

# Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses

Sal 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 host-based intrusion detection system (IDS) for Microsoft Windows. The core of the system is an algorithm that detects attacks on a host machine by looking for anomalous accesses to the Windows Registry. We present two anomaly detection algorithms for use in our IDS system and evaluate their performance. The key idea is to first train a model of normal registry behavior on a windows host, and use this model to detect abnormal registry accesses at run-time. The normal model is trained using clean (attack-free) data. 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.

## 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 systems 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 [?, ?]. 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 [?]. 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 the ultimate distribution to the vulnerable systems. Since malicious software may propagate through email, often the malicious software can reach the 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 than distinct signatures of that software. Unfortunately, the commercial IDS systems that are widely in use 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 data mining techniques such as anomaly detection, in IDS systems [?, ?]. Anomaly detection algorithms build models of normal behavior in order to detect behavior that deviates from normal behavior and which may correspond to an attack [?, ?]. The main advantage of anomaly detection is that it can detect new attacks and can be an effective defense against new malicious software. Anomaly detection algorithms have been applied to network intrusion detection [?, ?, ?] and also to the analysis of system calls for host based intrusion detection [?, ?, ?, ?, ?]. There are two problems to the system call approach to host based IDS which inhibits their use in actual deployment. The first is that the computational overhead of monitoring all system calls is very high, which degrades the performance of a system. The second is that system calls themselves are irregular by nature, which makes it difficult to differentiate between normal and malicious behaviors, which may cause a high false positive rate.

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

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 activity that corresponds to malicious software. The main advantages of monitoring the Windows Registry is that the activity is regular by nature, can be monitored with low computational overhead, and almost all system activities interact with the registry.

We present two new anomaly detection algorithms. The first anomaly detection algorithm, PAD (Probabilistic Anomaly Detection), is a registry-specific version of PHAD (Packet Header Anomaly Detection), an anomaly detection algorithm originally presented to detect anomalies in packet headers [?]. The second anomaly detection algorithm uses one class support vector machines (OCSVMs) to detect anomalous activity in the Windows registry. We show that the data

generated by a registry sensor is useful in detecting malicious behavior. We shall 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 this data to detect abnormal registry behavior which corresponds to the actions of malicious software. By showing the results of our experiments and detailing how various malicious activities use the registry, we show that the registry is a good source of data for intrusion detection. 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.

We present results of experiments evaluating the RAD system and demonstrate that it is effective in detecting attacks while maintaining a low rate of false alarms. 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 [?]. 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.

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 is used by malicious software. Section 4 fully develops the probabilistic anomaly detection algorithm and the One-class support vector machine algorithm employed in our studies of registry 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 to 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 anomaly detection system applied to Windows Registry data to detect 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 (for example, the file system anomaly detection system we call FWRAPS [15], and the Malicious Email Tracking systems MET [16].)

Anomaly detection systems were first proposed by Denning [1] for network-based intrusion detection, and later implemented in NIDES [2] 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. describe a framework and system for auditing and data mining and feature selection for intrusion detection. This framework consists of classification, link analysis and sequence analysis for constructing intrusion detection models [3,4] 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, applied to a variety of different audit sources, some specific to user commands for masquerade detection [5], others specific to network protocols and LAN traffic for detecting denial of service attacks [24, 34] or Trojan execution, or application or system call-level data for malware detection [6, 7], 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 [14]. These include estimating probabilistic or statistical distributions over temporal data [8, 9], supervised machine learning [10, 11] and unsupervised cluster-based algorithms [12]. Some approaches consider the correlation of multiple models [13].

The work reported in this paper is, to the best of our knowledge, the first sensor devoted to anomaly detection for the Windows platform. There are several other host-based intrusion detection and prevention systems [17] primarily focused on misuse detection driven by signature- or specification-based techniques. The RAD system is intended to be integrated with other host-based security systems to enrich their coverage and to leverage their (bad) behavior-blocking capabilities, while providing 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. 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. Rather than choosing one or the other, 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 Windows programs. The Windows Registry is also the source for all security information: policies, user names, and passwords. The registry also stores much of the important run-time configuration information that programs need to run.

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 to monitor attacks because many attacks show up as 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, notwithstanding the fact that other programs can arbitrarily access the information. Although some versions of Windows include security permissions and Registry logging, both features are rarely used (because of the computational overhead and the complexity of the configuration options).

