more general, or by taking the union of the two behaviors. We use *concept analysis* to identify the behaviors that can be combined with little or no increase in the coverage rate of the negative set, while maintaining the coverage rate of the positive set. As the space of possible specifications is very large, we use probabilistic sampling to approximate the optimal solution.

We now present each step in more detail as it applies to the construction of a signature for the LDPINCH family. According to the Symantec Security Response [5], 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, which it sends to a remote server.

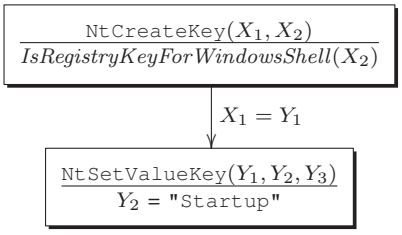
B. Behavior Mining

The first step of the behavior-mining algorithm extracts portions of the behavior graphs of programs from the malware that correspond to behavior that is *significant* to the programs' intent. To quantify the significance of a graph g, we use the *information gain* between the malicious and benign sets; information gain characterizes the degree to which g predicts membership in either set via the subgraph relation. This makes the goal of significant graph mining conceptually straightforward: find a set of distinct subgraphs among the malware families that optimizes information gain.

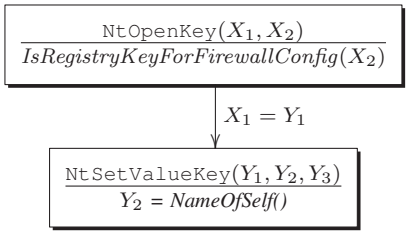
We use *structural leap mining* [4] to achieve this goal. Structural leap mining exploits the correlation between structural similarity and similarity in information gain value to quickly prune the candidate search space. Specifically, candidate subgraphs are enumerated from small to large size in a search tree and their information gain value 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 randomly "leaped over" since it is likely they will share similar information gain. This *horizontal* pruning is effective in pruning subgraphs that represent similar behaviors. Based on the observation



(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.

that a set of malware is not likely to be sufficiently characterized by a single significant subgraph, we modified the original leap mining algorithm to return the top k significant subgraphs. To avoid redundancy among these subgraphs we require that each pair in our top-k list has a structural similarity that falls below a predefined threshold. For further details, we refer the reader to the original papers [6], [4].

Figure 2 presents a small portion of the graph mining output produced by applying our algorithm to the LDPINCH family and some common benign programs. 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. At the time this work was originally published, human analysts had

not previously reported this behavior for LDPINCH.

C. 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. We define a behavior specification to be a *disjunction* of a set of *conjunctions* of behavior graphs. A program matches a specification if it exhibits *all* of the behaviors in *any* of the disjuncts. The behavior graphs used in each conjunction are those produced by the leap mining algorithm. Considering the behavior graphs listed in Figure 2, we could create a specification for the LDPINCH family consisting of a single disjunct by conjoining each graph. This specification 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 an entire family of is often too specific to capture the wide range of behavior observed across other families, or even unseen instances in the same family.

To generalize such narrowly-defined specifications, we developed an algorithm for generating specifications composed only of the significant behaviors descriptive of a given corpus of programs as a whole. The algorithm uses *simulated annealing* to randomly search the space of possible specifications for one that maximizes a suitable objective, which we have defined as:

$$\mathsf{Obj}(\mathit{Spec}, M, B) = \begin{cases} \mathit{fp}(\mathit{Spec}, B) & \text{if } \mathit{tp}(\mathit{Spec}, M) \geq t \\ \infty & \text{otherwise} \end{cases}$$

Here, *Spec* is a candidate specification, M is the set of malicious samples, and B is the set of benign samples. *fp* and *tp* measure the false and true positive rates of *Spec*, respectively. t is a parameter, which corresponds to a targeted false positive rate: the function Obj looks for specifications that produce a given true positive rate t, while minimizing the false positive rate. An optimal value is 0, and any specification that fails to meet the target true positive rate is given the value ∞.

