A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection

Salvatore J. Stolfo, Frank Apap, Eleazar Eskin, Katherine Heller, Shlomo Hershkop, Andrew Honig, and Krysta Svore
{sal,fapap,eeskin,heller,shlomo,kmsvore}@cs.columbia.edu

Department of Computer Science, Columbia University, New York NY 10027, USA

Abstract. We present a component anomaly detector for a host-based intrusion detection system (IDS) for Microsoft Windows. The core of the detector is a learning-based anomaly detection algorithm that detects attacks on a host machine by looking for anomalous accesses to the Windows Registry. We present and compare two anomaly detection algorithms for use in our IDS system and evaluate their performance. One algorithm called PAD, for Probabilistic Anomaly Detection, is based upon a probability density estimation while the second uses the Support Vector Machine framework. The key idea behind the detector is to first train a model of normal Registry behavior on a Windows host, even when noise may be present in the training data, and use this model to detect abnormal Registry accesses. At run-time the model is used to check each access to the Registry in real-time to determine whether or not the behavior is abnormal and possibly corresponds to an attack. The system is effective in detecting the actions of malicious software while maintaining a low rate of false alarms. We show that the probabilistic anomaly detection algorithm exhibits better performance in accuracy and in computational complexity over the support vector machine implementation under three different kernel functions.

1 Introduction

Microsoft Windows is one of the most popular operating systems today, and also one of the most often attacked. Malicious software running on the host is often used to perpetrate these attacks. There are two widely deployed first lines of defense against malicious software, virus scanners and security patches. Virus scanners attempt to detect malicious software on the host, and security patches are operating system updates to fix the security holes that malicious software exploits. Both of these methods suffer from the same drawback. They are effective against known attacks but are unable to detect and prevent new types of attacks.

Most virus scanners are signature based meaning they use byte sequences or embedded strings in software to identify certain programs as malicious [16, 31]. If a virus scanner's signature database does not contain a signature for a specific malicious program, the virus scanner can not detect or protect against that program. In general, virus scanners require frequent updating of signature

databases, otherwise the scanners become useless [38]. Similarly, security patches protect systems only when they have been written, distributed and applied to host systems. Until then, systems remain vulnerable and attacks can and do spread widely.

In many environments, frequent updates of virus signatures and security patches are unlikely to occur on a timely basis, causing many systems to remain vulnerable. This leads to the potential of widespread destructive attacks caused by malicious software. Even in environments where updates are more frequent, the systems are vulnerable between the time new malicious software is created and the time that it takes for the software to be discovered, new signatures and patches created by experts, and ultimately distributed to the vulnerable systems. Since malicious software may propagate through network connections or email, it often reaches vulnerable systems long before the updates are available.

A second line of defense is through IDS systems. Host-based IDS systems monitor a host system and attempt to detect an intrusion. In the ideal case, an IDS can detect the effects or behavior of malicious software rather then distinct signatures of that software. Unfortunately, widely used commercial IDS systems are based on signature algorithms. These algorithms match host activity to a database of signatures which correspond to known attacks. This approach, like virus detection algorithms, requires previous knowledge of an attack and is rarely effective on new attacks. Recently however, there has been growing interest in the use of anomaly detection in IDS systems, as first proposed by Denning [18] in a comprehensive IDS framework. Anomaly detection algorithms use models of normal behavior in order to detect deviations which may correspond to attacks [15]. The main advantage of anomaly detection is that it can detect new attacks and effectively defend against new malicious software. Anomaly detection algorithms may be specification-based or data mining or machine learning-based [30, 32], and have been applied to network intrusion detection [27, 29] and also to the analysis of system calls for host based intrusion detection [19, 21, 24, 28, 37].

Specification- or behavior-based anomaly detectors, such as the STAT approach [39], represent normal execution of a system using state transition or fine state machine representations [33]. Anomalies are detected at run-time when the execution of a process or system violates the predefined normal execution model. Data mining or machine learning-based anomaly detectors automatically learn a normal model without human intervention.

In this paper, we examine a new approach to host-based IDS that monitors a program's use of the Windows Registry. We present a system called RAD (Registry Anomaly Detection), which monitors and learns a normal model of the accesses to the Registry in real-time and detects the actions of malicious software at run-time using the learned model.

The Windows Registry is an important part of the Windows operating system and is very heavily used, making it a good source of audit data. By building a sensor on the Registry and applying the information gathered to an anomaly detector, we can detect evidence of the activity of malicious software. The main advantages of monitoring the Windows Registry are that Registry activity is

regular by nature, it can be monitored with low computational overhead, and almost all system activities interact with the Registry.

We present and comparatively evaluate two learning-based anomaly detection algorithms used in this work. The first anomaly detection algorithm, Probabilistic Anomaly Detection (PAD), is a Registry-specific version of PHAD (Packet Header Anomaly Detection), an anomaly detection algorithm originally presented to detect anomalies in TCP packet headers [32]. Mahoney and Chan formally compared performance of the PHAD algorithm to other prior algorithms and was shown to exhibit substantial improvement in accuracy. PAD is a more robust and extensive improvement over PHAD. The second anomaly detection algorithm we present uses a One-Class Support Vector Machine (OCSVM) to detect anomalous activity in the Windows Registry using different kernel functions. The paper will also discuss the modifications of the PHAD algorithm as it is applied in the RAD system, as well as the details of the OCSVM algorithm under three different kernel functions.

Furthermore, we show that the data generated by a Registry sensor is useful in detecting malicious behavior. We also describe how various malicious programs use the Registry, and what data can be gathered from the Registry to detect these malicious activities. We then apply our two anomaly detection algorithms to the data gathered under controlled experiments and evaluate their ability to detect abnormal Registry behavior caused by malicious software. By showing the results of our experiments and detailing how various malicious activities affect the Registry, we show that the Registry is a good source of data for intrusion detection.

We also present a comparison of the two anomaly detection algorithms and show PAD outperforms the OCSVM algorithm due to the use of the estimator developed by Friedman and Singer [22]. This estimator uses a Dirichlet-based hierarchical prior to smooth the distribution and account for the likelihoods of unobserved elements in sparse data sets by adjusting their probability mass based on the number of values seen during training. An understanding of the differences between these two models and the reasons for differences in detection performance is critical to the development of effective anomaly detection systems in the future.

RAD is designed as a component sensor to provide evidence of a security event for use in a host-based IDS. It is not a complete security solution. Our goal is to integrate RAD and correlate its output with other host sensors (such as a file system anomaly detector [9] or a Masquerade detector [40]) to provide broader coverage against many attacks. See [10] for an example of a host-based correlation engine.

In the following section we present background and related research in the area of anomaly detection systems for computer security. Section 3 fully describes the Windows Registry and the audit data available for modeling system operation. Examples of malicious executables and their modus operandi are described to demonstrate how the Registry may be used by malicious software. Section 4 fully develops the PAD and OCSVM algorithms employed in our studies of Reg-

istry data. This is followed in Section 5 by a description of the RAD architecture which serves as a general model for any host-based anomaly detector. A set of experiments are then described with performance results for each algorithm in Section 6. Section 7 discusses our ongoing research on a number of theoretical and practical issues which will fully develop RAD as an effective tool to guard Windows platforms from malicious executables. The paper concludes in Section 8.

2 Related Research

RAD is a host-based intrusion detection system applied to Windows Registry data which detects anomalous program behavior. In general, some event or piece of data may be anomalous because it may be statistically unlikely, or because the rules that specify the grammar of the data or event are not coherent with the current example (or both). This means that there needs to be a well defined set of rules specifying all data or events that should be regarded as normal, not anomalous. This has been the primary approach of nearly all intrusion detection systems; they depend upon a prior specification of representative data or of normal program or protocol execution.

The alternative is a statistical or machine learning approach where normalcy is inferred from training during normal use of a system. Thus, rather than writing or specifying the rules *a priori*, here we learn the rules implicitly by observing data in an environment where there are many examples of normal events or data that are in compliance with the implicit rules. This is the approach taken in our work on RAD and other anomaly detectors applied to other audit sources for security tasks.

Anomaly detection systems were first proposed by Denning [18] as an integrated component with host-based misuse detectors and later implemented in NIDES [27] to model normal network behavior in order to detect deviant behavior that may correspond to an attack against a network computer system. W. Lee et al. [30] describe a framework and system for auditing, data mining and feature selection for the automatic computation of intrusion detection models. This framework consists of classification, link analysis and sequence analysis for constructing intrusion detection models and may be applied to network data or to host data. A variety of other work has appeared in the literature detailing alternative algorithms to establish normal profiles and applied to a variety of different audit sources. Some are specific to user commands for masquerade detection [40], others such as SPADE [41], NIDES [27] and PHAD [32] are specific to network protocols and LAN traffic for detecting a variety of attacks, or application or system call-level data for malware detection [21, 28], to name a few. A variety of different modeling approaches have been described in the literature to compute baseline profiles. Many are based upon statistical outlier theory [15]. These include estimating probabilistic or statistical distributions over temporal data [34], supervised machine learning [30] and unsupervised cluster-based algorithms [12].

The work reported in this paper is, to the best of our knowledge, the first sensor devoted to learning-based anomaly detection for the Windows platform and specifically for the Windows Registry. The contribution of this paper is to demonstrate the utility of Windows Registry monitoring as a valuable source of additional information to detect malicious executables as well as the introduction of a learning-based approach to automatically specify normal Registry behavior. The paper also contributes to a deeper understanding of the tradeoffs between a probabilistic and a Support Vector Machine framework for anomaly detection.

There are several other host-based intrusion detection and prevention systems primarily focused on misuse detection driven by signature- or specification-based techniques. We cannot compare the learning-based RAD to a number of commercial products that provide rules-based registry monitoring since those systems are unpublished. It is not possible to expose their internal logic for a detailed comparative evaluation. However, the RAD system is intended to be integrated with other host-based security systems to broaden their coverage and to leverage their behavior-blocking capabilities. RAD provides a learning approach that models the specific characteristics of a distinct machine, rather than depending upon general-purpose rules that may not cover specific unique cases and may provide evidence of malicious activities that may not be covered by a set of rules.

The RAD system automatically learns relationships between all of the features extracted from registry queries, not just process name and key name. More subtle interactions are captured by modeling the conditional probabilities across all pairs of features. Thus the learned models reveal actual system behavior and performance and a completely deployable system can be architected to automatically adapt over time to newly seen behavior patterns when new software is installed.