### 3.2 Analysis of Malicious Registry Accesses

Most Windows programs access a certain set of Registry keys during normal execution. Furthermore, most users use a certain set of programs routinely while running their machines. This may be a set of all programs installed on the machine or more typically 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 the normal registry behavior is computed over clean data, then these kinds of registry operations will not appear in 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 and how they produce anomalous registry activity are described below.

- **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 creates typically 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 determine it as anomalous. This program also launches many other programs (`LoadWC.exe`, `Patch.exe`, `runonce.exe`, `bo2k_1_o_int1.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's actually a very simple program that simply 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 usually is accessed by a different program that created that key.

- **Disable Norton:** This is a very simple exploitation of the registry that disables Norton Antivirus. This attack toggles one record in the registry, the key `HKLM\SOFTWARE\INTEL \LANDesk \VirusProtect6\CurrentVersion \Storagees \Files\System \RealTimeScan \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. It retrieves the hashed SAM file containing the passwords for all users and then uses either a dictionary or brute force approach to find the passwords. This program also uses flaws in the Windows encryption scheme which allows the program to discover some of the characters in the password. This program uses the registry by creating its own section in the registry. This will consist of 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, 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 attack by the system. The installation of a new program on a system will be viewed as anomalous activity. Programs often create new sections of the registry and many new keys on installation. This will 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. 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. The algorithm could be modified to ignore alarms while the install shield program was running because that would mean that the user is aware that a new program is being installed. Another option

is to simply prompt the user when a detection occurs so that the user can let the anomaly detection system know that a legitimate installed program is under way and that 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.

## 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 each 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, a set of five features are extracted from each registry access. Using these feature values over normal data, a model of normal registry behavior is generated. This model of normalcy consists of a set of consistency checks applied to the features. When detecting anomalies, the model of normalcy determines whether the values in features of the current registry access are consistent with the normal data or not. If new activity is 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	" BCOFHIHBAHF"	" BCOFHIHBAHF"

**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. The final column shows which fields register as anomalous. Note that the pairs of features must be 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 generates an anomaly.

## 4.2 PAD Anomaly Detection Algorithm

Using the features that we monitor from each registry access, we train a model over features extracted from normal data. That model allows us to classify registry accesses as either normal or malicious.

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. We apply a probabilistic algorithm described in Eskin, 2002 [?] and here we provide a short summary of the algorithm. The algorithm is similar to the heuristic algorithm that was proposed by Chan and Mahoney in the PHAD system [?], but is more robust.

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 [?] and the references therein.) In our setting, part of the problem is that each of the features have 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 consistency check eval-

uates 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.

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 a virtual count to each possible element. This is equivalent to using a Dirichlet estimator [?]. For anomaly detection, as pointed out in Mahoney and Chan, 2001 [?], 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.

To estimate our likelihoods we use the estimator presented in Friedman and Singer, 1999 [?] 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.  $\alpha$  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

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

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.

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 [?] and complete details of the anomaly detection algorithm are given in [?].

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 will be represented by large numbers of records where many of those records will be considered anomalous.

Some records are anomalous because they have a value for a feature that is inconsistent with the normal data. However, some records are anomalous because they have an inconsistent combination of features although each feature itself may be normal. Because of this, we examine pairs of features. For example, let us consider the registry access displayed in Table 1. The basic features for the normal program `aim.exe` versus the malicious program `aimrecover.exe` do not appear anomalous. However, the fact that the program `aimrecover.exe` is accessing a key that is usually associated with `aim.exe` is in fact an anomaly. Only by examining the combination of the two raw features can we detect this anomaly.

The PAD algorithm takes time  $O(v^2 R^2)$ , 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  $O(v R^2)$ .

### 4.3 OCSVM Anomaly Detection Algorithm

As an alternative to the PAD algorithm for model generation and anomaly detection, we apply an algorithm based on the one class SVM algorithm given in [?]. Previously, OCSVMs have not been used in Host-based anomaly detection systems. The OCSVM code was developed by [?] and 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) and iteratively finds the maximal margin hyperplane which best separates the training data from the origin. The OCSVM may be viewed as a regular two-class 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:

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

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  $f(\mathbf{x}) < 0$  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:

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

subject to the constraints

$$0 \leq \alpha_i \leq \frac{1}{\nu l} \quad (6)$$

and

$$\sum_i \alpha_i = 1 \quad (7)$$

where  $\alpha_i$  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),  $\nu$  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. Given a feature map:

$$\phi : X \rightarrow R^N \quad (8)$$

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

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \quad (9)$$

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 kernel:  $K(x, y) = e^{-\|x-y\|^2/(2\sigma^2)}$ , where  $\sigma^2$  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).

