Agent-Based Trace Learning in a Recommendation-Verification System for Cybersecurity

William Casey,* Evan Wright,* Jose Andre Morales,* Michael Appel,* Jeff Gennari* and Bud Mishra†
*Software Engineering Institute, Carnegie Mellon University
†Courant Institute, New York University

***Abstract*—Agents in a social-technological network can be thought of as strategically interacting with each other by continually observing their own local or hyperlocal information and communicating suitable signals to the receivers who can take appropriate actions. Such interactions have been modeled as information-asymmetric signaling games and studied in our earlier work to understand the role of deception, which often results in general loss of cybersecurity. While there have been attempts to model and check such a body of agents for various global properties and hyperproperties, it has become clear that various theoretical obstacles against this approach are unsurmountable. We instead advocate an approach to dynamically check various liveness and safety hyperproperties with the help of recommenders and verifiers; we focus on empirical studies of the resulting signaling games to understand their equilibria and stability. Agents in such a proposed system may mutate, publish, and recommend strategies and verify properties, for instance, by using statistical inference, machine learning, and model checking with models derived from the past behavior of the system. For the sake of concreteness, we focus on a well-studied problem of detecting a malicious code family using statistical learning on trace features and show how such a machine learner—in this study a classifier for *Zeus/Zbot*—can be rendered as a property, and then be deployed on endpoint devices with trace monitors. The results of this paper, in combination with our earlier work, indicate the feasibility and way forward for a recommendation-verification system to achieve a novel defense mechanism in a social-technological network in the era of ubiquitous computing.**

I. Introduction

Current trends in technology point to increasing ubiquity of social-network and application-centric frameworks. While these trends have dramatic security implications that highlight the need to detect deceptive behaviors, they also underscore the importance of developing new methods for malware detection and deterrence.

The problem facing the agents of a social-technological network is to identify and classify the various forms of deception and attack in traces (e.g. sequence of observed events) executed on endpoint devices. Just as attackers may employ deception to achieve an attack (for example, a benign-sounding flashlight app actually opens a back door to surveil the endpoint device's GPS coordinates as reported in [1]), a defensive user may also check and validate that an app abides by a specific system security property such as *non-surveillance*, which could be validated on the endpoint device using a trace monitor. The transactions in social-technological networks embody many such repeated games with payoffs and costs. In the example of the flashlight app, the sender of the app receives the benefit of asymmetric information relative to each receiving agent (i.e., each endpoint device which installed the flashlight app). Each receiver incurs the costs of the loss of privacy and unawareness of information asymmetry, profitably exploited by the sender.

We model these interactions in a social-technological network as repeated signaling games. To better understand the possible dynamics, our team has developed a simulation system that tests the dynamic behavior of a population with access to a recommendation-verification system that can suggest and score defense options to deter ongoing attacks. Defense options may include malware detectors, system property monitors, validation exams for software, hardware/software trust certificates, etc. The system provides equities for the development of effective defenses to ongoing attacks and when defense options are transferable they allow populations to adapt to novel threats. Furthermore, the currency of such a system is M-Coin certificates, which provide proofs concerning the integrity of options or even properties of app behavior [2].

In this paper we consider a full implementation of a recommendation-verification system. Our goal here is to show how machine (statistical) learning of trace features particular to a malware family could provide a basic defense option specified as system properties to be implemented on endpoint devices. The requirements of a recommendation-verification system offering defensive options to users are 1) options can be measured for effectiveness, 2) options may be widely transitioned across a large number of endpoint devices via user recommendation or otherwise, 3) options are mutable by user agents, and 4) user agents may hold a portfolio of mixed strategic options. To achieve mutability by user agents—an important requirement for adapting to attack evolution —defensive options must be human interpretable and have low complexity so that user agents can manipulate options in meaningful and strategic ways. In order for users to employ a mixed strategy, the strategic options must be defined within the context of an algebra; we suggest properties or hyperproperties. Note further, a honey-net could supplant the users in exploring strategies without incurring a high cost.

By reconsidering the well known problem of machine learning traces in the context of a recommendation-verification system, we 1) create a practical use of properties and hyperproperties that can be implemented on endpoint devices via a trace monitor and 2) demonstrate the feasibility of the recommendation-verification system by meeting the requirements for defense options.

The feasibility of the recommendation-verification system opens the way to new defense mechanisms that are scalable to populations of users in a social-technological network in the era of ubiquitous computing. This paper is organized as follows: Section II is background, Section III is key definitions, Sections IV and V present on API scraping and trace learning, Section VI discusses results, and Section VII contains conclusions. This extended abstract focuses on the empirical studies, and relegates the theoretical analysis to the full paper.

II. Background

Signaling games have been widely used in biology and economics to understand strategic interactions between two agents, one informed and the other uninformed via a set of signals. Behavior modeling of agent-based populations in cybersocial systems via signaling games was introduced in [2] and was later extended to minority games with epistatic signaling. Both simulation studies are used to understand how a recommendation-verification system may operate practically. In signaling games the parameters of costs/payoffs were shown to have dramatic outcomes on expected system (of population of agents) behavior. Epistatic signaling games—where defense options consider a vast attack surface —provide more realistic simulations yet retain many of the dynamics discovered in signaling games. The system-wide effects of an early-adapter advantage were explored in the minority game variation; that work allows us to explore the effects of preferentially rewarding early challenging receivers who adapt effective defenses in response to an ongoing attack technique—an important condition for any system that provides incentives for challenges to adapt (via mutation or other means) to novel attacks. Further exploration investigated the use of strong and transparent metrics for scoring security challenges (e.g., properties) and how such metrics may lead to more effective population-wide responses to emerging attacks. While the simulation studies address population behavior and dynamics, the question of how to implement such a system remained open. In this paper we seek to demonstrate how challenge options for a recommendation-verification system could be realized with a methodology that learns the properties of traces from a particular malicious code family *Zeus/Zbot* (henceforward referred to as *Zeus*). Later in the discussion we revisit the aspect of measuring effectiveness, measured intrinsically in the results section, which in a recommendation-verification system for a population should be an extrinsic (albeit heuristic) measure in terms of a score for challenge strategy.