There is of course a tradeoff between complexity, effective coverage and generality when one compares a machine learning-based approach to a specification-based approach. We posit that a rule based system has some disadvantages when compared to the RAD system. A rule based system requires a human expert to craft specific rules to cover the complete range of policies and behaviors in the system. General Windows Registry policies (for example that only a single application has the right to access a distinct key) may or may not be correct for all applications, or all versions of the underlying platform. In addition these specific behaviors might change over time, making the specific rules incomplete or at worst, very wrong. In addition, generally rule based systems require frequent updates to cover new attacks.Rather than choosing one or the other approach, we believe leveraging both may provide better security; this is one of a number of related research topics we explore in Section 7 on Future Work.

3 Modeling Registry Accesses

3.1 The Windows Registry

In Microsoft Windows, the Registry file is a database of information about a computer's configuration. The Registry contains information that is continually

referenced by many different programs. Information stored in the Registry includes the hardware installed on the system, which ports are being used, profiles for each user, configuration settings for programs, and many other parameters of the system. It is the main storage location for all configuration information for many Window programs. The Windows Registry is also used by some applications as the repository for security related information: policies, user names, profiles and passwords. It stores much of the important run-time configuration information that programs need to execute.

The Registry is organized hierarchically as a tree. Each entry in the Registry is called a key and has an associated value. One example of a Registry key is

`HKCU\Software\America Online\AOL Instant Messenger (TM)`
`\CurrentVersion\Users\aimuser\Login\Password`

This is a key used by the AOL instant messenger program. This key stores an encrypted version of the password for the user name `aimuser`. Upon start up the AOL instant messenger program queries this key in the Registry in order to retrieve the stored password for the local user. Information is accessed from the Registry by individual Registry accesses or queries. The information associated with a Registry query is the key, the type of query, the result, the process that generated the query and whether the query was successful. One example of a query is a read for the key shown above. For example, the record of the query is:

Process: `aim.exe`
Query: `QueryValue`
Key: `HKCU\Software\America Online\AOL Instant Messenger`
`(TM)\CurrentVersion\Users\aimuser\Login\Password`
Response: `SUCCESS`
ResultValue: `" BCOFHIHBBAHF"`

The Windows Registry is an effective data source for monitoring attacks because many attacks are detectable through anomalous Registry behavior. Many attacks take advantage of Windows' reliance on the Registry. Indeed, many attacks themselves rely on the Windows Registry in order to function properly.

Many programs store important information in the Registry, regardless of the fact that other programs can arbitrarily access the information. Although some versions of Windows include security permissions and Registry logging, both features may not be used (because of the computational overhead and the complexity of the configuration options). RAD has been designed to be low overhead and efficient. In the initial implementation we avoided using the native logging tools in favor of a specific sensor that extracts only Registry data of interest to our algorithms. We detail the implementation in Section 5.

3.2 Analysis of Malicious Registry Accesses

Most Windows programs access a certain set of Registry keys during normal execution. Furthermore, users tend to have a typical set of programs that they

routinely run on their machines. This may be the set of all programs installed on the machine or, more commonly, a small subset of these programs. Another important characteristic of Registry activity is that it tends to be regular over time. Most programs either only access the Registry on start-up and shutdown, or access the Registry at specific intervals. This regularity makes the Registry an excellent place to look for irregular, anomalous activity, since a malicious program may substantially deviate from normal activity and can be detected.

Many attacks involve launching programs that have never been launched before and changing keys that have not been changed since the operating system had first been installed by the manufacturer. If a model of normal Registry behavior is computed over clean data, then these kinds of Registry operations will not appear while training the model. Furthermore malicious programs may need to query parts of the Registry to get information about vulnerabilities. A malicious program can also introduce new keys that will help create vulnerabilities in the machine.

Some examples of malicious programs used in this study and how they produce anomalous Registry activity are described below. There are newer versions for several of these for more recent versions of Windows than used in this study. We chose to use these exploits since they were readily available, and they attack known vulnerabilities in the particular version of Windows (NT 4.0) used as our target victim in this work. The behavior of these exploits and there attack upon the Registry are sufficient to demonstrate the utility of RAD.

- **Setup Trojan:** This program when launched adds full read/write sharing access on the file system of the host machine. It makes use of the Registry by creating a Registry structure in the networking section of the Windows keys. The structure stems from `HKLM\Software\Microsoft\Windows \CurrentVersion \Network\LanMan`. It then typically creates eight new keys for its own use. It also accesses `HKLM \Security\Provider` in order to find information about the security of the machine to help determine vulnerabilities. This key is not accessed by any normal programs during training or testing in our experiments and its use is clearly suspicious in nature.
- **Back Orifice 2000:** This program opens a vulnerability on a host machine, which grants anyone with the back orifice client program complete control over the host machine. This program does make extensive use of the Registry, however, it uses a key that is very rarely accessed on the Windows system. `HKLM\Software\Microsoft\VBA\Monitors` was not accessed by any normal programs in either the training or test data, which allowed our algorithm to identify it as anomalous. This program also launches many other programs (`LoadWC.exe, Patch.exe, runonce.exe, bo2k_1_o_intl.e`) as part of the attack all of which made anomalous accesses to the Windows Registry.
- **Aimrecover:** This is a program that steals passwords from AOL users. It is a very simple program that reads the keys from the Registry where the AOL Instant Messenger program stores the user names and passwords. The reason that these accesses are anomalous is because `Aimrecover` is accessing a key that is usually only accessed by the program which created that key.

– **Disable Norton:** This program very simply exploits the Registry so that Norton Antivirus is disabled. This attack toggles one record in the Registry, the key `HKLM\SOFTWARE\INTEL \LANDesk \VirusProtect6\CurrentVersion \Storages \Files\System \Real-TimeScan \OnOff`. If this value is set to 0 then Norton Antivirus real-time system monitoring is turned off. Again this is anomalous because of its access to a key that was created by a different program.
– **L0phtCrack:** This program is probably the most popular password cracking program for Windows machines. This program creates its own section in the Registry involving many create key and set value queries, all of which will be on keys that did not exist previously on the host machine and therefore have not been seen before.

Another important piece of information that can be used in detecting attacks is that all programs observed in our data set, and presumably all programs in general, cause Windows Explorer to access a specific key. The key

`HKLM\Software\Microsoft\Windows NT \CurrentVersion\Image File Execution Options\processName`

where processName is the name of the process being executed, is a key that is accessed by Explorer each time an application is run. Therefore we have a reference point for each specific application being launched to determine malicious activity. In addition many programs add themselves in the auto-run section of the Windows Registry under

`HKLM\Software\Microsoft\Windows \CurrentVersion\Run` .

While this is not malicious in nature, this is a rare event that can definitely be used as a hint that a system is being attacked. Trojan programs such as Back Orifice utilize this part of the Registry to auto load themselves on each boot.

Anomaly detectors do not look for malicious activity directly. They look for deviations from normal activity. It is for this reason that any deviation from normal activity will be declared an alert by the system. The installation of a new program on a system may be a fairly rare event (in relation to normal client use of a machine) and thus may be viewed as anomalous activity. Programs often create new sections of the Registry and many new keys on installation. This may cause a false alarm, much like adding a new machine to a network may cause an alarm on an anomaly detector that analyzes network traffic. Hence, an anomaly detector such as RAD may generate too many false positives, or worse, it may be blind to malicious installations if the logic of the system chooses to ignore program installs.

There are a few possible solutions to this problem. Malicious programs are often stealthy and install quietly so that the user does not know the program is being installed. This is not the case with most user initiated (legitimate) application installations that make themselves (loudly) known.

A complete host-based IDS solution that incorporates RAD as a component may be architected to handle this case. For example, RAD may be modified to use

a specific model trained over install shield runs, which could model the behavior of install shield separately from normal Registry accesses. Another option is to simply prompt the user when a RAD alarm occurs so that the user can let the anomaly detection system know that a legitimate program is being installed and therefore the anomaly detection model needs to be updated with a newly available training set gathered in real-time. This is a typical user interaction in many application installations where user feedback is requested for configuration information. In addition a white list of good programs can be gathered in this way to inform the system that the user has approved of some specific program installs.

A full treatment of these design issues are beyond the scope of this paper and are a core part of our future work we describe in Section 7. For the present paper, we first test the thesis that Registry monitoring, and learning-based anomaly detection in particular, provides a useful component in the arsenal of security features for a host system.

4 Registry Anomaly Detection

The RAD system has three basic components: an audit sensor, a model generator, and an anomaly detector. The audit sensor logs Registry activity to either a database where it is stored for training, or to the detector to be used for analysis. The model generator reads data from the database and creates a model of normal behavior. This model is then used by the anomaly detector to decide whether each new Registry access should be considered anomalous.

In order to detect anomalous Registry accesses, five features are extracted from each Registry access. Using these feature values over normal data, a model of normal Registry behavior is generated. When detecting anomalies, the model of normalcy determines whether the feature values of the current Registry access are *consistent* with the normal data. If they are not consistent, the algorithm labels the access as anomalous.

4.1 RAD Data Model

The RAD data model consists of five features directly gathered from the Registry sensor. The five raw features used by the RAD system are as follows.

- **Process:** This is the name of process accessing the Registry. This is useful because it allows the tracking of new processes that did not appear in the training data.
- **Query:** This is the type of query being sent to the Registry, for example, `QueryValue`, `CreateKey`, and `SetValue` are valid query types. This allows the identification of query types that have not been seen before. There are many query types but only a few are used under normal circumstances.
- **Key:** This is the actual key being accessed. This allows our algorithm to locate keys that are never accessed in the training data. Many keys are used

only once for special situations like system installation. Some of these keys can be used to create vulnerabilities.
- **Response:** This describes the outcome of the query, for example `success`, `not found`, `no more`, `buffer overflow`, and `access denied`.
- **Result Value:** This is the value of the key being accessed. This will allow the algorithm to detect abnormal values being used to create abnormal behavior in the system.

Feature	**aim.exe**	**aimrecover.exe**
Process	aim.exe	aimrecover.exe
Query	QueryValue	QueryValue
Key	HKCU\Software\America Online \AOL Instant Messenger (TM) \CurrentVersion\Users \aimuser\Login\Password	HKCU\Software\America Online \AOL Instant Messenger (TM) \CurrentVersion\Users \aimuser\Login\Password
Response	SUCCESS	SUCCESS
Result Value	" BCOFHIHBBAHF"	" BCOFHIHBBAHF"

Table 1. Registry Access Records. Two Registry accesses are shown. The first is a normal access by AOL Instance Messenger to the key where passwords are stored. The second is a malicious access by AIMrecover to the same key. Note that the pairs of features are used to detect the anomalous behavior of AIMrecover.exe. This is because under normal circumstances only AIM.exe accesses the key that stores the AIM password. Another process accessing this key should generate an anomaly alert.