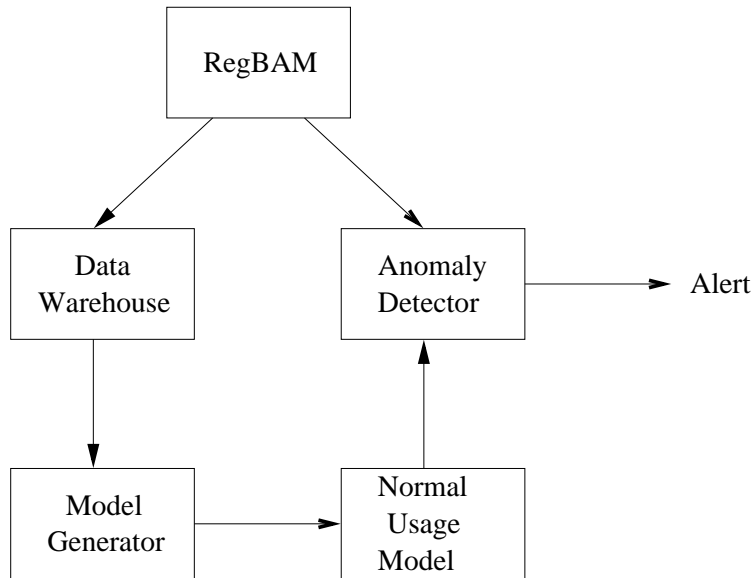
## 5 Architecture

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.

### 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. BAMs implement an architecture and interface for sensors across the system. They include a hook into the audit stream (in this case the registry) and various communication and data-buffering components. BAMs use an XML data representation similar to the IDMEF standard (of the IETF) for IDS systems [?]. BAMs are described in more detail in [?].

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 tap into the registry 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 [?]. After gathering the registry data, RegBAM can be configured for two distinct purposes. One use is 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



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

[?]. 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 sent directly to the anomaly detector where it is processed in real time. An alternative method to collect the registry accesses is to use the Windows auditing mechanism. All registry accesses can be logged in the Windows Event Log. Each read or write can generate multiple records in the Event Log. However, this method is problematic 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.

## 5.2 Model Generation Infrastructure

Similar to the Adaptive Model Generation (AMG) architecture [?], the system uses RegBAM to collect registry access records. Using this database of collected records from a training run, the model generator then creates a model of normal usage.

The model generator uses the algorithm discussed in Section 4 to build a model that represents normal usage. It utilizes the data stored in the database which was generated by RegBAM during training. The model itself is comprised and stored as serialized Java objects. This allows for a single model to be gener-

ated and to be easily distributed to additional machines. Having the model easily deployed to new machines is a desirable feature, since in a typical network, many Windows machines have similar usage patterns. This allows the same model to be used for multiple host machines.

### **5.3 Real-Time Anomaly Detector**

For real time detection, RegBAM feeds live data for analysis by an 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 by a user 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. 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 increase the chance of detecting new attacks.

### **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 50,000 records per hour. Our distributed architecture is designed to minimize the resources used by the host machine. It is possible to spread the work load on to several separate machines, so that the only application running on the host machine is the lightweight RegBAM. However this will increase network load due to the communication between components. These two concerns can be used to configure the system 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.

## **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.

## 6.1 Data Generation

In order to evaluate the RAD system, we gathered data by running a registry sensor on a host machine. Since there are no publicly available data sets containing registry accesses, we collected our own data. Beyond the normal 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 simulations were done by hand to simulate 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 training data collected 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, browslist, bok2ss (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 4.

The reason for running some of the attacks twice, was to test the effectiveness of our system. 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 irregular. 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 first executed.

## 6.2 Experiments

The training and testing environments were set up to replicate a simple yet realistic model of usage of Windows systems. The system load and the applications that were run were meant to resemble what one may deem typical in normal private settings.