Formal methods including model checking and properties (as sets of traces) and hyperproperties (as sets of properties) are referenced as ways to address the growing problem of malware and cybersecurity in today's ecology of computing (ref: Science of Cyber Security report titled 'JASON' [3]). Hyperproperties are also suggested as a potential means to formally describe attack behavior and malicious use cases. Hyperproperties are shown [4] to compactly describe security properties such as *non-interference*, where a guarantee that a program complies with an access control model is succinctly described as a hyperproperty. We are most likely the first to suggest the possibility that properties or hyperproperties could be dynamically checked and used through challenges to attacks in a recommendation-verification system. Such challenge options could be realized on endpoint devices using trace monitors. To facilitate the requirements of recommendation-verification systems, we must describe detectors (i.e., challenge options) in a formal and standard way that is also human interpretable; we suggest hyperproperties as an ideal format.

Many examples demonstrate the use of machine learning in the area of cybersecurity [5]–[8]. For an overview and survey of machine learning in malware analysis, we recommend [9]. In this paper, we focus on machine learning methods that produce interpretable models dating back to the seminal work of Quinlan: [10], [11], which develops algorithms for inducing a simple and interpretable model from structured features; [12] for boosting a classifier by combining an ensemble of weaker learners; and [13] for ensemble boosting for interpretable decision trees.

Our technique illustrated on *Zeus* is related in subject to [7], which confines the learning objectives to features obtained from observations of traditional runtime behavior. We pursue a trace-based approach most similar to [6], whose authors construct an analogous analysis technique but limit the feature space to API count profiles and stop short of measuring critical performance metrics (including model complexity) needed in a distributed recommendation-verification system. We extend the learning objective from the feature space of API count profiles to a feature space that includes primitive features (e.g., k-mers or subsequences of k events) of local ordering (of function call sequences); the outcome suggests that this extension leads to models that are more concise and less complex. We further show how to translate the results to a formal property, which could be deployed in a federated response.

III. Definitions

Let Σ be a set of **states**. A **trace** is any countable sequence over Σ:

$$\sigma_i \in \Sigma \text{ for } i \in S \text{ with } S \subset \mathbb{N}.$$

In sequence notation σ_i denotes the i^{th} state in the trace sequence. Trace data in practice may be thought of as a finite sequence over a set of states. Trace sequences can express a program execution sequence with a wide variety of granularity specified by the states encoded as Σ. In the theoretical context, a trace σ may be either **finite** ($\sigma \in \Sigma^*$) or **countably infinite** ($\sigma \in \Sigma^{\aleph_0}$).

Properties. A property is a set of traces. Properties are further studied in a theoretical context where they may be categorized into types, including **safety properties**, which are properties determined by the occurrence of critical and irredeemable events in finite time, and **liveness properties**, which are properties that for any finite trace may be continued to form a member of the property. Clarkson and Schneider introduced hyperproperties in [4] to derive more compact descriptions of security properties. For hyperproperties they also discuss the distinction between safety and liveness and show a decomposition theorem. The decomposition theorem for properties states that any property can be described as the intersection (conjunction) of a safety property and a liveness property. The decomposition theorem may be proved using set systems over Σ^* as the lower Vietoris topology. During the methodology, properties (as sets of traces) are the learning objective, and we explicitly show methods for computing properties that are indicative of *Zeus* malware.

Developing trace data. In the context of executable behavior the states of Σ may be the observable events associated with kernel functions expressed during runtime. A tracing technique, such as the technique introduced in the next section, may be considered, in the abstract, to be a mapping taking as input a static binary executable b and running the binary in a monitor to produce a string over Σ^*. To denote the abstract mapping between binary executable and trace sequence, we introduce the function Φ, which for every b is capable of producing a value $\Phi(b) \in \Sigma^*$ in sequence space.

Property learning using trace features. To establish a starting position trace features may include counts of epochs (e.g., $k{-}mer$ substrings) occurring in trace and their statistics. We define a problem focused on learning a property (as a set of traces) from trace features. In this case we consider learning a property by observing trace objects for a specific malware family using supervised knowledge—meaning that at least part of the answer (of what constitutes the malware family) is known *a priori* from other methods/means. The result is a classifier, which should more compactly represent the property (or set of traces) than the set itself.

To be more precise, given a sample of traces from T_0 (constituting the behavior of a malware family), we consider the problem of learning a classifier for T_0 as a *property* from a background sample of traces from other malware families $T_1, T_2, \ldots, T_K$. The result is a classifier that takes as input a trace and determines if the trace is within the property T_0.

IV. Methodology: API scraping

Design and implementation. *API scraping* is a method for collecting trace data from an executable program. Trace data may be analyzed to gain an understanding of how a program behaves in a system. In general, abstract programming interface (API) scraping is the art of specifying and capturing the data necessary to gain understanding of behavior.

To implement API scraping we use binary instrumentation. Binary instrumentation is the process of instrumenting a running program with monitoring code to record or augment runtime events, such as function calls or data flow through a given function. Binary instrumentation may be designed to record massive amounts of data at the finest resolution of processor actions; however, due to the inherent tradeoff between resolution and the resulting signal-to-noise ratio in recorded output, it is most often designed to capture events that provide the most relevant information for understanding behavior. Therefore, rather than capture all data related to every possible action in the process, we design a technique for selectively and strategically instrumenting functions (often specified in an API, thus the name *API scraping*).

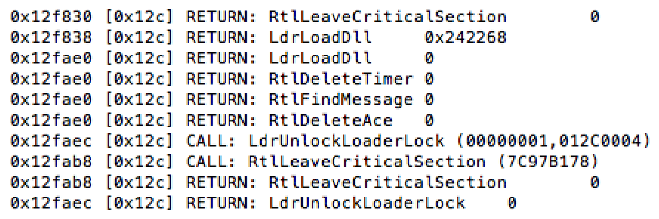
This technique allows an arbitrary list of kernel or otherwise accessible functions to be specified for instrumentation. In general, any function whose prototype can be discovered and loaded within an executable image can be listed for scraping by our technique. The results of allowing the strategic selection of which functions to trace via *API scraping* are higher signal-to-noise ratios in recovered traces (for example, we often monitor the functions called directly from the executable module and discard nested or supporting API calls); more complete coverage of binary behavior fluxing through a given API; and increased abilities to specialize tracing needs to specific threats. Complete coverage of the kernel space is important when studying malware because even a determined adversary is limited to the use of kernel function (however obscure) for accomplishing goals and tasks.