4.2 PAD Anomaly Detection Algorithm

Using the features that we extract from each Registry access, we train a model over normal data. This model allows us to classify Registry accesses as either normal or not.

Any anomaly detection algorithm can be used to perform this modeling. Since we aim to monitor a significant amount of data in real-time, the algorithm must be very efficient. Furthermore, one of the most vexing problems for anomaly detection algorithms is how to treat noise in the training data. Many papers present standard modeling algorithms that rely upon cleaned training data which is simply impractical in contexts where there is far too much data to inspect and clean. Previous algorithms, for examples those based upon clustering (see [12] and references cited therein) had no systematic way of treating noise other than to assume small or low density clusters may be outliers. Similarly, modeling algorithms based upon the Support Vector Machine framework (treated in Section 4.3) assume outliers are detectable by recognizing points on the "wrong side" of the maximal margin hyperplane.

The Probabilistic Anomaly Detection (PAD) algorithm was designed to train a normal *probabilistic* model in the presence of noise. The algorithm was inspired by the heuristic algorithm that was proposed by Chan and Mahoney in the PHAD system [32], but is more robust. PHAD would not detect data that represents an attack in the training set because it would not label such data as an anomaly; it would assume all data was normal. PAD's design is based upon the premise that low probability events or data records with a low likelihood are regarded as noise, and hence are recognized as anomalies. If one assumes that attack data are a minority of the training data (otherwise attacks would be high frequency events) than they would be recognized as attacks by PAD implicitly as low probability events and treated as anomalies at detection time. (If attacks were prevalent and high probability events, then they are normal events, by definition.)

PAD also extends the PHAD algorithm by considering conditional probabilities over all pairs of features of the data. This modeling tends to identify a broader class of unlikely data with low conditional probabilities. This is crucial to understanding PAD's performance as an anomaly detection algorithm. It is often the case that multi-variate data is indeed inconsistent with prior training data not because of any individual feature value (which may have been sampled frequently in the training data), but rather in the combination of feature values which may never or rarely have been seen before.

In general, a principled probabilistic approach to anomaly detection can be reduced to density estimation. If we can estimate a density function $p(x)$ over the normal data, we can define anomalies as data elements that occur with low probability. In practice, estimating densities is a very hard problem (see the discussion in Schölkopf et al., 1999 [35] and the references therein.) In our setting, part of the problem is that each of the features has many possible values. For example, the *Key* feature has over $30,000$ values in our training set. Since there are so many possible feature values, relatively rarely does the same exact record occur in the data. Data sets with this characterization are referred to as sparse.

Since probability density estimation is a very hard problem over sparse data, we propose a different method for determining which records from a sparse data set are anomalous. We define a set of consistency checks over the normal data. Each consistency check is applied to an observed record. If the record fails any consistency check, we label the record as anomalous.

We apply two kinds of consistency checks. The first kind of consistency check evaluates whether or not a feature value is consistent with observed values of that feature in the normal data set. We refer to this type of consistency check as a first order consistency check. More formally, each Registry record can be viewed as the outcome of 5 random variables, one for each feature, X_1, X_2, X_3, X_4, X_5. Our consistency checks compute the likelihood of an observation of a given feature which we denote $P(X_i)$.

The second kind of consistency check handles pairs of features as motivated by the example in Table 1. For each pair of features, we consider the conditional

probability of a feature value given another feature value. These consistency checks are referred to as second order consistency checks. We denote these likelihoods $P(X_i|X_j)$. Note that for each value of X_j, there is a different probability distribution over X_i.

In our case, since we have 5 feature values, for each record, we have 5 first order consistency checks and 20 second order consistency checks. If the likelihood of any of the consistency checks is below a threshold, we label the record as anomalous. PAD is designed to estimate all such consistency checks, some of which may never possibly generate an anomaly alert. Some simple enhancements to PAD may be made to eliminate or prune these consistency checks which produces a computational performance improvement, but has no effect on the detection performance of the pruned model.

What remains to be shown is how we compute the likelihoods for the first order ($P(X_i)$) and second order ($P(X_i|X_j)$) consistency checks. Note that from the normal data, we have a set of observed counts from a discrete alphabet for each of the consistency checks. Computing these likelihoods reduces to simply estimating a multinomial. In principal we can use the maximum likelihood estimate which just computes the ratio of the counts of a particular element to the total counts. However, the maximum likelihood estimate is biased when there is relatively small amounts of data. When estimating sparse data, this is the case. We can smooth this distribution by adding virtual counts to every possible element, thereby giving non-zero probability mass to yet unseen elements which may appear in the future. This is equivalent to using a Dirichlet estimator [17]. For anomaly detection, as pointed out in Mahoney and Chan, 2001 [32], it is critical to take into account how likely we are to observe an unobserved element. Intuitively, if we have seen many different elements, we are more likely to see unobserved elements as opposed to the case where we have seen very few elements. This intuition explains why PAD performs well as an anomaly detection algorithm that trains well even with noisy training data.

To estimate our likelihoods we use the estimator presented in Friedman and Singer, 1999 [22] which explicitly estimates the likelihood of observing a previously unobserved element. The estimator gives the following prediction for element i

$$P(X = i) = \frac{\alpha + N_i}{k^0\alpha + N}C \tag{1}$$

if element i was observed and

$$P(X = i) = \frac{1}{L - k^0}(1 - C) \tag{2}$$

if element i was not previously observed. α is a prior count for each element, N_i is the number of times i was observed, N is the total number of observations, k^0 is the number of different elements observed, and L is the total number of possible elements or the alphabet size. The scaling factor C takes into account how likely it is to observe a previously observed element versus an unobserved

element. C is computed by

$$C = \left(\sum_{k=k^0}^{L} \frac{k^0\alpha + N}{k\alpha + N} m_k \right) \left(\sum_{k \geq k^0} m_k \right)^{-1} \tag{3}$$

where $m_k = P(S = k) \frac{k!}{(k-k^0)!} \frac{\Gamma(k\alpha)}{\Gamma(k\alpha+N)}$ and $P(S = k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability. Although the computation of C is expensive, it only needs to be done once for each consistency check at the end of training.

While it is true that as more elements are observed, the less the prior influences the estimator and the better the estimator will become, this should hardly be surprising. *All* estimators improve as the amount of observed data increases, the job of the prior is to make intelligent predictions in the case of sparse data. The risk of overwhelming useful information given by the data with the prior is minimal since the hyperparameter α can be set as to modify the strength of the prior.

The prediction of the probability estimator is derived using a mixture of Dirichlet estimators each of which represent a different subset of elements that have non-zero probability. Details of the probability estimator and its derivation are given in [22].

PAD is relatively efficient in space and time, even though it builds an very detailed model of the training data. The PAD algorithm takes time $O(v^2R^2)$, where v is the number of unique record values for each record component and R is the number of record components. The space required to run the algorithm is $O(vR^2)$.

Note that this algorithm labels every Registry access as either normal or anomalous. Programs can have anywhere from just a few Registry accesses to several thousand. This means that many attacks may be represented by large numbers of anomalous records. In the experimental evaluation presented in Section 6.2 we compare the statistical accuracy of RAD from the perspective of correctly labeled Registry records as well as correctly labeled processes. Processes are deemed anomalous if the record they generate with the minimum score is below the threshold. (Alternative alert logic can of course be defined.)

The performance of the detector obviously varies with the threshold setting and the particular decision logic (record based or process based). (A fuller treatment evaluating anomaly detectors and their coverage is given by Maxion [13].) However, since relatively few processes and malicious programs are available in proportion to the number of Registry records, the comparison between PAD and the OCSVM algorithm, described next, is best viewed in terms of the record based performance results. The comparative evaluation of detection performance is presented in Section 6.3 from which we can draw certain conclusions about the alternative algorithms.

4.3 OCSVM Anomaly Detection Algorithm

As an alternative to the PAD algorithm for model generation and anomaly detection, we apply an algorithm, described in [23], based on the One-Class Support Vector Machine (OCSVM) algorithm given in [35]. Previously, the OCSVM has not been used in host-based anomaly detection systems. The OCSVM code we used [14] has been modified to compute kernel entries dynamically due to memory limitations. The OCSVM algorithm maps input data into a high dimensional feature space (via a kernel function) and iteratively finds the maximal margin hyperplane which best separates the training data from the origin. As in the PAD algorithm, the OCSVM trains on all normal data. The OCSVM may be viewed as a regular two-class Support Vector Machine (SVM) where all the training data lies in the first class, and the origin is taken as the only member of the second class. Thus, the hyperplane (or linear decision boundary) corresponds to the classification rule:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + \mathbf{b} \tag{4}$$

where $\mathbf{w}$ is the normal vector and b is a bias term. The OCSVM solves an optimization problem to find the rule f with maximal geometric margin. We can use this classification rule to assign a label to a test example $\mathbf{x}$. If $f(\mathbf{x}) < \mathbf{0}$ we label $\mathbf{x}$ as an anomaly, otherwise it is labeled normal. In practice there is a trade-off between maximizing the distance of the hyperplane from the origin and the number of training data points contained in the region separated from the origin by the hyperplane.

4.4 Kernels

Solving the OCSVM optimization problem is equivalent to solving the dual quadratic programming problem:

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{5}$$

subject to the constraints

$$0 \leq \alpha_i \leq \frac{1}{\nu l} \tag{6}$$

and

$$\sum_{i} \alpha_i = 1 \tag{7}$$

where α_i is a Lagrange multiplier (or "weight" on example i such that vectors associated with non-zero weights are called "support vectors" and solely determine the optimal hyperplane), ν is a parameter that controls the trade-off between maximizing the distance of the hyperplane from the origin and the number of

data points contained by the hyperplane, l is the number of points in the training dataset, and $K(x_i, x_j)$ is the kernel function. By using the kernel function to project input vectors into a feature space, we allow for nonlinear decision boundaries. This means that although we only use a linear decision boundary to cut out one region of *feature space*, this region can map to arbitrary multimodal regions in *input space* (for example, in Gaussian RBF kernels, which we define in the next few lines; the amount of multimodality can be controlled using the variance, σ, parameter).