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 normal 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. Fig 2 shows how varying the threshold affects the output of the detector. The actual recorded scores for the PAD algorithm plotted in the figure are displayed in Table 2. 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 4 is sorted in order to show the results for classifying processes. From the table we can see 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

Threshold Score	False Positive Rate	Detection Rate
6.847393	0.001192	0.005870
6.165698	0.002826	0.027215
5.971925	0.003159	0.030416
5.432488	0.004294	0.064034
4.828566	0.005613	0.099253
4.565011	0.006506	0.177161
3.812506	0.009343	0.288687
3.774119	0.009738	0.314301
3.502904	0.011392	0.533084
3.231236	0.012790	0.535219
3.158004	0.014740	0.577908
2.915094	0.019998	0.578442
2.899837	0.020087	0.627001
2.753176	0.033658	0.629136
2.584921	0.034744	0.808431
2.531572	0.038042	0.869797
2.384402	0.050454	1.000000

**Table 2.** Varying the threshold score and its effect on False Positive Rate and Detection Rate.

Threshold	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 3.** The effects of varying the threshold on the false positive rate and the detection rate.

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`, `SP00LSS.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. As we have mentioned, our

future work on RAD will model and measure a Windows system for a far longer period of time over many more processes in order to generate a meaningful ROC curve in terms of processes. The measurements reported next are cast in terms of registry query records.

### 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 2.

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 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 one class SVM 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 classification.

Many of the false positives were from processes that were simply not run 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 demonstrated in [?, ?]. The PAD system 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 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 occur unless a novel technique which incorporates more discriminative information is 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.

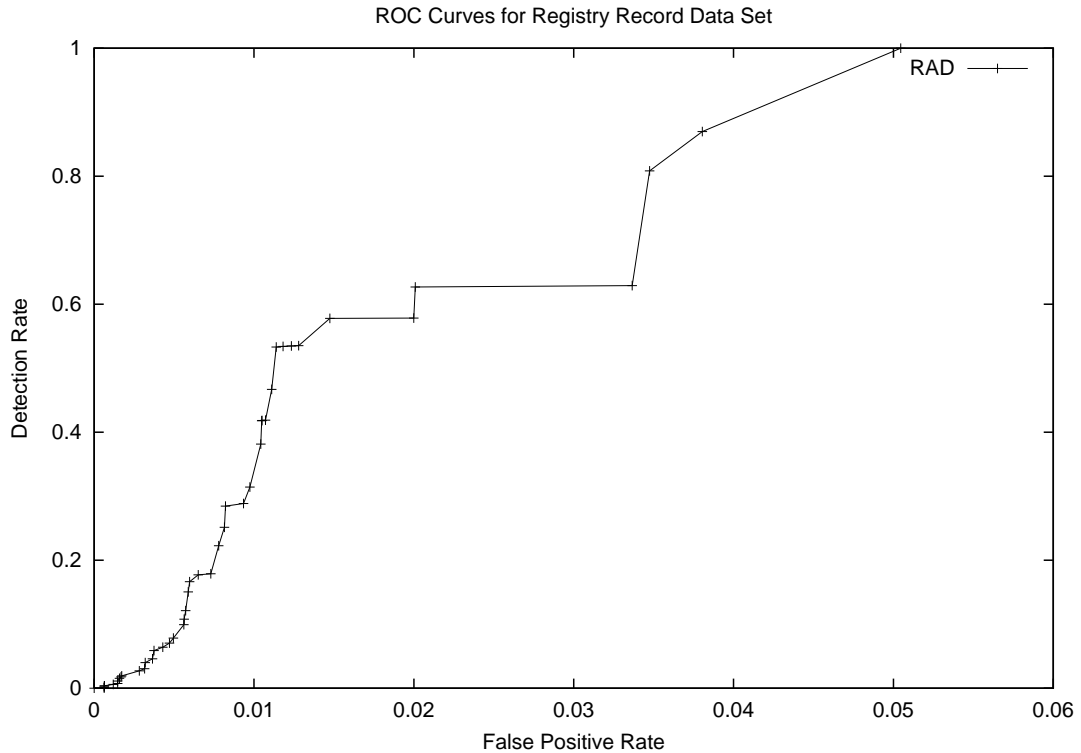
The advantage of the PAD algorithm over the OCSVM system lies in the use of a hierarchical prior to estimate probabilities. A scaling factor (see equation (4)) is computed and applied to a Dirichlet prediction which assumes that all possible elements have been seen, giving varying probability mass to outcomes unseen in the training set. In general, knowing the likelihood of encountering a previously unencountered 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 with our OCSVM system.