In our technique, we focus on capturing events that relate to system interaction by recording the events of functions (*entry* and *exit*) made by an instrumented program. Our implementation utilizes the Intel Pin binary instrumentation tool [14] for dynamic instrumentation and includes a programmable pattern-matching language. We selected Intel Pin because it provides the functionality needed to instrument an executable file at the function call level, is relatively stable, and supports multiple platforms with a unified process model.

Tailoring API scraping for behavioral sequences. Central to the design of our *API scraping* technique is that the validating agent selects which functions to scrape. We denote the selected list of function as $F = \{y_1, y_2, \ldots, y_M\}$, with each y_i a specific function (perhaps from a system API listing). For each specified function, a monitor is established in the process image that produces a report for the *entry* event (occurring when the function is called) and the *exit* event (occurring when the executing process returns control from the function to the caller).

Our API scraping implementation for Windows XP, 7, and 8 operating systems is shown in Figure 1. It utilizes Intel Pin and reports a sequence of kernel function API calls with the address of parameters and return values. We instrument a list of 527 Windows kernel-level functions, therefore $|F| = 527$. Of the instrumented functions, 283 are `Rtl` functions, 224 are `NT` functions, and 17 are `Ldr` functions.

Last, for a given listing of functions F implemented in API scraping, we denote the binary instrumentation as a function: $\Phi_F(b) \in \Sigma^*$, which takes as input a binary executable b and produces a trace sequence Σ^*. In this setting, the elements of Σ are limited to function and the action type (*entry* or *exit*);



```
0x12f830 [0x12c] RETURN: RtlLeaveCriticalSection        0
0x12f838 [0x12c] RETURN: LdrLoadDll     0x242268
0x12fae0 [0x12c] RETURN: LdrLoadDll     0
0x12fae0 [0x12c] RETURN: RtlDeleteTimer 0
0x12fae0 [0x12c] RETURN: RtlFindMessage 0
0x12fae0 [0x12c] RETURN: RtlDeleteAce   0
0x12faec [0x12c] CALL: LdrUnlockLoaderLock (00000001,012C0004)
0x12fab8 [0x12c] CALL: RtlLeaveCriticalSection (7C97B178)
0x12fab8 [0x12c] RETURN: RtlLeaveCriticalSection        0
0x12faec [0x12c] RETURN: LdrUnlockLoaderLock    0
```

Figure 1. Trace implementation using Intel Pin binary instrumentation tool. Each line reports a function event such as CALL (*entry*) and RETURN (*exit*) from the functions in F of our API-scraped list. Each action record records execution pointer, thread ID, and parameter/return addresses from left to right.

for now the other recorded fields are ignored.

Deception, polymorphism, and *Zeus* traces. In the context of a social-technological network it is important to recognize that implementing polymorphic techniques (to be deceptive) comes at a fixed cost to the malware designer; such techniques dramatically increase the difficulty and cost of agent-based checking (e.g., detection, classification, or signature generation). For this reason and to demonstrate the difficulties that a polymorphic attack family can present we consider the *Zeus* botnet crimeware [15]–[17], which is designed to go undetected via polymorphism, establish system persistence, and infect web browsers to achieve a man-in-the-app attack that allows the operator to intercept or counterfeit web traffic, usually to electronic banking sites. Using *Zeus* we are also able to demonstrate the distinction between the efforts to develop static signatures as opposed to considering behavior sequence from trace data.

Because the *Zeus* family is a sophisticated threat employing both deception (obfuscation) and anti-analysis techniques, it is an ideal test subject for the methodology of learning properties (for a difficult and polymorphic case) and for showing how the result may be used in a recommendation-verification system by codifying the detection as a property. The *Zeus* malware family is known to be polymorphic (employing several layers of obfuscation) and implements anti-debugging techniques [15] [18]. Polymorphic techniques employed by malware families (such as *Zeus*) increase the difficulty of static analysis by obfuscating or degrading the signal that individual elements of the family are in fact related, at least from the perspective of their executable images stored on disk.

However the *Zeus* malware family is not considered to be metamorphic, so analysis of behavior sequences in trace data should in principle lead to the discovery of invariants across all the elements of the family.

To show that the *Zeus* malware samples are polymorphic we apply a clone mapping study to a control set of binary executable images produced using the *Zeus* builder version 1.2.4.2; this study highlights the levels of code obfuscation as shown in Figure 2. Using the same control set of binary images we provide a visualization of API scraping by plotting the execution cursor over trace sequences in Figure 3; this visualization demonstrates the similarities in trace and suggests that patterns in trace may be simpler to analyze and summarize for *Zeus* and possibly other cases of polymorphism.

Our results section shows that the behavioral sequences admit compact descriptions as properties, which are machine learnable and simple to state because of the relatively low complexity of the resulting classifiers. We provide a discussion of the validity of our API scraping methodology by showing critical actions of the *Zeus* binary process as recorded by the API scraping method despite the documented anti-debugging techniques employed by the bots.

To obtain Figures 2 and 3, we generate a control data set by starting with the *Zeus* 1.2.4.2 builder and then feed a single (constant) configuration script into the builder to create multiple bots.

Figure 2 illustrates how these static images vary greatly. The technique [19] maps all code-clones (or common byte sequences) found in a set of generated *Zeus* bots and illustrates that there is very little code similarity to discover. The only interesting findings from the code-clone analysis are that a small sequence associated with the outermost unpacking function —which is designed to look like a UPX style decoder—is placed at a random position in the "text" section of the binary, and random bytes are sprayed into the sequence to prevent long matches of byte sequences greater than around 30 bytes. Clearly the builder employs a technique that randomizes values as well as location for the entry function to increase the difficulty and cost of creating signatures for the static images. The general problem of detecting the *Zeus* family may be made even more difficult because there are multiple versions of *Zeus* beyond version 1.2.4.2.