The simulated annealing algorithm works by local search. Starting from an initial specification, it evaluates the objective function on each candidate, and moves to a neighboring specification by adding or removing a random disjunct. Neighbors are drawn using the standard Metropolis sampler [7]. The sampler is defined in terms of a *cooling parameter* that decreases the sampler's probability of acceptance inverse-proportionally to the number of iterations that have transpired. For additional details on the synthesis algorithm, refer to the original paper [6].

III. EVALUATION ON DESKTOP MALWARE

In 2009, we evaluated the behavior mining and specification synthesis algorithm on a corpus of desktop malware samples that was representative of common threats at the time. We collected 912 malware samples and 49 benign samples. The malware samples were obtained from a honeypot over a period of eight months [8], 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.

For leap mining, we used a set of benign samples 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 workloads for the platform on which the signatures will be used, or the results may contain behaviors commonly perceived to be benign. To this end, we included a number of applications that interact with the web, standard office applications, administrative tools, entertainment applications, installers, and uninstallers. The hardest task for a behavior-based malware detector is to distinguish between malicious 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.

To prepare the data, 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.

To construct 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 en-

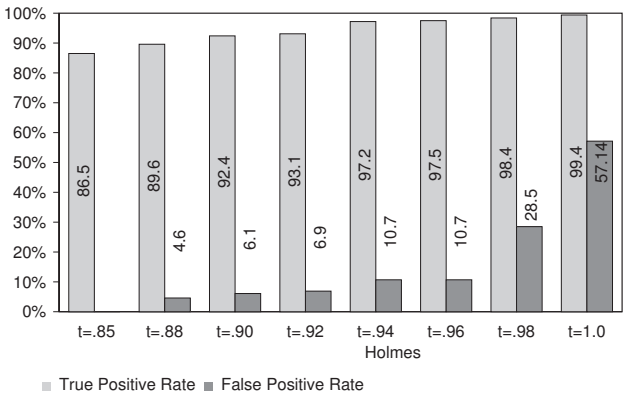


Figure 3. Detection results for multiple synthesized specifications show a clear tradeoff between true positives and false positives.

vironments as possible. We used executions of 120 seconds, as we found that this is enough time for most malware to install itself and 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 [9]. 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.

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 [9], [10], we can make no guarantees of total coverage.

Our results are summarized in Figure 3. Each percentage given in Figure 3 is the average over ten folds. With a threshold value of $t = 0.85$, HOLMES was able to construct a specification 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 the highlights from our results:

- The specifications synthesized by HOLMES from a known malware set allow 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 [11], and the 64% rate reported by previous research [12].
- 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 to complete for some network worms. *This is a significant improvement over the reported current time window of 54 days between malware release and detection by commercial products* [13].
- The behavior graph-mining algorithm 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.

For further details, consult the original paper [6].

IV. DISCUSSION AND FUTURE WORK

A. Experimental and structural concerns

Our experimental results indicated that discriminative malicious behaviors in desktop malware are shared across multiple families, and that they can be successfully identified and isolated using mostly-automated techniques—our algorithm combined 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 producing and interpreting the results of any study in specification synthesis.

First, we consider methodological 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 [14], [12], [15] and that the system-call interface is the right abstraction for characterizing user-space malware [16], [17]. 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 leverage our algorithms, given an appropriate behavior-graph construction mechanism. However, when extending the approach to new platforms and types of behaviors not considered in our original work, it will be necessary to consider alternative representations that might be more suitable. For example, information flow relations that specify allowed disclosures on web and mobile platforms seem increasingly pertinent in fighting common threats on these platforms. This is a promising area of future research, which we discuss further below.