## 7 Discussion and Future Work

The RAD system has been presented as an intrusion detection system that signals an alert if and when anomalous registry accesses are detected. The system 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. 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 causing an incorrect RAD action. Of course the rate or frequency of user challenges may render RAD a frustratingly annoying security system if it generates too many 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

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
LOADWC.EXE[?]	1	8.497072	8.497072	ATTACK
ipccrack.exe[?]	1	8.497072	8.497072	ATTACK
mstinit.exe[?]	11	7.253687	6.705313	ATTACK
bo2kss.exe[?]	12	7.253687	6.62527	ATTACK
runonce.exe[?]	8	7.253384	6.992995	ATTACK
browselist.exe[?]	32	6.807137	5.693712	ATTACK
install.exe[?]	18	6.519455	6.24578	ATTACK
SetupTrojan.exe[?]	30	6.444089	5.756232	ATTACK
AimRecover.exe[?]	61	6.444089	5.063085	ATTACK
happy99.exe[?]	29	5.918383	5.789022	ATTACK
bo2k_1_0_intl.e[?]	78	5.432488	4.820771	ATTACK
_INS0432._MP[?]	443	5.284697	3.094395	ATTACK
xtcp.exe[?]	240	5.265434	3.705422	ATTACK
bo2kcfg.exe[?]	289	4.879232	3.520338	ATTACK
l0phtcrack.exe[?]	100	4.688737	4.575099	ATTACK
Patch.exe[?]	174	4.661701	4.025433	ATTACK
bo2k.exe[?]	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
mmsgs.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

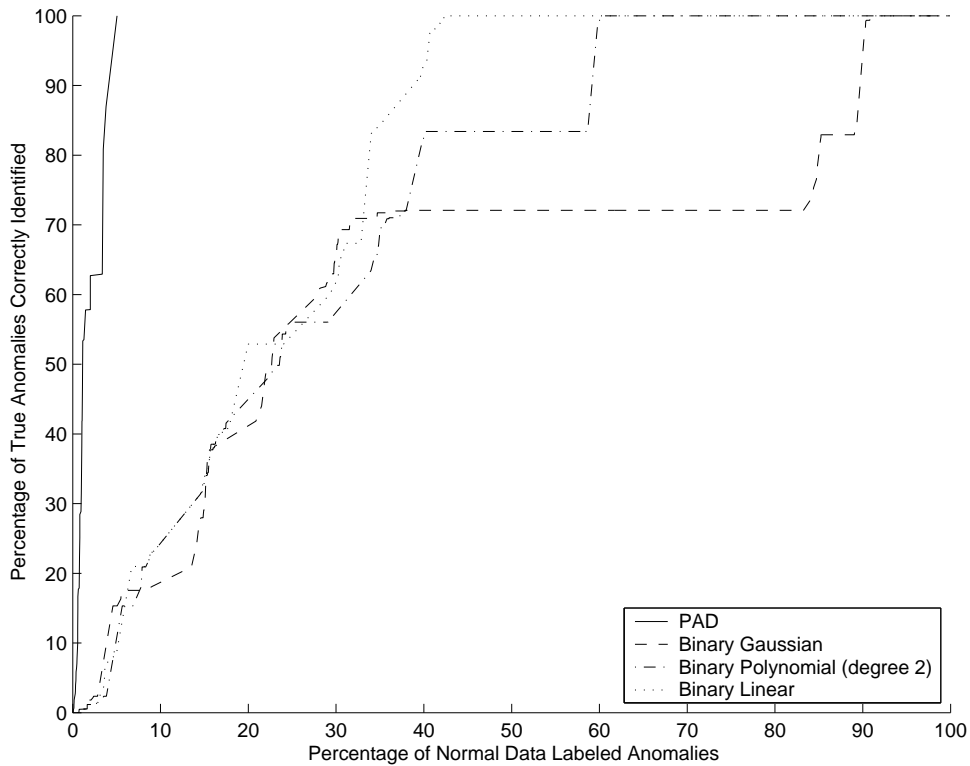
**Table 4.** Information about all processes in testing data including the number of registry accesses and the maximum and minimum score for each record as well as the classification. The top part of the table shows this information for all of the attack processes and the bottom part of the table shows 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.