In Figure 3 we illustrate the execution pointer position as a function of time for four zbot products (configuration script held constant) to discover that there is observable similarity in the trace functions. Associated with each $y-$position is the execution pointer position plotted as a function of time ($x-$axis); the execution position (for loaded kernel modules) is correlated to linked functions expressed during runtime. Even though not all traces record the same number of steps (or time interval), the similarity may be viewed with dilation and truncation options. In addition, in Figure 3 we annotate the sequence with coloration on the $x-$axis of each trace function. These coloration sequences help to calibrate one trace against another to see the similarities in trace.

Validity of API scraping methodology. To determine whether the instrumentation techniques are non-interfering with the malicious sample (despite the anti-debugging techniques implemented), we test whether the *Zeus* binaries are able to achieve known characteristics of runtime as outlined in [15]–[17]. Specifically we check the characteristic that the *Zeus* binaries identify a persistent running process—in each case *winlogin* —and attach a secondary infection into initialization routines. For version 1.2.4.2 this level of system compromise is validated by checking for value `C:/WINDOWS/system32/sdra64.exe` appended to the key field `UserInit` for key `Winlogon`. Each bot traced achieves these steps, indicating that the technique of API scraping is robust to the anti-debugging features of the

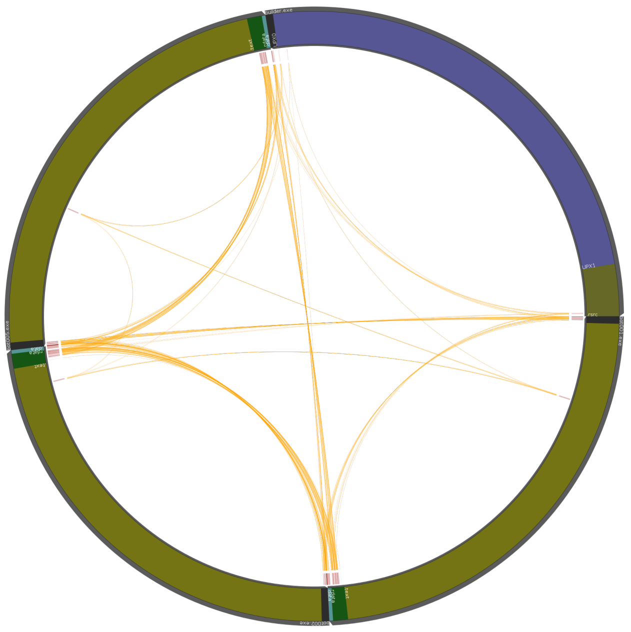


Figure 2. High levels of Code Polymorphism and Obfuscation: The image above displays the code-clone map for static binary images for the *Zeus* 1.2.4.2 builder and three *Zeus* products generated using an identical configuration setting. The static binary images are displayed in sequence about the annular ring. The image reveals a high level of obfuscation. The builder (purple band) starts at the 12:00 position and extends to the 3:00 position. Bots appear in clockwise order: product 1, product 2, and product 3. The orange counter arcs through the interior of the image represent all code clones or string matches found in more than one binary, exceeding 12 bytes, and having boosted Shannon entropy.

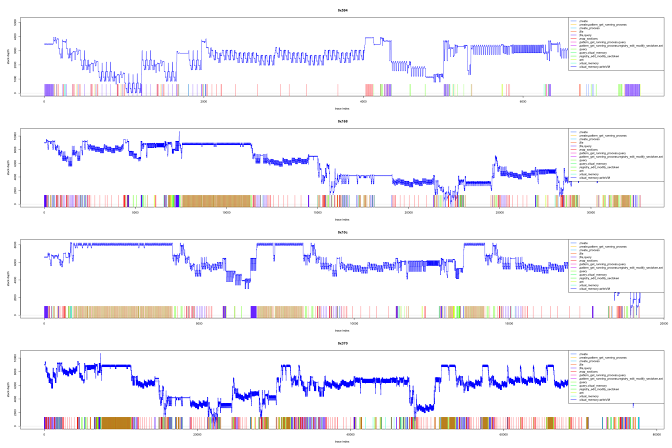


Figure 3. Traces clarify the commonality for *Zeus* products. Here we plot the execution pointer position (as a function of time on the x-axis) for four bot images (instrumented). Each image is allotted a runtime of 30 seconds, during which kernel API functions are hooked and API sets are grouped and annotated as colored regions on the x-axis, revealing visual similarity for process execution across all of the products.

malware family.

V. Methodology: Learning Properties

With the API scraping technique established, this section focuses on the overall problem of learning characteristics of trace in a given family of malware. We start by outlining the overall process to develop a classifier for a given malware family.

Given: T_0 A property (set of traces).
Process:

- Baseline: Develop a stratified sampling over comparable objects (i.e., other properties as sets of traces from malware families). The properties whose union is termed *baseline* are denoted as $T_1 \cup T_2 \cup \ldots \cup T_K$.
- Compute: Learn a classifier for property T_0 vs. baseline in terms of the sequential features of trace.

Output: A classifier specified in terms of a property's prescribing trace features of target family and proscribing features of baseline.

Given a computed classifier for T_0, the intrinsic validation measures of the binary classifier include the following:

- Accuracy (holdout) using 10-fold cross validation.
- Statistical power measures. True positive vs. false positive, precision and recall.
- Model complexity or description length of classifier.
- Model interpretability.

Of the intrinsic validation measures, we are particularly interested in complexity and interpretability, which make practical the use of a recommendation-verification system.