Methodological threats to *internal validity* relate to the system's ability to construct accurate and precise behavior graphs, in terms of events, dependencies, and security labels. Missing events, dependencies, or labels can prevent a specification from finding the optimally discriminating specification, even when given sufficient time. Our algorithm builds on existing work to create the behavior graph, and although it may have benefitted from more powerful tools that use, for example, dynamic taint tracing [14], [12] and multipath analysis [10], the tools we used were trusted as "standard" for that platform at the time. The set of security labels we used to annotate our specifications was derived from the Microsoft Windows documentation, and our discussions with malware analysts who had been working on that platform for many years. While we were fortunate to have worked on a platform with a large base of existing tools and commonly-accepted analysis techniques, those who wish to bring a similar technique to bear on new platforms will face more of a challenge. They will need to find or develop reliable ways of obtaining and building on the information necessary to represent malicious behaviors of interest on the target platform, and demonstrate that their techniques for doing so are valid. This concern amounts to one of the most immediate problems in need of attention in this area.

Finally, there is the question of whether the malware and benign sets used to evaluate the system are representative of real-world scenarios. In our evaluation, we analyzed only 912 malware samples and 49 benign programs, and in principle cannot claim that the results generalize to other settings. However, the malware samples were "real", in the sense that they were collected "in the wild" from a set of publicly-accessible honeypots over an eight-month period. Similarly, the benign applications we used were some of the most popular applications on the Microsoft Windows platform at the time, the reasoning being that a false positive is actually a *relative* notion that is dependent on the likelihood that someone will actually use an application that causes a specification to erroneously raise an alarm. Our goal was to make the synthesis algorithm *adaptive by design*, so that new specifications can be produced when completely new malware appear or users tend towards benign workloads with different characteristics; as long as the training and evaluation sets are "representative" of malicious and benign workloads common on the current platform, then the specifications should be effective on that platform. Thus, we believe that the data used for future experiments in this area must be given careful thought to demonstrate that any performance claims made of the algorithm are not merely the result of an underlying bias in the algorithm or behavior representation. In general, the algorithm should be evaluated on a wide range of different *types* of malware and benign workloads, if not multiple platforms. For example, it would be very compelling to show that a general algorithm for synthesizing mobile malware specifications works similarly on both Android and Windows Phone.

B. Efficient behavior matching

One problem that remains pertinent to behavior-based malware detection is that of matching a specification to an executable's runtime trace in real time. In work subsequent to this, we found that this problem is in fact NP-Complete [18], with the time complexity scaling in both the size of the specification (its number of edges and nodes) and the length of the runtime trace. The space complexity is asymptotically identical to the time complexity. The NP-Completeness of the problem implies that backtracking is necessary in an offline setting, and in dynamic settings this translates into a requirement of maintaining, in memory, a potentially exponential number of *partial matches* between the specification and trace. As new trace events are observed, each partial match must be extended and re-checked, leading to both time and space infeasability for even moderately-sized specifications and traces.

This creates an immediate problem for anyone wishing to actually make use of the specifications mined by techniques similar to those presented in this paper—while it may be feasible and fruitful to use them for offline analysis, it is not clear that they can be used effectively for real-time malware behavior detection and blocking. Future work in this area will need to address this problem convincingly, with concrete demonstrations of feasibility on relevant platforms and real-world malware samples. Malware detectors must remain *sound*, in the sense that they will never fail to flag a malicious program; this precludes any sort of approach that puts a hard limit on the number of partial matches that will be maintained by the program, as "forgetting" a partial match might result in a missed detection. Several heuristic approaches come to mind, such as using platform-specific information to eliminate certain partial matches. For

example, any partial match that requires an open file handle to complete can be forgotten once that file handle has been closed. In the same paper containing our proof, we discuss a few possible strategies for approximating the matching problem without sacrificing soundness. However, these result in an increased false positive rate, and it is challenging to derive a precise upper-bound on the number of false positives that will be introduced on real platforms, given real traces. Any future work that attempts to approximate the matching procedure will need to include extensive empirical results on the false-positive trade-off. This seems to be an ideal opportunity for industrial collaboration, as industrial labs have unprecedented access to relevant, emerging malware threats.