**Fig. 2.** Figure showing varying the threshold on the data set.

windows asking whether to accept a cookie from a website, or to allow certain network connections from personal firewall applications.

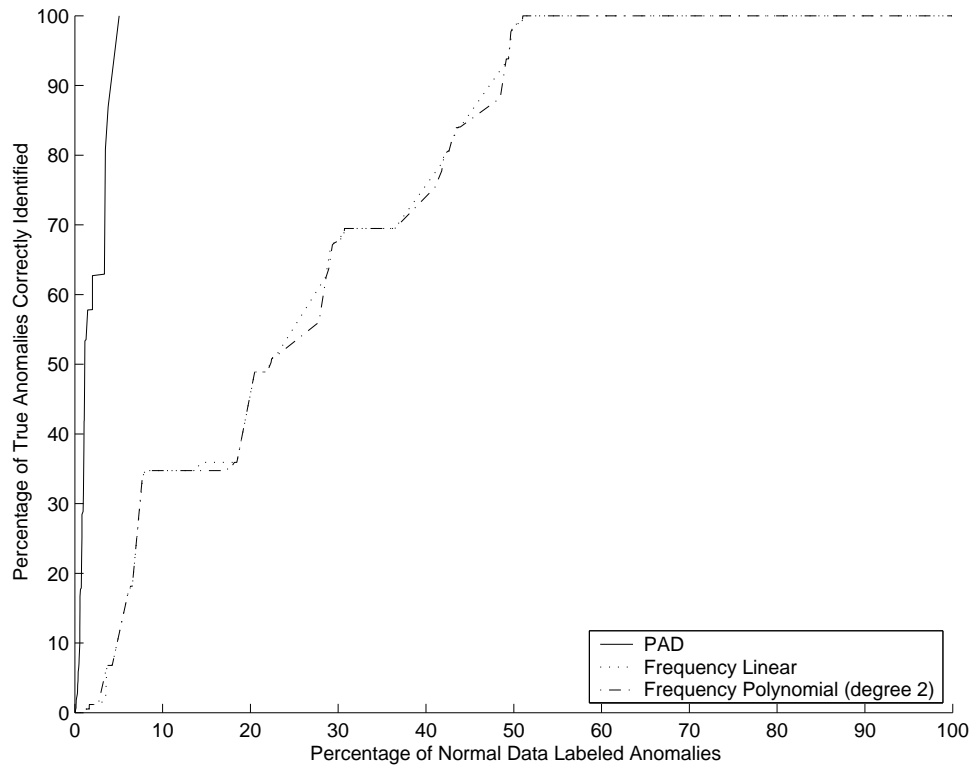
The experimental results reported were completed using a pristine Windows machine where the registry data was guaranteed to be "normal" data without embedded attacks. This approach is sensible if one presumes a standard Windows environment is shipped with a first "factory standard registry data model" pre-computed 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. 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



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

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.)

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 continuous 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



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

the tradeoff between efficiency and accuracy. Ideally, the sensor, the learning system and the detector should not require an inordinate amount of system resources.

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 a standard to allow the threshold setting to be automatically selected. 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.

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.) This means that if a virus or worm writer exploits some vulnerable and standard windows application or system component (as is commonly the case with fast spreading worms and viruses), that component is very likely stored in exactly the same standard location and easily found and targeted by malicious software without need to search the registry (or the file system) for its location. Hence, the malicious code can easily by-pass accessing registry information to find its quarry.

This suggests a general strategy to increase security of all systems, essentially to make each machine as unique and distinct as possible so that replicated vulnerabilities are not trivially exploited. This notion of "diversity" has been noticed by others and is an active area of research. The point here for a Windows environment suggests that each shipped Windows box should have a unique Registry with unique keys and file structures forcing a worm or virus attack to search the registry for the location of its target. Forcing this access to the Registry would logically have a higher chance of being detected by a system such as RAD prior to the malicious software reaching its target and performing its damage.