Data set. We considered an example set of 1,933 unique *Zeus* binaries labeled by the *contagio* website. To represent all other non-*Zeus* malware, we used a baseline set of 9,620 malware samples chosen from a large corpus of manually labeled PE32 malware; this corpus included 151 (i.e., $K = 151$) distinct malware families other than *Zeus*. For each baseline family, we used a stratified sample and bound the maximum number from each family to control for large families in our baseline dataset. Stratified sampling is a technique where the number of representatives of the kth family T_k is proportional to $\frac{|T_k|}{\sum_{i=0}^{K} |T_i|}$ and therefore is proportional to an underlying frequency estimated as the frequency of observed tagged artifacts within a corpus. For now, we assume the frequency of tags in our corpus of collected artifacts corresponds to underlying frequencies of artifact prevalence in the wild.

Statistical learning. With T_0 set to the traces derived from the *Zeus* (tagged artifacts from *contagio* [20]) and a background $T_1 \cup \ldots T_{151}$ established, we return to the problem of learning the distinguishing features of trace for T_0 vs. background. Each trace $t_x = \Phi_F(x)$, obtained by applying *API scraping* with selected function set F to the binary executable image x, is an ordered behavior sequence. With a given trace t held constant, we derive three count vectors as features which we group as follows:

1) Function expression profile: Total count—for each function y let f_y^+ count the total number of events (entry or exit events) for a given trace t. Balanced count—for each function

y, let f_y^- be the difference (subtraction) of events (as number of exit events minus entry events) for function y for a given trace t.[1].

2) Function transition profile for $k-$mers: Total count—for each contiguous subsequence of function events of length k as $(y_1, y_2, \ldots, y_k)$, let $\pi_{(y_1,y_2,\ldots,y_k)}$ count the number of occurrences of this subsequence of events (regardless of whether the event is either *entry* or *exit*) for a given trace t.

Although in our experiments we limit $k-$mers size to $k = 2$ the technique easily extends to any number of transitional $k-$mers.

For each binary x we obtain the following count feature vectors from its trace t_x:

$$\begin{aligned} \underline{f}^+(x) &= \langle f_y^+ \rangle_{y \in F} \\ \underline{f}^-(x) &= \langle f_y^- \rangle_{y \in F} \\ \underline{\pi}(x) &= \langle \pi_{(y,z)} \rangle_{(y,z) \in F \times F} \end{aligned}$$

Therefore for a corpus of binaries $\langle x_i \rangle_{i=1}^N$ we are able to derive observed feature vectors:

$$\langle \underline{f}^+(x_i), \underline{f}^-(x_i), \underline{\pi}(x_i) \rangle_{i=1}^N$$

With these vectors (for the entire data set) in combination with family tagging (e.g., $t_{x_i} \in T_j$, which indicates the supervised knowledge that binary i is contained in family j), we can consider the supervised trace property-learning problem with stratified sampling. We outlined the method to derive intrinsic measures for the supervised trace property-learning problem as an experiment:

Experiment

- Training:
  - Stratified Sample: For each k select a sub-sample of size 90% from T_k.
  - Learn distinguishing trace features.
- Test and Evaluate: Using the resulting model, test on all data.

In the experiments we explore the use of these learning methods as implemented in the Weka framework [21]:

- Naive Bayes: A model that assumes independence of features; we expect it to perform poorly but include it to provide a point of reference.
- C4.5: A general and interpretable model without assumptions of feature independence.
- Random Forest: One of the most widely deployed methods; uses boosting.
- C4.5 with Adaboost: An ensemble enhancement to the existing C4.5 to demonstrate the effect of Adaboost, a popular ensemble technique.

[1]Generally benign code should balance function calls (with few exceptions), however in the case of malware, non-zero-balanced counts may be a particularly interesting feature resulting from exploits.

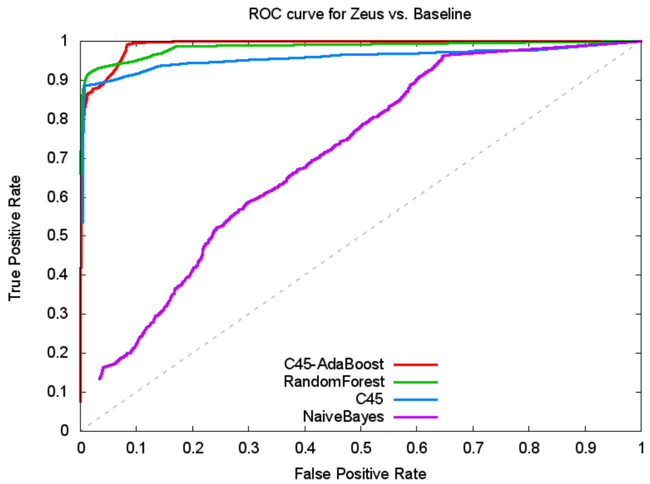
All methods selected above produce interpretable outputs, in contrast to methods such as neural nets, which is one element of the motivating criteria. For each method we consider the intrinsic validation measures below.

VI. Results

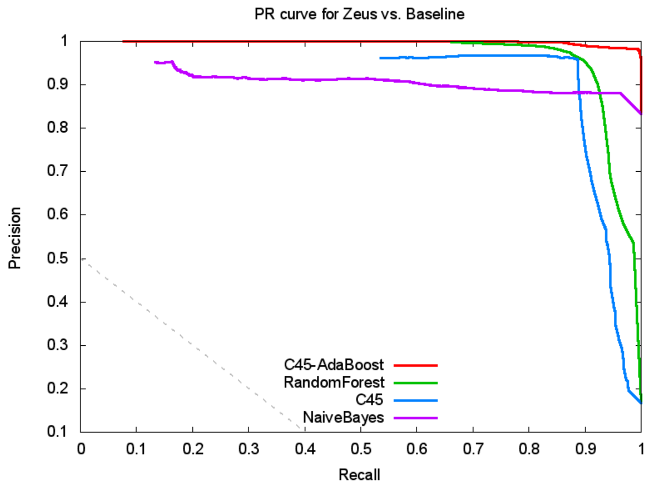
Our experiments indicate the ability of machine learning methods to accurately and compactly identify the similarity in traces even for highly polymorphic malware such as *Zeus*. While the accuracy we attain is 97.95% and therefore comparable with other previous results, we wish to emphasize how other desired properties of a learning method are accomplished with particular attention to model complexity. We present several images of the classification features (with tagging), provide the intrinsic validation metrics for each classifier, and indicate how each malware classifier can be used to create properties for the assignment. Following the results we discuss what implications they have for a recommendation-verification system in a social-technological network.