Given a feature map:

$$\phi : X \rightarrow R^N \tag{8}$$

where ϕ maps training vectors from input space X to a high-dimensional feature space, we can define the kernel function as:

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \tag{9}$$

Feature vectors need not be computed explicitly, and in fact it greatly improves computational efficiency to directly compute kernel values $K(x, y)$. We used three common kernels in our experiments:

Linear kernel: $K(x, y) = (x \cdot y)$
Polynomial kernel: $K(x, y) = (x \cdot y + 1)^d$, where d is the degree of the polynomial
Gaussian RBF kernel: $K(x, y) = e^{-\|x-y\|^2/(2\sigma^2)}$, where σ^2 is the variance

Our OCSVM algorithm uses sequential minimal optimization to solve the quadratic programming problem, and therefore takes time $O(dL^3)$, where d is the number of dimensions and L is the number of records in the training dataset. Typically, since we are mapping into a high dimensional feature space d exceeds R^2 from the PAD complexity. Also for large training sets L^3 will significantly exceed v^2, thereby causing the OCSVM algorithm to be a much more computationally expensive algorithm than PAD. An open question remains as to how we can make the OCSVM system in high bandwidth real-time environments work well and efficiently. All feature values for every example must be read into memory, so the required space is $O(d(L+T))$, where T is the number of records in the test dataset. Although this is more space efficient than PAD, we compute our kernel values dynamically in order to conserve memory, resulting in the added d term to our time complexity. If we did not do this the memory needed to run this algorithm would be $O(d(L+T)^2)$ which is far too large to fit in memory on a standard computer for large training sets (which are inherent to the Windows anomaly detection problem).

5 Architecture

The basic architecture of the RAD system consists of three components, the Registry auditing module (RegBAM), the model generator, and the real-time anomaly detector. An overview of the RAD architecture is shown in Figure 1.

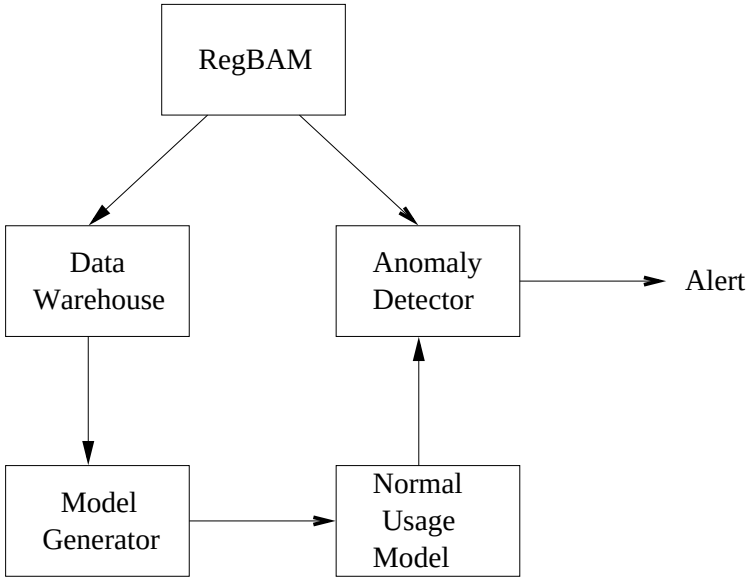

Fig. 1. The RAD System Architecture. RegBAM outputs to the data warehouse during training model and to the anomaly detector during detection mode.

5.1 Registry Basic Auditing Module

The RAD sensor is composed of a Basic Auditing Module (BAM) for the RAD system which monitors accesses to the Registry. A BAM implements an architecture and interface for sensors across the system. It includes a hook into the audit stream (in this case the Registry) and various communication and data-buffering components. The BAM uses an XML data representation similar to the IDMEF standard (of the IETF) for IDS systems [26]. The BAM is described in more detail in [25].

The Registry BAM (RegBAM) runs in the background on a Windows machine as it gathers information on Registry reads and writes. RegBAM uses Win32 hooks to the underlying Win32 subsystem to tap into and log all reads and writes to the Registry. RegBAM is akin to a wrapper and uses a similar architecture to that of SysInternal's Regmon [36].

While gathering Registry data in real-time, RegBAM can be configured in one of two ways. The first as the audit data source for model generation. When RegBAM is used as the data source, the output data is sent to a database where it is stored and later used by the model generator described in Section 5.2 [25]. The second use of RegBAM, is as the data source for the real-time anomaly detector described in Section 5.3. While in this mode, the output of RegBAM is passed directly to the anomaly detector where it is processed in real-time and evaluated in real-time for anomalies. (The scores for the individual Registry accesses, are displayed by a GUI as well.)

An alternative method of collecting Registry access data is to use the Windows auditing mechanism over the Windows Registry. All Registry accesses can be logged in the Windows Event Log with each read or write generating multiple records in the Event Log. However, this method was problematic in our first implementation because the event logs are not designed to handle such a large amount of data. Simple tests demonstrated that by turning on all Registry auditing the Windows Event Logger caused a major resource drain on the host machine, and in many cases caused the machine to crash. The RegBAM application provides an efficient method for monitoring all Registry activity, with far less overhead than the native tools provided by the Windows operating system by only extracting the feature values needed by the modeler and detector. Recent updates to Windows have improved the native auditing tools and hence RegBAM could be redesigned (as originally planned) to take advantage of these native utilities. In either case, it is the designers choice which of the methods of tapping Registry queries would perform best in real-time operation.

5.2 Model Generation Infrastructure

The initial RAD system architecture is similar to the Adaptive Model Generation (AMG) architecture reported in [25], but uses RegBAM to collect Registry access records. Using this database of collected records from a training run, the model generator computes a model of normal usage. A second version of RAD has been implemented that is entirely host-based and operates in real-time as a standalone sensor. Thus, RAD may be used as a network application gathering remote sensor data using RegBAM running on each individual host, or as a standalone application on each host. In the case where a centralized database is used to store remote host data, models may be easily shared and distributed if one so desires.

The model generator uses the algorithms discussed in Section 4 to build models that represent normal usage. It utilizes the data stored in the database which was generated by RegBAM during training. The model is stored in a compact binary representation. A typical hour of normal usage generates about 100,000 records and in binary mode of storage totals about 15 Megabytes in size.

This core architecture builds a model from some particular set of training data acquired during some chosen period of time. The question then is how to treat changing or drifting environments; models may have to be relearned and refreshed as the host environment changes over time. In the real-time RAD implementation recently completely, the system is entirely hands-free and capable of automatic model update. The models computed can either be *cumulative* or *adaptive*. In the mode where a cumulative model is built, a single model is computed and constantly updated with new data (data that is deemed by the detector as normal) from all available data inspected by the detector and the model is used in perpetuity. Adaptive models are trained incrementally in *training epochs*. During some period of time, some percentage of available training data is used to generate a model. A model computed during one epoch may be retired in favor of a second new model computed during a second training

epoch. The former tends to grow a model that increases in size over time, while the latter tends to use fairly constant space. The tradeoffs, however, in accuracy between the two approaches is not fully understood and is an active part of our ongoing research.

5.3 Real-Time Anomaly Detector

For real-time detection, RegBAM feeds live data for analysis by the anomaly detector. The anomaly detector will load the normal usage model created by the model generator and begin reading each record from the output data stream of RegBAM. The algorithm discussed in Section 4 is then applied against each record of Registry activity. The score generated by the anomaly detection algorithm is compared to a configurable threshold to determine if the record should be considered anomalous. A list of anomalous Registry accesses are stored and displayed as part of the detector output. A user configured threshold allows the user to customize the alarm rate for the particular environment. Lowering the threshold, will result in more alarms being issued. Although this can raise the false positive rate, it can also decrease the rate of false negatives.

We note that the activity of the RegBAM itself is also evaluated both during the training and testing of the system. RegBAM will access the Registry to load and save user configurations of the GUI and model checksums from trained models. Hence, RegBAM is itself modeled by RAD.

The alarm system on the real-time version of RAD also gives the user the ability to kill a running process or to add the specific binary of the process to a white list so it is not incorrectly killed in the future. Thus, RAD provides user control over incorrect actions applied by the real-time version of RAD.

5.4 Efficiency Considerations

In order for a system to detect anomalies in a real-time environment it can not consume excessive system resources. This is especially important in Registry attack detection because of the heavy amount of traffic that is generated by applications interacting with the Registry. While the amount of traffic can vary greatly from system to system, in our experimental setting (described below) the traffic load was about 100,000 records per hour.

We created efficient data structures to buffer data writes and anomaly detection calculations. In addition per-process history is efficiently stored and retrieved using a hashtable structure along with other efficient data structures for calculating average history.

Although the system can run efficiently on most machines, our distributed architecture is designed to minimize the resources used by the host machine making it possible to spread the work load on to several separate machines. This allows a light installation of only the RegBAM sensor on the host machine, while processing takes place in a central location. Having a lightweight install will increase network load due to the communication between components. These two loads can be balanced by configuring a group of hosts to create the proper

proportion between host system load and network load. The RegBAM module is a far more efficient way of gathering data about Registry activity than full auditing with the Windows Event Log.

Our measurements indicated a CPU usage between 3% and 5% in total for both the real-time RegBAM and PAD processes running on a Pentium Celeron 1GHZ with 512MB RAM. The actual memory footprint was under 3 Megabytes of system RAM, far less than most typical programs which can consume from 10 to 30 Megabytes of system memory.

6 Evaluation and Results

The system was evaluated by measuring the detection performance over a set of collected data which contains some attacks. Since there are no other existing publicly available detection systems that operate on Windows Registry data we were unable to compare our performance to other systems directly. However, our goal is to evaluate the relative performance between the two algorithms, PAD and OCSVM. We describe these results in the following sections.

6.1 Data Generation

In order to evaluate the RAD system, we gathered data by running a Registry sensor on a host machine. Beyond the normal installation and execution of standard programs, such as Microsoft Word, Internet Explorer, and Winzip, the training also included performing housekeeping tasks such as emptying the Recycling Bin and using the Control Panel. All data was acquired during routine use of a Windows machine by a real user. All data used for this experiment is publicly available online in text format at `http://www.cs.columbia.edu/ids/rad`. The data includes a time stamp and frequency of the launched programs in relation to each other.

The RegBAM system can run on any flavor of Windows. This includes Windows 98, Windows XP, Windows 2000, NT 4.0 and above. We used NT 4.0 for the experiments reported in this paper since at the time of the experiment we had a sufficient collection of malicious programs designed for that operating system running on a machine we were willing to victimize.