Although this strategy may be attractive to increase the security of each system, it is extremely unlikely this will be embraced. The economic advantage to standards enjoyed by third party software vendors and standard support features is too strong a force to allow "diversity" to succeed as a security strategy. However, the particular behavior of the user may provide sufficient uniqueness that may be modeled by an anomaly detection system such as RAD without resorting to a potentially complex strategy requiring each base platform to be distinct from all others. This latter approach, to model the user's specific behavior, is part of our ongoing studies on intrusion and anomaly detection systems.

## 8 Conclusions

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 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. However, understanding the reasons for this will lead to an improvement of the OCSVM system and will expedite the future development of anomaly detectors. 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. 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.

We plan on testing the system under a variety of environments and conditions to better understand its performance. Future plans include combining the RAD system with another detector that evaluates Windows Event Log data. This will allow for various data 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 attack rather than individual attacks. 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.

Finally, since most users accept the default installation location when installing a program, the location of programs tends to be the same on all computers. Thus an attack does not need to query the registry for program location information. By forcing a location declaration other than the default location, a given program will not have the same location on all Windows machines. Attacks will have to query the registry to discover program locations, thus forcing all attacks to be monitored by the anomaly detector. A system such as this would improve the anomaly detection capabilities of the RAD system since no malicious attacks can bypass querying the registry. This would enhance the protection of the system against malicious users.

## 9 Acknowledgements

The work reported in this paper was supported by a DARPA contract No:F30602-02-2-0209.

Program Name	Label	Number of Records	Min. Record Value	Max. 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
wscrip.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[?]	ATTACK	12	-0.712317	-0.375161
bo2k_1_0_intl.e[?]	ATTACK	78	-1.083098	-0.375161
browselist.exe[?]	ATTACK	32	-0.798770	-0.411763
bo2kcfg.exe[?]	ATTACK	289	-1.423272	-1.423272
bo2k.exe[?]	ATTACK	883	-1.423272	-1.091776
mstinit.exe[?]	ATTACK	11	-1.423272	-1.423272
runonce.exe[?]	ATTACK	8	-1.423272	-1.423272
Patch.exe[?]	ATTACK	174	-1.083098	-0.375161
install.exe[?]	ATTACK	18	-1.083098	-0.375161
xtcp.exe[?]	ATTACK	240	-1.083098	-0.285123
l0phtcrack.exe[?]	ATTACK	100	-0.798581	-0.285123
LOADWC.EXE[?]	ATTACK	1	-1.423272	-1.423272
happy99.exe[?]	ATTACK	29	-0.570017	-0.411575

**Table 5.** Information about test records for the linear kernel in the binary setting. 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. [?] AIMCrack. [?] BackOrifice. [?] Backdoor.xtcp. [?] Browse List. [?] Happy 99. [?] IPCrack. [?] L0pht Crack. [?] Setup Trojan.

Program Name	Label	Number of Records	Min. Record Value	Max. 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
_LNS0432._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[?]	ATTACK	12	-4.022096	-1.789672
bo2k_1_0_intl.e[?]	ATTACK	78	-5.211794	-1.789672
browselist.exe[?]	ATTACK	32	-4.087124	-1.789672
bo2kcfg.exe[?]	ATTACK	289	-6.239865	-6.239865
bo2k.exe[?]	ATTACK	883	-6.239865	-5.245378
mstinit.exe[?]	ATTACK	11	-6.239865	-6.239865
runonce.exe[?]	ATTACK	8	-6.239865	-6.239865
Patch.exe[?]	ATTACK	174	-5.211794	-1.789672
install.exe[?]	ATTACK	18	-5.211794	-1.794766
xtcp.exe[?]	ATTACK	240	-5.211794	-1.543456
l0phtcrack.exe[?]	ATTACK	100	-4.194165	-1.543456
LOADWC.EXE[?]	ATTACK	1	-6.239865	-6.239865
happy99.exe[?]	ATTACK	29	-3.309843	-1.794766

**Table 6.** Information about test records for the second order polynomial kernel in the binary setting. The maximum and minimum discriminants are given, as well as the assigned classification label. Listed next to the attack processes is the attack source. [?] AIMCrack. [?] BackOrifice. [?] Backdoor.xtcp. [?] Browse List. [?] Happy 99. [?] IPCrack. [?] L0pht Crack. [?] Setup Trojan.