Sparse feature space and reduction. The computed feature space for the classification problem is rather large, containing counts over the sets $F \times F \times (F \times F)$. Because our API scraping tools targets a total of $|F| = 527$ functions, this amounts to a feature space as large as $|F|^4 = 77,133,397,441$. All methods investigated naturally discount any non-positive counts; therefore, a natural sparsity in the feature space exists and is exploited by these methods. In addition to the sparse occurrence of features, all methods are sensitive to selection of the most distinguishing features and thus select the more distinguishing features with higher priority.

Accuracy. This result was measured as average accuracy in 10-fold cross-validation testing. C4.5 with Adaboost is just above 97.95% improving slightly on the performance of C4.5 with no ensemble methods (97.4%), while the accuracy of Random Forest is similar (97.5%). Our average accuracy measures in the cross-validation testing may not demonstrate the tradeoff of false positives vs. true positives, which is an important consideration in detection systems where there is often a high cost to false positives (e.g., the abandonment of a detection system). To address this shortcoming, we present the receiver operating characteristic (ROC) below as Figure 4(a), which includes Naive Bayes as a comparison point for the methods of C4.5, C4.5 with Adaboost, and Random Forest. The comparison (to Naive Bayes) suggests how these classifiers perform in contrast to a relatively simple method that makes invalid assumptions regarding the independence of features. As we can see in Figure 4(a) the Naive Bayes performance is the poorest performer, doing slightly better than random guessing. With the power measure of receiver operator characteristic (ROC), we demonstrate false positive vs. true positive tradeoffs; however, these tradeoffs may fail to account for large imbalances in class size for which precision and recall may offer alternative views. Because our data corpus contained a baseline dataset approximately four times the size of the *Zeus* dataset, our class sizes are highly varied.



(a) ROC



(b) PR

Figure 4. Statistical power: (a) receiver operating characteristic for various supervised binary classifiers for trace features (b) precision and recall characteristics for various supervised binary classifiers for trace features

To address this concern we present the additional statistical power measures of *precision* and *recall* in Figure 4(b); these measures are more robust to the effects of variable class size.

Model complexity. For trace-based classifiers, complexity addresses our specific interest in obtaining compact descriptions for malware sets as properties. Low-complexity descriptions of properties (themselves fairly large objects) address the requirement that a property be mutable and interpretable so that it may be adapted to evolution in attack method. Implementing trace detection may be done with a wide variety of techniques, but because we are most interested in models that can be interpreted and modified by agents of a social-technological network to create adapted defense options, we are interested in lower-complexity models: the lower the complexity, the more admissible the model is to direct reasoning by an agent or an agent team aligned to address an attack threat in the wild. To explore this aspect of the resulting classifier, we create a measure of complexity as the number of decisions in the resulting decision tree and experiment with how the accuracy may depend on iterative refinement, which decreases the model complexity.

Complexity measures are also of theoretical interest and actually motivated the definition of hyperproperties as a means to more compactly describe security properties. From a practical viewpoint, model complexity is a concern typically associated with model generality and helps avoid model over-fitting. For us, another very practical outcome is that a low-complexity model provides a simple structure to summarize a large object (large set of trace objects, themselves long sequences of behavior action). Consider again the trace sequences viewable in Figure 3, for which our visual capacities perceive structure (at least more so than the clone study in Figure 2, which reveals highly polymorphic images). The complexity measure summarizes how efficiently the model can capture this similarity in structures in the trace sequences. Below in Figure 5 we illustrate an outcome of one of our experiments; the model complexity correlates directly with the size (number of edges and number of nodes) of the decision tree.

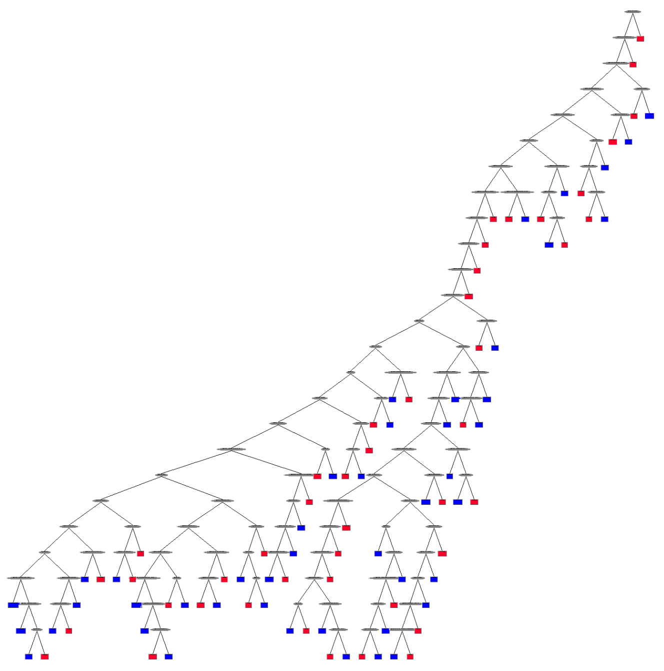


Figure 5. Example trace classifier (computed for the entire data set): Red nodes are features prescribed by property T_0, and blue nodes are features proscribed or alternatively prescribed by the background set.