The training data for our experiment was collected on Windows NT 4.0 over two days of normal usage (in our lab). We informally define normal usage to mean what we believe to be typical use of a Windows platform in a home setting. For example, we assume all users would log in, check some internet sites, read some mail, use word processing, then log off. This type of session is assumed to be relatively typical of many computer users. Normal programs are those which are bundled with the operating systems, or are in use by most Windows users. Creating realistic testing environments is a very hard task and testing the system under a variety of environments is a direction for future work.

The simulated home use of Windows generated a clean (attack-free) dataset of approximately 500,000 records. The system was then tested on a full day of test

data with embedded attacks executed. This data was comprised of approximately 300,000 records most of which were normal program executions interspersed with approximately 2,000 attacks. The normal programs run between attacks were intended to simulate an ordinary Windows session. The programs used were Microsoft Word, Outlook Express, Internet Explorer, Netscape, AOL Instant Messenger, and others.

The attacks run include publicly available attacks such as aimrecover, browselist, bo2kss (back orifice), install.exe xtxp.exe both for backdoor.XTCP, l0phtcrack, runattack, whackmole, and setuptrojan. Attacks were only run during the one day of testing throughout the day. Among the twelve attacks that were run, four instances were repetitions of the same attack. Since some attacks generated multiple processes there are a total of seventeen distinct processes for each attack. All of the processes (either attack or normal) as well as the number of Registry access records in the test data is shown in Table 2.

Some of the attacks were run twice or more. Many programs act differently when executed a second time within a Windows session. In the experiments reported below our system was less likely to detect a previously successful attack on the second execution of that attack. The reason is that a successful attack creates permanent changes to the Registry and hence on subsequent queries the attack no longer appears abnormal. Thus the next time the same attack is launched it is more difficult to detect since it interacts less with the Registry.

We observed that this is common for both malicious and regular applications since many applications will do a much larger amount of Registry writing during installation or when executed for the first time.

6.2 Experiments

We trained the two anomaly detection algorithms presented in Section 4 over the normal data and evaluated each record in the testing set. We evaluate our system by computing two statistics. We compute the *detection rate* and the *false positive rate*.

The typical way to evaluate the performance of RAD would be to measure detection performance over processes labeled as either normal or malicious. However, with only seventeen malicious processes at our disposal in our test set, it is difficult to obtain a robust evaluation for the system. We do discuss the performance of the system in terms of correctly classified processes, but also measure the performance in terms of the numbers of records correctly and incorrectly classified. Future work on RAD will focus on testing over long periods of time to measure significantly more data and process classifications as well as alternative means of alarming on processes. (For example, a process may be declared an attack on the basis of one anomalous record it generates, or perhaps on some number of anomalous records.) There is also an interesting issue to be investigated regarding the decay of the anomaly models that may be exhibited over time, perhaps requiring regenerating a new model.

The detection rate reported below is the percentage of records generated by the malicious programs which are labeled correctly as anomalous by the model.

The false positive rate is the percentage of normal records which are mislabeled anomalous. Each attack or normal process has many records associated with it. Therefore, it is possible that some records generated by a malicious program will be mislabeled even when some of the records generated by the attack are accurately detected. This will occur in the event that some of the records associated with one attack are labeled normal. Each record is given an anomaly score, S, that is compared to a user defined threshold. If the score is greater than the threshold, then that particular record is considered malicious. Figure 2 shows how varying the threshold effects the output of the detector. The actual recorded scores for the PAD algorithm plotted in the figure are displayed in Table 3. In Tables 5 and 6, information on the records and their discriminants are listed for the linear and polynomial kernels using binary feature vectors within the OCSVM algorithm.

Table 2 is sorted in order to show the results for classifying processes. From the table we can see that if the threshold is set at 8.497072, we would label the processes `LOADWC.EXE` and `ipccrack.exe` as malicious and would detect the Back Orifice and IPCrack attacks. Since none of the normal processes have scores that high, we would have no false positives. If we lower the threshold to 6.444089, we would have detected several more processes from Back Orifice and the BrowseList,BackDoor.xtcp, SetupTrojan and AimRecover attacks. However, at this level of threshold, the following processes would be labeled as false positives: `systray.exe`, `CSRSS.EXE`, `SPOOLSS.EXE`, `ttssh.exe`, and `winmine.exe`. Similarly, for the OCSVM algorithm results in Table 5, it is seen that if the threshold is set at -1.423272, then the bo2kcfg.exe would be labeled as attack, as would msinit.exe and ononce.exe. False labels would be given to WINLOGON.exe, systray.exe and other normal records. The measurements reported next are cast in terms of Registry query records. Recall, our primary objective in this work is to compare the alternative algorithms, PAD and OCSVM, where we now turn our attention.

6.3 Detection

We can measure the performance of our detection algorithms on our test data by plotting their Receiver Operator Characteristic (ROC) curves. The ROC curve plots the percentage of false positives (normal records labeled as attacks) versus the percentage of true positives. As the discriminant threshold increases, more records are labeled as attacks. Random classification results in 50% of the area lying under the curve, while perfect classification results in 100% of the area lying under the curve. We plot the ROC curve for the PAD algorithm shown in Figure 2 and Table 3.

We obtained kernels from binary feature vectors by mapping each record into a feature space such that there is one dimension for every unique entry for each of the five given record values. This means that a particular record has the value 1 in the dimensions which correspond to each of its five specific record entries, and the value 0 for every other dimension in feature space. We then computed

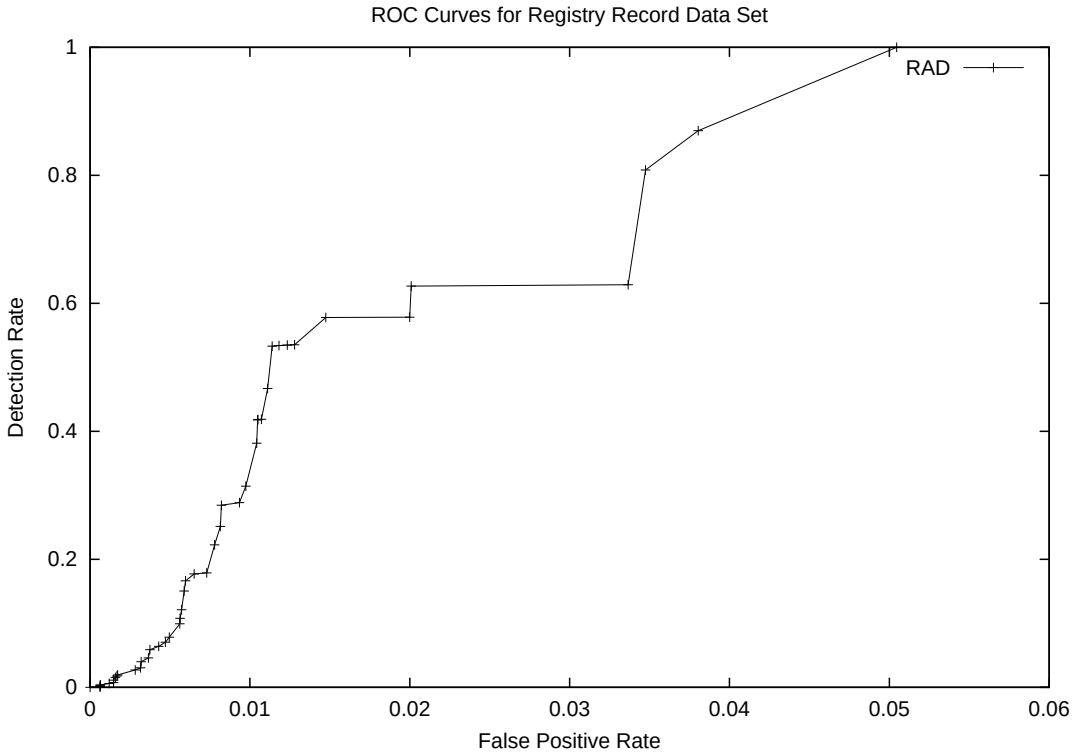

Fig. 2. ROC showing performance by varying the score threshold on the data set.

linear kernels, second order polynomial kernels, and Gaussian kernels using these feature vectors for each record.

We also computed kernels from frequency-based feature vectors such that for any given record, each feature corresponds to the number of occurrences of the corresponding record component in the training set. For example, if the second component of a record occurs three times in the training set, the second feature value for that record is three. We then used these frequency-based feature vectors to compute linear and polynomial kernels.

Results from our OCSVM system are shown with the results of the PAD system on the same dataset in Figures 3 and 4. Figure 3 is the ROC curve for the linear and polynomial kernels using binary feature vectors. We have used a sigma value of 0.84 for our Gaussian function. The binary linear kernel most accurately classifies the records. Figure 4 is the ROC curve for the linear and polynomial kernels using frequency-based feature vectors. The frequency-based linear and frequency-based polynomial kernels demonstrate similar classification abilities. Overall, in our experiments, the linear kernel using binary feature vectors results in the most accurate model.

Many of the false positives were from processes that were simply not run

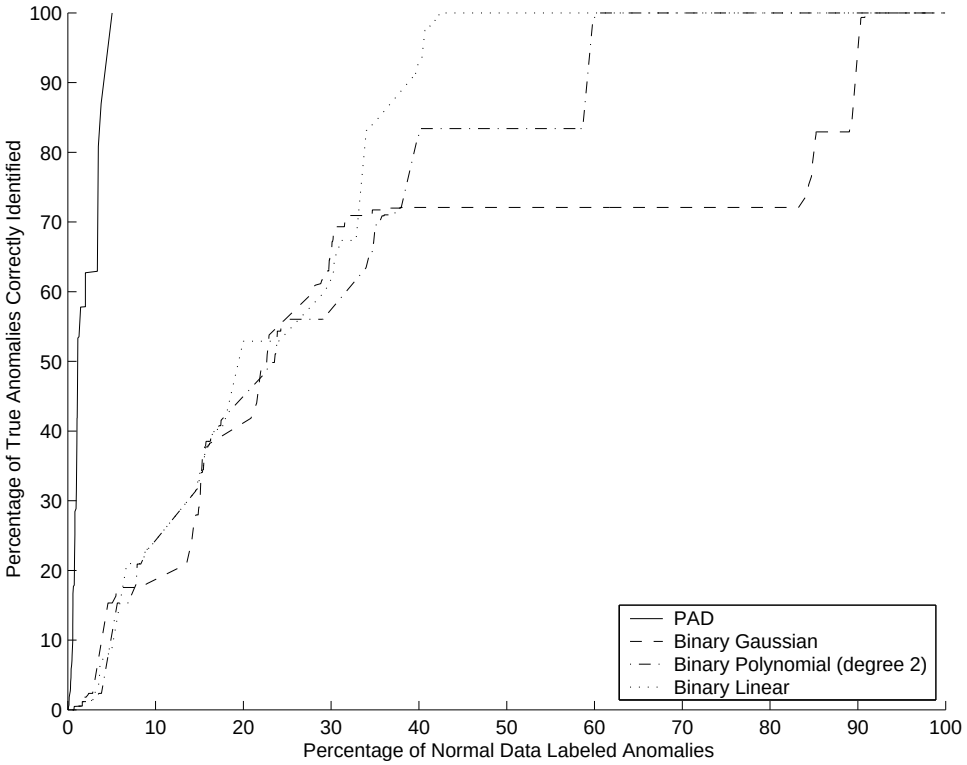

Fig. 3. ROC curve for the kernels using binary feature vectors with the OCSVM algorithm (false positives versus true positives).

as a part of the training data but were otherwise normal Windows programs. A thorough analysis of what kinds of processes generate false positives is a direction for future work.