C. Efficient Synthesis and advances in pattern mining

Structural leap mining [4], as used in this work, is faster than generic frequent graph pattern mining as it only finds subgraph patterns that are unique in malware. However, due to the complexity of isomorphism testing and the inelastic pattern definition, both algorithms share the same weakness: Patterns with slightly different structure are often missed. In our recent work [19], we introduced the concept of proximity pattern, which is defined as a set of labels that co-occur in a neighborhood. Proximity pattern relaxes the rigid structure constraint of frequent subgraphs, while introducing connectivity to frequent itemsets. Empirical results show that it can reveal more suspicious behaviors, thus being able to improve malware detection. Its mining algorithm also runs orders of magnitude faster than the original leap mining algorithm, making real-time malware behavior mining practical, as well as significantly increasing the size of the datasets on which our algorithm is relevant.

D. New threats and platforms

Since the original publication of this algorithm, the types of commonly-encountered malware threats have changed significantly, as well as the platforms on which they run. When we carried out this work and the subsequent experiments, malware written for Windows was so much more prevalent than that found on other systems that one rarely heard of countermeasures tailored to other systems. The malware targeting Windows was just beginning to focus on exfiltration attacks, with the rise of online banking use driving the application of keyloggers and other "snooping" routines in common desktop malware. This trend has continued, and moved onto new platforms such as mobile, where families such as ANDROID.BMASTER and LUCKYCAT are designed to exfiltrate a wide range of personal information to remote sites. Adapting specification synthesis to address these threats remains an interesting challenge, as many benign mobile applications engage in similar behavior. Many of the specifications our algorithms produced to identify exfiltration threats on the desktops relied on *preparation* steps taken by the malware at installation time, e.g., adding whitelist entries to the local firewall. Discriminative specifications for mobile threats may have to consider factors that were not present in the desktop setting, such as interaction with the user (to permit a benign flow) or "sanitizing" transformations that are applied to the data to protect anonymity.

In addition to the challenges involved in synthesizing specifications for new platforms, the matter of obtaining the necessary information to drive analysis and perform behavioral detection on these platforms presents new challenges as well. While binaries compiled for certain mobile platforms, such as Android and Windows Phone, are somewhat easier to work with than the x86 binaries that malware analysts have traditionally struggled with, it is not clear what type of software event will prove most fruitful as a basis for constructing behavior graphs. Because Android is Linux-based, system calls are a promising candidate, but the situation is not as clear on Windows Phone and iOS. Furthermore, any detector will need root access to the phone to perform real-time behavior analysis, and most users will not be willing to void their carrier contract by enabling this. Finally, the energy and performance constraints on these devices sharpens the problem posed by the complexity of the behavior matching problem, as described previously. Before behavior-based detection can be brought to bear on these platforms, each of these problems will need to be addressed.

V. 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 [20], and Rieck *et al.* proposed a classification technique that uses support vector machines to produce class labels [16] 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.

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.* [12] and Christodorescu *et al.* [15], use notions of software behavior that correspond very closely to our own, 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 discuss, 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 [12]. 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.* [21] 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 [22] 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 [23], and Panorama uses whole-system taint analysis in a similar vein to detect more general classes of spyware [14].

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.* [24], 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.* [25] and Sankaranarayanan *et al.* [26] 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.* [27] 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 III).

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 [28]. It has since been applied to a number of problems in computer science, in particular software engineering. Ganapathy *et al.* [29] 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 [30], [31] and program understanding [32]. 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.

VI. 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 [13].

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] "Kaspersky security bulletin 2012," http://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012.

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

[3] 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.

[4] 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.

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

[6] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Oakland*, 2010.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Fredrikson, M. Christodorescu, and S. Jha, "Dynamic behavior matching: A complexity analysis and new approximation algorithms," in *Automated Deduction CADE-23*, 2011.

[19] A. Khan, X. Yan, and K.-L. Wu, "Towards proximity pattern mining in large graphs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. New York, NY, USA: ACM Press, 2010, pp. 867–878.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] S. Sankaranarayanan, F. Ivančı, and A. Gupta, "Mining li-

brary 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] ——, "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.