Low complexity and robustness. Through iterative parameter refinement, the model in our first iteration (shown in Figure 5) can be reduced in complexity by decreasing the targeted confidence parameter in C4.5. By decreasing the confidence parameter, we can induce more concise models. We explore the effect on accuracy in Table I. This procedure explores this notion of robustness and suggests that very simple models (induced by decreasing the confidence parameter) can retain many of the features needed to accurately distinguish *Zeus*' behavior from that of the baseline. Specifically Table I shows that while more concise models can be induced by pruning the resulting decision tree of C4.5 (no boosting) to create a sequence of models with decreasing complexity (quantified as number of leaves/branches), much of the accuracy measured with 10-fold cross validation can be maintained.

iteration	leaves	branches	accuracy
1	91	181	97.44%
2	74	147	97.16%
3	55	109	97.03%
4	43	85	96.48%

Table I
ROBUSTNESS OF INDUCED MODELS WITH LOWER COMPLEXITY

Features selected. We consider the contribution of each feature class to the overall result presented above in Figure 5. A direct count of how many features are from $\underline{f}^+$, $\underline{f}^-$, and $\underline{\pi}$ in the tree at iteration 1 (pre-pruned tree) establishes the counts to be $145, 166$, and 658 respectively. These counts show that 67.905% of the features selected by the model are from the transition class $\underline{\pi}$, indicating that the learning method may find distinct strengths to leveraging information associated with the transitional $k-$mers even for $k = 2$.

Computing a property from a classifier. Below we illustrate how the results of property learning may themselves be used to compute a system security property that could be checked on an endpoint device using a trace monitor. This step completes the overall objective of creating a defense option that meets the requirements of a recommendation-verification system.

Below we present the formal equations that are to be proscribed from an observing trace monitor on an endpoint device as $\mathcal{C}_{zeus}$ in Figure 6(a). Figure 6(b) shows other malware families as $\mathcal{C}_{baseline}$. Figure 6(c) and 6(d) provide a partial view of definitions for variables v_i for $i \in \{1, 2, \ldots 180\}$ defined as a system of 180 variables.

Discussion. Our experiments show how machine learning can be directly linked to the formal and theoretical framework of *properties* that may be transferred among agents in a social-technological network. Our machine learning results of accuracy are consistent with other studies, but we further explore model complexity as a first-order consideration; we speculate that the addition of primitive subsequence $k-$mers as potential features contributes greatly to the results with low complexity and robustness even when inducing simple models.

Each of these findings is important for the overall goal of designing a recommendation-verification system. Consider this single example of a *Zeus* detector created with machine learning of trace sequences, published to a recommendation-verification system as a property for endpoint deployment along with intrinsic measures of accuracy, statistical power, and complexity. This property can be transmitted as a defense option to any agent in a social-technological network. Any agent can therefore adapt this strategic option with an M-coin incentive, and the option can then be evaluated for its *extrinsic*

$$\begin{aligned}\mathcal{C}_{zeus} =& v_{180} \cup v_{179} \cup v_{177} \cup v_{174} \cup v_{170} \cup \\ & v_{168} \cup v_{164} \cup v_{161} \cup v_{157} \cup v_{155} \cup \\ & v_{154} \cup v_{153} \cup v_{152} \cup v_{150} \cup v_{146} \cup \\ & v_{141} \cup v_{138} \cup v_{136} \cup v_{133} \cup v_{130} \cup \\ & v_{129} \cup v_{120} \cup v_{118} \cup v_{110} \cup v_{109} \cup \\ & v_{108} \cup v_{106} \cup v_{102} \cup v_{89} \cup v_{86} \cup \\ & v_{82} \cup v_{78} \cup v_{75} \cup v_{72} \cup v_{66} \cup \\ & v_{64} \cup v_{59} \cup v_{58} \cup v_{53} \cup v_{50} \cup \\ & v_{41} \cup v_{40} \cup v_{36} \cup v_{32} \cup v_{28}\end{aligned}$$

(a) zeus

$$\begin{aligned}\mathcal{C}_{baseline} =& v_{178} \cup v_{175} \cup v_{172} \cup v_{171} \cup v_{165} \cup \\ & v_{163} \cup v_{158} \cup v_{151} \cup v_{148} \cup v_{147} \\ & \cup v_{143} \cup v_{142} \cup v_{140} \cup v_{135} \cup v_{132} \cup \\ & v_{131} \cup v_{128} \cup v_{122} \cup v_{121} \cup v_{119} \cup \\ & v_{113} \cup v_{107} \cup v_{104} \cup v_{101} \cup v_{88} \cup \\ & v_{85} \cup v_{83} \cup v_{81} \cup v_{77} \cup v_{74} \cup \\ & v_{73} \cup v_{71} \cup v_{65} \cup v_{62} \cup v_{57} \cup \\ & v_{54} \cup v_{51} \cup v_{48} \cup v_{46} \cup v_{39} \cup \\ & v_{35} \cup v_{33} \cup v_{31} \cup v_{27} \cup v_{25} \cup v_{23}\end{aligned}$$

(b) baseline

$$\begin{aligned}v_1 =& (\pi_{NtFreeVirtualMemory,RtlInitString} \le 0) \\ v_2 =& v_1 \cap (\pi_{NtSetInformationThread,NtQueryVirtualMemory} \le 0) \\ v_3 =& v_2 \cap (\pi_{RtlLeaveCriticalSection,RtlOemStringToUnicodeString} \le 0) \\ v_4 =& v_3 \cap (\pi_{RtlInitUnicodeString,RtlQueryRegistryValues} \le 0) \\ v_5 =& v_4 \cap (\pi_{NtSetInformationThread,RtlGetNtProductType} \le 0) \\ v_6 =& v_5 \cap (\pi_{NtAllocateVirtualMemory,RtlInitString} \le 0) \\ v_7 =& v_6 \cap (\pi_{NtSetInformationThread,NtProtectVirtualMemory} \le 0) \\ v_8 =& v_7 \cap (\pi_{NtSetInformationThread,RtlUnicodeToMultiByteSize} \le 0)\end{aligned}$$