Part of the reason why the RAD system is successfully able to discriminate between malicious and normal records is that accesses to the Windows Registry are very regular, which makes normal Registry activity relatively easy to model.

The results of the OCSVM system produce less accurate results than the PAD system. The PAD model is able to more accurately discriminate between normal and anomalous records. The OCSVM system labels records with fair accuracy, but could be improved with a stronger kernel, where more significant information is captured in the data representation.

The ability of the OCSVM to detect anomalies is highly dependent on the information captured in the kernel (the data representation). Our results show that the kernels computed from binary feature vectors or frequency-based feature vectors alone do not capture enough information to detect anomalies as well as the PAD algorithm. With other choices of kernels, similar results will likely occur unless a novel technique which incorporates more discriminative information is

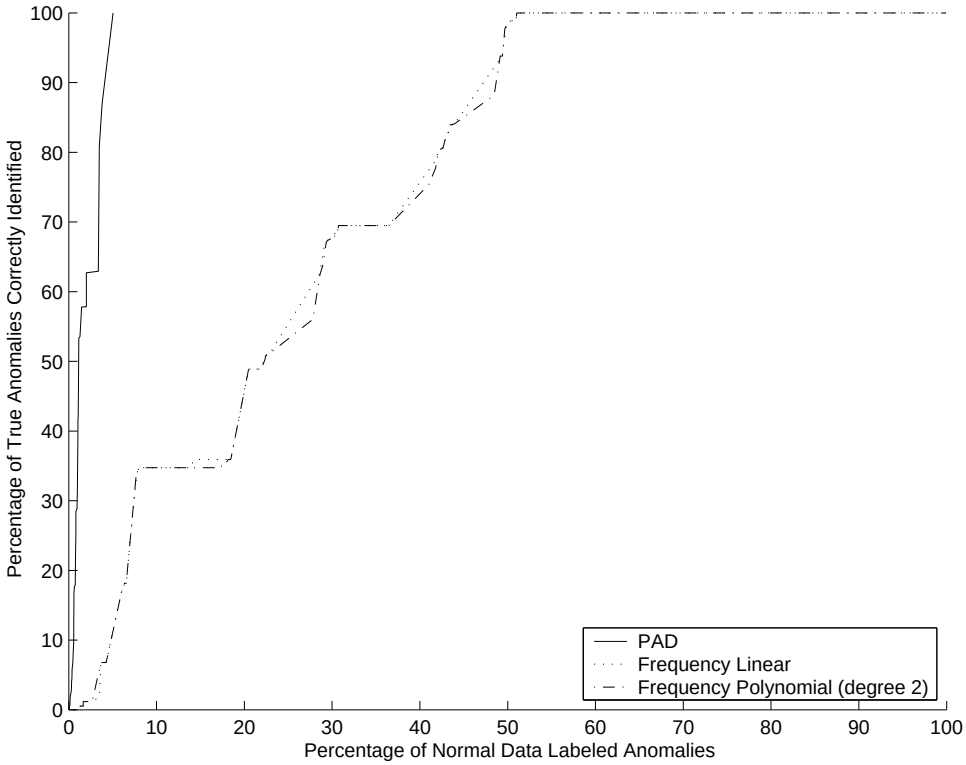

Fig. 4. ROC curve for the kernels using frequency-based feature vectors with the OCSVM algorithm (false positives versus true positives).

used to compute the kernel. A simple example of this is if we have a dataset in which good discrimination depends upon pairs of features, then we will not be able to discriminate well with a linear decision boundary regardless of how we tweak its parameters. However, if we use a polynomial kernel we can account for pairs of features and will discriminate well. In this manner, having a well defined kernel which accounts for highly discriminative information is extremely important. For the purpose of this research, we believe our kernel choices are sufficient to reliably compare the OCSVM system with PAD.

The advantage of the PAD algorithm over the OCSVM system lies in the use of a hierarchical prior to estimate probabilities. A scaling factor (see equation (4)) is computed and applied to a Dirichlet prediction which assumes that all possible elements have a non-zero chance of being seen, giving varying probability mass to outcomes unseen in the training set. In general, knowing the likelihood of encountering a previously unobserved feature value is extremely important for anomaly detection, and it would be valuable to be able to incorporate this information into a kernel for use in an OCSVM system.

7 Discussion and Future Work

The work on RAD and anomaly detection in general has created a furtile ground for future research covering practical issues of useability and reliability, as well as some fundamental issues regarding adversaries who will seek the means of thwarting this new generation of detection systems. We address several of these issues and highlight our future work.

7.1 False Positives and Correlated Sensors

The RAD system has been presented as a component sensor of a host-based intrusion detection system that signals an alert if and when anomalous Registry accesses are detected. These alerts are intended to be correlated with other sensor alerts to mitigate against false positives, and identify true attacks more reliably. Hence, the results for PAD, for example, indicating a 5% false positive rate with a perfect detection capability for the data sets used in this study, should not be viewed as a noisy sensor with too many false positives. Rather, one should regard RAD as a fairly good sensor providing substantive evidence of a security event to be validated by other evidence from other sensors.

We plan on testing the system under a variety of environments and conditions to better understand its performance in a long-lived real-time setting. Future plans include combining the RAD system with another detector that evaluates Windows Event Log data, and a file system access anomaly detector. This will allow for various correlation algorithms to be used to make more accurate system behavior models which we believe will provide a more accurate anomaly detection system with better coverage of attack detection. Part of our future plans for the RAD system include adding data clustering and aggregation capabilities. Aggregating alarms will allow for subsets of Registry activity records to be considered malicious as a group initiated from one complex multi-stage attack rather than individual attacks.

Furthermore, a system of correlated sensors to detect true attacks can be extended to an intrusion *prevention* system by integrating the detector output with a decision process that challenges the execution of a process that has caused an alert (or a set of alerts). This challenge may either terminate the process, or alternatively the alert may be presented to the user who may decide to allow the process to complete or not. This has the advantage of informing the user of an anomalous event that he or she otherwise would not be aware of, and provides the means of mitigating against any false positives from the component sensors causing an incorrect action. Of course the rate or frequency of user challenges may render such a host-based intrusion prevention system a frustratingly annoying security system if it generates too many alarms in too short a time frame. The results reported for the test cases studied indicate that RAD can be an effective sensor with high accuracy and low false positive rates. However, the conditions that led to an incorrect RAD decision may be regarded and used as additional training data to reduce the implied false positives even further. This interactive mode is familiar to many users who are likely already accustomed

to interact with security systems, such as browser-initiated pop-up Windows asking whether to accept a cookie from a website, or to allow certain network connections from personal firewall applications.

7.2 Training Regimen

The experimental results reported were completed using a pristine Windows machine where the Registry data was guaranteed to be normal data without embedded attacks. This approach is sensible if one presumes a standard Windows environment is shipped with a first factory standard Registry data model precomputed for the user. However, each machine would require subsequent training of updated models specific to how the machine is used and updated with additional software. This process must be completely automated in a hands-free fashion with little user intervention. The newer real-time version of RAD has been designed for this purpose. (Our ongoing work concerns long-lived process execution, and a full evaluation of RAD performance over many thousands of process executed over weeks of typical use. Results of this test of RAD are forthcoming.)

When training updated models one cannot legitimately assume training data would be attack free. We have cast the anomaly detection algorithms in a fashion that allows training over data that does not necessarily need to be clean, normal data. The very nature of the training algorithms (the Probabilistic Anomaly Detection algorithm is based upon an estimated probability distribution and the Support Vector Machine algorithm is based on a maximal margin hyperplane) allow for some amount of noise in the training data to still operate effectively. We have not tested this feature in the work reported here on a Windows platform. However, PAD has been effectively deployed in a network sensor that trains a normal model in a completely unsupervised fashion. The core requirement is the assumption that attacks or noise are a minority of the training data. (If this is not the case, then normal data will be the statistical outliers, which in practical contexts is rather unlikely.)

7.3 Continuous Learning and Self-Calibration

In our future work several additional features are required to make RAD an easy to deploy and use security technology. This includes the means of continuously learning updated models, as the platform is updated and the environment drifts, and self-calibration of the model output thresholds. In the former case, we are experimenting with several strategies to continuously update learned models. In the simplest case once a model has been produced by the training algorithm and deployed to the run-time detector, the learning algorithm continues to operate in the background training a new model in a scheduled fashion. The new model may replace the previously deployed model. Alternative strategies are being evaluated including incremental learning versions of each algorithm as well as strategies based upon model comparisons, i.e. comparing performance of two models to either correlate their outputs or as a means of deciding which model to use at

run-time. The significant issues that are not yet well understood involve the tradeoff between efficiency and accuracy. Ideally, the sensor, the learning system and the detector should not require an inordinate amount of system resources. As the system now stands, no model has grown beyond 200 MB's of data, and the load is under 5%.

With respect to self-calibration, the current approach that seems most sensible is to measure the distribution of model scores of the training data and to select a threshold that admits a small percentage of alerts over that data. Hence, a user specified percentage (say .1%) may be used as an initial starting point to allow the threshold setting to be automatically adjusted. The operational impact of this process is part of our ongoing study on anomaly detection systems. What we do not yet understand is whether a threshold setting for RAD should remain static or whether it should be self adjusted at run time to account for the dynamics of the environment and how it may shift.

7.4 Mimicry Attack

It is entirely possible that a system such as RAD may be thwarted by malicious code that avoids any access to the Registry, or that may mimic a normal Registry query. The mere fact that each Windows environment is shipped to many millions of users implies that there are many potential vulnerable systems with exactly the same standard environments. (Each such system has a standard Windows and System32 directory with all of the key OS DLL's and executables targeted by malicious code.) Hence, crafty attackers may study this common information and architect malicious code to behave like other normal Windows processes avoiding detection altogether by a mimicry attack. Other have been studying this issue as well and have posited the concept of *diversity*; essentially to make each machine, platform and environment as unique and distinct as possible so that common behaviors are not so common and are not trivially mimicked.

7.5 Self-protection

Any host-based security system is subject to insider attack in a variety of ways. One such inside attack is to stealthily alter the anomaly detection models rendering the sensor useless. (Of course other brute force methods may be more effective, such as killing detector processes, yet these actions may also may be easily detectable.)

We also plan to store the system Registry behavior model as part of the Registry itself. The motivation behind this is to use the anomaly detector to protect the system behavior model from being maliciously altered, hence making the model itself secured against attack. These additions to the RAD system will make the system a more complete and effective tool for detecting malicious behavior on the Windows platform.

8 Conclusion

By using Registry activity on a Windows system, we were able to label all processes as either attacks or normal, with relatively high accuracy and low false positive rate, for the experiments performed in this study. We have shown that Registry activity is regular, and described ways in which attacks would generate anomalies in the Registry. Thus, an anomaly detector for Registry data may be an effective component in an intrusion detection system augmenting other host-based detection systems. It would also improve protection of systems in cases of new attacks that would otherwise pass by scanners that have not been updated on a timely basis.

In the comparative evaluation of our OCSVM algorithm and our PAD algorithm, we have shown that the PAD algorithm is more reliable, with substantially better computational complexity. The PAD algorithm perfectly classifies all true anomalies while mislabeling only 5% of normal data; whereas the OCSVM will misclassify nearly 40% of normal data before perfectly detecting true anomalies. Also, the OCSVM algorithm has time complexity $O(dL^3)$, where d is the number of dimensions and L is the number of records in the training dataset, and space complexity $O(d(L+T))$, where T is the number of records in the test dataset. PAD has time complexity $O(v^2R^2)$, where v is the number of unique record values for each record component and R is the number of record components, and space complexity $O(vR^2)$, making PAD a much more efficient algorithm as well. The OCSVM system needs a fair amount of improvement before it is competetive with PAD. However, understanding the reasons for PAD's higher classification accuracy will lead to an improvement of the OCSVM learning algorithm and will expedite the future development of anomaly detectors using the SVM framework. Since there is currently no effective way to learn a most optimal kernel for a given dataset, we must rely on our domain knowledge in order to develop a kernel that leads to a highly accurate anomaly detection system. PAD only requires an estimate of the possible range of values of each feature sampled from the data stream. By analyzing algorithms such as PAD which currently discriminate well, we can identify information which is important to capture in our data representation and is crucial for the development of a more optimal kernel for the SVM framework.

Acknowledgements

The work reported in this paper was supported by a DARPA contract No:F30602-02-2-0209. We thank the anonymous reviewers for their careful reading of an earlier draft that substantially improved the final paper.

References

1. Aim Recovery. `http://www.dark-e.com/des/software/aim/index.shtml`.
2. Back Orifice. `http://www.cultdeadcow.com/tools/bo.html`.

3. BackDoor.XTCP.
http://www.ntsecurity.new/Panda/Index.cfm?FuseAction=Virus&VirusID=659.
4. BrowseList.
http://e4gle.org/files/nttools/,http://binaries.faq.net.pl/security_tools.
5. Happy 99. http://www.symantex.com/qvcenter/venc/data/happy99.worm.html.
6. IPCrack.
http://www.geocities.com/SiliconValley/Garage/3755/toolicq.html,
http://home.swipenet.se/~w-65048/hacks.htm.
7. L0pht Crack. http://www.atstack.com/research/lc.
8. Setup Trojan. http://www.nwinternet.com/~pchelp/bo/setuptrojan.txt.
9. Shlomo Hershkop, Ryan Ferster, Linh H. Bui, Ke Wang and Salvatore J. Stolfo. Host-based Anomaly Detection Using Wrapping File Systems. CU Tech Report April 2004
10. Derek Armstrong, Sam Carter, Gregory Frazier and Tiffany Frazier. A controller-based autonomic defense system. Proc. DARPA Information Survivability Conference and Exposition DISCEX III, 2003.
11. F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.
12. Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy and Salvatore Stolfo. A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. Data Mining for Security Applications. Kluwer 2002.
13. EE, Kymie MC Tan, Roy A. Maxion. Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. IEEE Symposium on Security and Privacy, 2002.
14. A. Arnold. SVM anomaly detection C code. *IDS Lab, Columbia University*, 2002.
15. V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 1994.
16. Fred Cohen. *A Short Course on Computer Viruses*. ASP Press, Pittsburgh, PA, 1990.
17. M. H. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, New York, 1970.
18. D. E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, SE-13:222–232, 1987.
19. Eleazar Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.
20. Eleazar Eskin. Probabilistic anomaly detection over discrete records using inconsistency checks. Technical report, Columbia University Computer Science Technical Report, 2002.
21. Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. pages 120–128. IEEE Computer Society, 1996.
22. N. Friedman and Y. Singer. Efficient bayesian parameter estimation in large discrete domains, 1999.
23. K. Heller, K. Svore, A. Keromytis, and S. Stolfo. One class support vector machines for detecting anomalous windows registry accesses. *Proceedings of the Data Mining for Computer Security Workshop at IEEE ICDM 2003*, 2003.
24. S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji. Intrusion detect using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

25. Andrew Honig, Andrew Howard, Eleazar Eskin, and Salvatore Stolfo. Adaptive model generation: : An architecture for the deployment of data minig-based intrusion detection systems. In *Data Mining for Security Applications*. Kluwer, 2002.
26. Internet Engineering Task Force. Intrusion detection exchange format. In *http://www.ietf.org/html.charters/idwg-charter.html*, 2000.
27. H. S. Javitz and A. Valdes. The nides statistical component: Description and justification. Technical report, SRI International, 1993.
28. W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix processes execution traces for intrusion detection. pages 50–56. AAAI Press, 1997.
29. W. Lee, S. J. Stolfo, and K. Mok. Data mining in work flow environments: Experiences in intrusion detection. In *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999.
30. Wenke Lee, Sal Stolfo, and Kui Mok. A data mining framework for building intrusion detection models. 1999.
31. MacAfee. Homepage - macafee.com. *Online publication*, 2000. http://www.mcafee.com.
32. M. Mahoney and P. Chan. Detecting novel attacks by identifying anomalous network packet headers. Technical Report CS-2001-2, Florida Institute of Technology, Melbourne, FL, 2001.
33. C.C. Michael and Anup Ghosh. Simple, state-based approaches to program-based anomaly detection. ACM Trans. on Information and System Security (TISSEC), Vol 5, Issue 3, 2002.
34. C. Taylor and J. Alves-Foss. NATE - Network Analysis of Anomalous Traffic Events, A Low-Cost approach. New Security Paradigms Workshop, 2001.
35. B. Schölkopf, J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. Technical Report 99-87, Microsoft Research, 1999. To appear in *Neural Computation*, 2001.
36. SysInternals. Regmon for Windows NT/9x. *Online publication*, 2000. http://www.sysinternals.com/ntw2k/source/regmon.shtml.
37. Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: alternative data models. pages 133–145. IEEE Computer Society, 1999.
38. Steve R. White. Open problems in computer virus research. In *Virus Bulletin Conference*, 1998.
39. Giovanni Vigna, Fredrik Valeur and Richard Kemmerer. Designing and implementing a family of intrusion detecton systems. Proc. 9th European Software Engineering Conference, Finland, 2003.
40. Roy Maxion, and Tahlia Townsend. Masquerade Detection Using Truncated Command Lines. International Conference on Dependable Systems and Networks (DSN-02), 2002.
41. J. Hoagland. SPADE. Silican Defense, http://www.silicondefense.com/software/spice, 2000.

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
LOADWC.EXE[2]	1	8.497072	8.497072	ATTACK
ipccrack.exe[6]	1	8.497072	8.497072	ATTACK
mstinit.exe[2]	11	7.253687	6.705313	ATTACK
bo2kss.exe[2]	12	7.253687	6.62527	ATTACK
runonce.exe[2]	8	7.253384	6.992995	ATTACK
browselist.exe[4]	32	6.807137	5.693712	ATTACK
install.exe[3]	18	6.519455	6.24578	ATTACK
SetupTrojan.exe[8]	30	6.444089	5.756232	ATTACK
AimRecover.exe[1]	61	6.444089	5.063085	ATTACK
happy99.exe[5]	29	5.918383	5.789022	ATTACK
bo2k_1_0_intl.e[2]	78	5.432488	4.820771	ATTACK
_INS0432._MP[2]	443	5.284697	3.094395	ATTACK
xtcp.exe[3]	240	5.265434	3.705422	ATTACK
bo2kcfg.exe[2]	289	4.879232	3.520338	ATTACK
l0phtcrack.exe[7]	100	4.688737	4.575099	ATTACK
Patch.exe[2]	174	4.661701	4.025433	ATTACK
bo2k.exe[2]	883	4.386504	2.405762	ATTACK
systray.exe	17	7.253687	6.299848	NORMAL
CSRSS.EXE	63	7.253687	5.031336	NORMAL
SPOOLSS.EXE	72	7.070537	5.133161	NORMAL
ttssh.exe	12	6.62527	6.62527	NORMAL
winmine.exe	21	6.56054	6.099177	NORMAL
em_exec.exe	29	6.337396	5.789022	NORMAL
winampa.exe	547	6.11399	2.883944	NORMAL
PINBALL.EXE	240	5.898464	3.705422	NORMAL
LSASS.EXE	2299	5.432488	1.449555	NORMAL
PING.EXE	50	5.345477	5.258394	NORMAL
EXCEL.EXE	1782	5.284697	1.704167	NORMAL
WINLOGON.EXE	399	5.191326	3.198755	NORMAL
rundll32.exe	142	5.057795	4.227375	NORMAL
explore.exe	108	4.960194	4.498871	NORMAL
netscape.exe	11252	4.828566	-0.138171	NORMAL
java.exe	42	4.828566	3.774119	NORMAL
aim.exe	1702	4.828566	1.750073	NORMAL
findfast.exe	176	4.679733	4.01407	NORMAL
TASKMGR.EXE	99	4.650997	4.585049	NORMAL
MSACCESS.EXE	2825	4.629494	1.243602	NORMAL
IEXPLORE.EXE	194274	4.628190	-3.419214	NORMAL
NTVDM.EXE	271	4.59155	3.584417	NORMAL
CMD.EXE	116	4.579538	4.428045	NORMAL
WINWORD.EXE	1541	4.457119	1.7081	NORMAL
EXPLORER.EXE	53894	4.31774	-1.704574	NORMAL
msmsgs.exe	7016	4.177509	0.334128	NORMAL
OSA9.EXE	705	4.163361	2.584921	NORMAL
MYCOME 1.EXE	1193	4.035649	2.105155	NORMAL
wscript.exe	527	3.883216	2.921123	NORMAL
WINZIP32.EXE	3043	3.883216	0.593845	NORMAL
notepad.exe	2673	3.883216	1.264339	NORMAL
POWERPNT.EXE	617	3.501078	-0.145078	NORMAL
AcroRd32.exe	1598	3.412895	0.393729	NORMAL
MDM.EXE	1825	3.231236	1.680336	NORMAL
ttermpro.exe	1639	2.899837	1.787768	NORMAL
SERVICES.EXE	1070	2.576196	2.213871	NORMAL
REGMON.EXE	259	2.556836	1.205416	NORMAL
RPCSS.EXE	4349	2.250997	0.812288	NORMAL