(c) variable definitions $v_1, \ldots, v_{18}$

$$\begin{aligned}v_{173} =& v_3 \cap (\pi_{RtlInitUnicodeString,RtlQueryRegistryValues} > 0) \\ v_{174} =& v_{173} \cap (\pi_{RtlCompactHeap,RtlFindMessage} \le 6) \\ v_{175} =& v_{173} \cap (\pi_{RtlCompactHeap,RtlFindMessage} > 6) \\ v_{176} =& v_2 \cap (\pi_{RtlLeaveCriticalSection,RtlOemStringToUnicodeString} > 0) \\ v_{177} =& v_{176} \cap (f^+_{NtQueryInformationJobObject} \le 0) \\ v_{178} =& v_{176} \cap (f^+_{NtQueryInformationJobObject} > 0) \\ v_{179} =& v_1 \cap (\pi_{NtSetInformationThread,NtQueryVirtualMemory} > 0) \\ v_{180} =& (\pi_{NtFreeVirtualMemory,RtlInitString} > 0)\end{aligned}$$

(d) variable definitions $v_{173}, \ldots, v_{180}$

Figure 6. Formal properties learned: (a) Equation formalizing learned *Zeus* property (b) Equation formalizing learned *baseline* property (c) Variables setting for variables $v_1, \ldots v_{18}$ (d) Variable setting for variables $v_{173} \ldots v_{180}$. Definitions are normalized and found in the form of either the terminal case of i) an inequality involving a single feature vector, or the non-terminal case of ii) a conjunction of a previously defined variable with an inequality involving a single feature vector.

effectiveness by a consensus or perhaps by comparison to other competing properties. Receiver agents can provide reputation benefit to the agents creating the most effective security properties (defense options) and its distribution evolving in proportion to the extrinsic effectiveness. Because the option is interpretable and exchanged as a property, it may be mutated or recombined by cross-over with other successful strategies to create novel options that may prove more effective than the source options. From the defense perspective, options can be subsampled to create ensembles and variations providing mixed strategies, innovation of strategies, trembling hand strategies, population heterogeneity, etc. Though we only briefly mention these defense options in this paper (we discuss them extensively in the full paper), they are now the basis of a practical recommendation-verification system.

VII. CONCLUSIONS AND FUTURE WORK

In an era of ubiquitous computing with agents within social-technological networks, we must adopt new ways of thinking about the problems of deception and attacks. As computing becomes more social and more ubiquitous, more and more agents encounter each other in a context of asymmetric information; this situation naturally creates an incentive to efficiently and effectively detect and react to deception. We foresee a fertile area developing at the intersection of evolutionary games (biology) and science of cybersecurity (technology), for which notions of adaptation and mutational drift/diffusion of strategy play a critical role in a population's ability to manage and mitigate a large attack and vulnerability surface.

We plan to extend this research further by considering API scraping and modeling with properties of the specific behaviors of executables. Currently we are developing an API scraping tool aimed at summarizing behavior in malicious PDFs and JavaScript exploits. We also plan to consider learning and model checking individual malware behaviors as properties.

Acknowledgments: Authors would like to thank Leigh Metcalf (SEI), Jonathan Spring (SEI), Rhiannon Weaver (SEI), and Thomson Nguyen (NYU) for lively and active discussion on the topic of science of cyber security giving rise to new ways to consider the cyber security. Jeff Kramer (Iowa State University) for early investigations of instrumentation code, Aaron Shelmire (Dell Secure Works) for analysis of Zeus unpacking functions. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute.

References

[1] M. Kassner, "Android flashlight app tracks users via gps, ftc says hold on," December December 11, 2013, [Online; posted December 11, 2013, 9:49 PM PST]. [Online]. Available: http://www.techrepublic.com/blog/it-security/why-does-an-android-flashlight-app-need-gps-permission/

[2] W. Casey, J. A. Morales, T. Nguyen, J. Spring, R. Weaver, E. Wright, L. Metcalf, and B. Mishra, "Cyber security via signaling games: Toward a science of cyber security," in *ICDCIT*, ser. Lecture Notes in Computer Science, R. Natarajan, Ed., vol. 8337. Springer, 2014, pp. 34–42.

[3] Mitre, *Science of Cyber-security*. JASON, MITRE Corporation, 2010. [Online]. Available: http://fas.org/irp/agency/dod/jason/cyber.pdf

[4] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[5] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici, "Unknown malcode detection via text categorization and the imbalance problem," in *ISI*. IEEE, 2008, pp. 156–161.

[6] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," in *Proceedings of the 5rd International Conference on Malicious and Unwanted Software : MALWARE 2010*, 2010.

[7] A. Mohaisen and O. Alrawi, "Unveiling zeus: Automated classification of malware samples," in *Proceedings of the 22Nd International Conference on World Wide Web Companion*, ser. WWW '13 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013, pp. 829–832. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487788.2488056

[8] V. Moonsamy, R. Tian, and L. Batten, "Feature reduction to speed up malware classification," in *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, ser. NordSec'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 176–188.

[9] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 2014, 2014.

[10] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986. [Online]. Available: http://dx.doi.org/10.1023/A:1022643204877

[11] ——, *C4. 5: programs for machine learning*. Morgan kaufmann, 1993, vol. 1.

[12] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: http://www-stat.stanford.edu/ tibs/ElemStatLearn/

[13] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[15] H. Nayyar and P. Bueno, "Clash of the Titans: ZeuS v SpyEye," 2010. [Online]. Available: http://www.sans.org/reading-room/whitepapers/malicious/clash-titans-zeus-spyeye-33393

[16] T. Ormerod, L. Wang, M. Debbabi, A. Youssef, H. Binsalleeh, A. Boukhtouta, and P. Sinha, "Defaming botnet toolkits: A bottom-up approach to mitigating the threat," in *Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, ser. SECURWARE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 195–200. [Online]. Available: http://dx.doi.org/10.1109/SECURWARE.2010.39

[17] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. M. Youssef, M. Debbabi, and L. Wang, "On the analysis of the zeus botnet crimeware toolkit," in *PST*, 2010, pp. 31–38.

[18] N. Falliere, "Windows anti-debug reference," *Retrieved October*, vol. 1, p. 2007, 2007.

[19] W. Casey and A. Shelmire, "Signature Limits: An Entire Map of Clone Features and their Discovery in Nearly Linear Time," *ArXiv e-prints*, Jul. 2014.

[20] contagio. (2014) contagio malware dump. [Online]. Available: http://contagiodump.blogspot.com/

[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278