Table 2. Information about all processes in testing data for the PAD algorithm including the number of Registry accesses and the maximum and minimum score for each record as well as the classification. The top part of the table shows this information for all of the attack processes and the bottom part of the table shows this information for the normal processes. The reference number (by the attack processes) give the source for the attack. Processes that have the same reference number are part of the same attack. [1] AIMCrack. [2] Back Orifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.

Threshold Score	False Positive Rate (%)	Detection Rate (%)
6.847393	0.1192	0.5870
6.165698	0.2826	2.7215
5.971925	0.3159	3.0416
5.432488	0.4294	6.4034
4.828566	0.5613	9.9253
4.565011	0.6506	17.7161
3.812506	0.9343	28.8687
3.774119	0.9738	31.4301
3.502904	1.1392	53.3084
3.231236	1.2790	53.5219
3.158004	1.4740	57.7908
2.915094	1.9998	57.8442
2.899837	2.0087	62.7001
2.753176	3.3658	62.9136
2.584921	3.4744	80.8431
2.531572	3.8042	86.9797
2.384402	5.0454	100.0000

Table 3. Varying the threshold score for the PAD algorithm and its effect on False Positive Rate and Detection Rate.

Threshold Score	False Positive Rate (%)	Detection Rate (%)
-1.08307	0.790142	0.373533
-1.08233	0.828005	0.480256
-1.07139	1.54441	0.533618
-0.968913	1.65734	1.17396
-0.798767	3.58736	3.89541
-0.79858	3.63784	5.60299
-0.798347	3.68999	6.77695
-0.767411	3.72054	6.83031
-0.746663	4.35691	7.47065
-0.746616	4.63025	8.00427
-0.71255	8.34283	20.9712
-0.712503	8.75201	22.0918

Table 4. The effects of varying the threshold for the OCSVM algorithm on False Positive Rate and Detection Rate.

Program Name	Classification	Number of Records	Minimum Record Value	Maximum Record Value
REGMON.EXE	NORMAL	259	-0.794953	-0.280406
SPOOLSS.EXE	NORMAL	72	-1.152717	-0.021361
CloseKey	NORMAL	429	-1.082720	-0.374784
OpenKey	NORMAL	502	-0.959895	-0.365539
QueryValue	NORMAL	594	-1.082909	-0.374972
EnumerateValue	NORMAL	28	-0.570206	-0.284935
DeleteValueKey	NORMAL	3	-1.078758	-0.370822
AimRecover.exe	NORMAL	61	-1.082720	-0.374784
aim.exe	NORMAL	1702	-1.064796	-0.356860
ttssh.exe	NORMAL	12	-0.969706	-0.375161
ttermpro.exe	NORMAL	1639	-1.083098	-0.285123
NTVDM.EXE	NORMAL	271	-0.798204	-0.410065
notepad.exe	NORMAL	2673	-1.083098	-0.285123
CMD.EXE	NORMAL	116	-1.139322	-0.375161
TASKMGR.EXE	NORMAL	99	-0.570017	-0.284935
_INS0432._MP	NORMAL	443	-1.423272	-1.423272
WINLOGON.EXE	NORMAL	399	-1.423272	-1.423272
systray.exe	NORMAL	17	-1.423272	-1.423272
em_exec.exe	NORMAL	29	-1.423272	-1.423272
OSA9.EXE	NORMAL	705	-1.083098	-0.375161
findfast.exe	NORMAL	176	-1.083098	-0.375161
WINWORD.EXE	NORMAL	1541	-1.083098	-0.375161
winmine.exe	NORMAL	21	-0.429351	-0.429351
POWERPNT.EXE	NORMAL	617	-1.083098	-0.285123
PING.EXE	NORMAL	50	-1.083098	-0.375161
QueryKey	NORMAL	11	-0.712317	-0.375161
wscript.exe	NORMAL	527	-1.083098	-0.375161
AcroRd32.exe	NORMAL	1598	-1.083098	-0.375161
0”	NORMAL	404	-1.083098	-0.375161
WINZIP32.EXE	NORMAL	3043	-1.083098	-0.375161
explore.exe	NORMAL	108	-1.083098	-0.375161
EXCEL.EXE	NORMAL	1782	-1.083098	-0.375161
bo2kss.exe[2]	ATTACK	12	-0.712317	-0.375161
bo2k_1_0_intl.e[2]	ATTACK	78	-1.083098	-0.375161
browselist.exe[4]	ATTACK	32	-0.798770	-0.411763
bo2kcfg.exe[2]	ATTACK	289	-1.423272	-1.423272
bo2k.exe[2]	ATTACK	883	-1.423272	-1.091776
mstinit.exe[2]	ATTACK	11	-1.423272	-1.423272
runonce.exe[2]	ATTACK	8	-1.423272	-1.423272
Patch.exe[2]	ATTACK	174	-1.083098	-0.375161
install.exe[3]	ATTACK	18	-1.083098	-0.375161
xtcp.exe[3]	ATTACK	240	-1.083098	-0.285123
l0phtcrack.exe[7]	ATTACK	100	-0.798581	-0.285123
LOADWC.EXE[2]	ATTACK	1	-1.423272	-1.423272
happy99.exe[5]	ATTACK	29	-0.570017	-0.411575

Table 5. Information about test records using the OCSVM algorithm with a linear kernel and binary feature vectors. The maximum and minimum discriminants are given for each process, as well as the assigned classification label. Listed next to the attack processes is the attack source. [1] AIMCrack. [2] BackOrifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.

Program Name	Classification	Number of Records	Minimum Record Value	Maximum Record Value
REGMON.EXE	NORMAL	259	-4.062785	-1.524777
SPOOLSS.EXE	NORMAL	72	-5.422540	-0.272565
CloseKey	NORMAL	429	-5.210662	-1.788163
OpenKey	NORMAL	502	-4.828603	-1.758730
QueryValue	NORMAL	594	-5.211228	-1.789106
EnumerateValue	NORMAL	28	-3.311164	-1.542890
DeleteValueKey	NORMAL	3	-5.1955757	-1.766465
AimRecover.exe	NORMAL	61	-5.210285	-1.792879
aim.exe	NORMAL	1702	-5.148589	-1.703827
ttssh.exe	NORMAL	12	-4.860299	-1.794766
ttermpro.exe	NORMAL	1639	-5.211794	-1.543456
NTVDM.EXE	NORMAL	271	-4.234352	-1.794766
notepad.exe	NORMAL	2673	-5.211794	-1.543456
CMD.EXE	NORMAL	116	-5.388013	-1.794766
TASKMGR.EXE	NORMAL	99	-3.309843	-1.543456
_INS0432._MP	NORMAL	443	-6.239865	-6.239865
WINLOGON.EXE	NORMAL	399	-6.239865	-6.239865
systray.exe	NORMAL	17	-6.239865	-6.239865
em_exec.exe	NORMAL	29	-6.239865	-6.239865
OSA9.EXE	NORMAL	705	-5.211794	-1.789672
findfast.exe	NORMAL	176	-5.211794	-1.794766
WINWORD.EXE	NORMAL	1541	-5.211794	-1.789672
winmine.exe	NORMAL	21	-1.794766	-1.794766
POWERPNT.EXE	NORMAL	617	-5.211794	-1.543456
PING.EXE	NORMAL	50	-5.211794	-1.789672
QueryKey	NORMAL	11	-4.022096	-1.789672
wscript.exe	NORMAL	527	-5.211794	-1.789672
AcroRd32.exe	NORMAL	1598	-5.211794	-1.794766
0”	NORMAL	404	-5.211794	-1.789672
WINZIP32.EXE	NORMAL	3043	-5.211794	-1.789672
explore.exe	NORMAL	108	-5.211794	-1.789672
EXCEL.EXE	NORMAL	1782	-5.211794	-1.789672
bo2kss.exe[2]	ATTACK	12	-4.022096	-1.789672
bo2k_1_0_intl.e[2]	ATTACK	78	-5.211794	-1.789672
browselist.exe[4]	ATTACK	32	-4.087124	-1.789672
bo2kcfg.exe[2]	ATTACK	289	-6.239865	-6.239865
bo2k.exe[2]	ATTACK	883	-6.239865	-5.245378
mstinit.exe[2]	ATTACK	11	-6.239865	-6.239865
runonce.exe[2]	ATTACK	8	-6.239865	-6.239865
Patch.exe[2]	ATTACK	174	-5.211794	-1.789672
install.exe[2]	ATTACK	18	-5.211794	-1.794766
xtcp.exe[3]	ATTACK	240	-5.211794	-1.543456
l0phtcrack.exe[7]	ATTACK	100	-4.194165	-1.543456
LOADWC.EXE[2]	ATTACK	1	-6.239865	-6.239865
happy99.exe[5]	ATTACK	29	-3.309843	-1.794766

Table 6. Information about test records using the OCSVM algorithm with a second order polynomial kernel and binary feature vectors. The maximum and minimum discriminants are given, as well as the assigned classification label. Listed next to the attack processes is the attack source. [1] AIMCrack. [2] BackOrifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] L0pht Crack. [8] Setup Trojan.