

# Anomaly Detection in Computer Security and an Application to File System Accesses<sup>\*</sup>

Salvatore J. Stolfo, Shlomo Hershkop, Linh H. Bui, Ryan Ferster, and Ke Wang

Columbia University, New York, NY 10027, USA  
{sal, shlomo, lhb2001, rlf92, kewang}@cs.columbia.edu

**Abstract.** We present an overview of anomaly detection used in computer security, and provide a detailed example of a host-based Intrusion Detection System that monitors file systems to detect abnormal accesses. The File Wrapper Anomaly Detector (FWRAP) has two parts, a sensor that audits file systems, and an unsupervised machine learning system that computes normal models of those accesses. FWRAP employs the Probabilistic Anomaly Detection (PAD) algorithm previously reported in our work on Windows Registry Anomaly Detection. FWRAP represents a general approach to anomaly detection. The detector is first trained by operating the host computer for some amount of time and a model specific to the target machine is automatically computed by PAD. The model is then deployed to a real-time detector. In this paper we describe the feature set used to model file system accesses, and the performance results of a set of experiments using the sensor while attacking a Linux host with a variety of malware exploits. The PAD detector achieved impressive detection rates in some cases over 95% and about a 2% false positive rate when alarming on anomalous processes.

**Keywords:** Host-Based Intrusion Detection, Anomaly Detection, File System, Wrapping

## 1 Introduction

Widely used commercial Intrusion Detection Systems (IDS) are based on signature matching algorithms. These algorithms match audit data of network or 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 not effective on new attacks.

Anomaly Detection is an important alternative detection methodology that has the advantage of defending against new threats not detectable by signature-based systems. In general, anomaly detectors build a description of **normal** activity, by training a model of a system under typical operation, and compare

---

<sup>\*</sup> This work has been supported in part by a contract from DARPA, Application-layer IDS, Contract No. F30602-00-1-0603.

the normal model at run time to detect deviations of interest. Anomaly Detectors may be used over any audit source to both train and test for deviations from the norm. Anomaly detection algorithms may be specification-based or data mining or machine learning-based [11, 12], and have been applied to network intrusion detection [9, 11] and also to the analysis of system calls for host based intrusion detection [4, 6, 10, 23].

Specification or behavior based anomaly detectors such as the STAT approach [19], represent normal execution of a system using programmer supplied state transition or finite state machine representations [14]. 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.

Anomaly detection systems are more art than science. A number of design choices are necessary to build an effective detector. First, one must design a monitor or auditing sensor that is able to extract data from some system or component, and do so without wasting resources and minimizing the impact on the system being monitored.

One must also design a set of informative “features” that are extracted from the audit data that provides the means of computing an effective model able to distinguish attacks from normal execution. Subsequently, one must understand what “attacks” may be buried in that audit data, i.e. whether sufficient information is manifest in the data to identify an attack from normal data. In approaches based upon machine learning, the design process is further complicated by “noise”. Too frequently in research on anomaly detection, authors state the requirement that the training data must be purely normal and attack-free. This is unrealistic in most cases of computer security auditing where systems are under constant attack and monitoring generates vast quantities of data including attacks. Hand-cleaning data to exclude noise, or attack information, or to label data accurately, is simply not possible. Hence, anomaly detection algorithms must be sensitive to noise and produce models that are robust.

Some approaches to host-based anomaly detection have focused on monitoring the operating system’s (OS) processes during program execution and alerting on anomalous sequences of system calls. For example, OS wrappers monitor each system call or DLL application and test a set of rules for “consistent” program execution [2]. This presumes that a program’s legitimate system call execution can be specified correctly by a set of predefined rules. Alternatively, some have implemented machine learning techniques that model sequences of normal execution traces and thus detect run time anomalies that exhibit abnormal execution traces [6].

There are several important advantages to auditing at the OS level. This approach may provide *broad coverage* and *generality*; for a given target platform it may have wide applicability to detect a variety of malicious applications that may run on that platform.

However, there are several disadvantages to anomaly detection at the OS monitoring level. *Performance* (tracing and analyzing system calls) is not cheap;

there is a substantial overhead for running these systems, even if architected to be as lightweight as possible. Second, the adaptability and extensibility of these systems complicates their use as updates or patches to a platform may necessitate a complete retraining of the OS trace models.

Furthermore, OS system call tracing and anomaly detection may have another serious deficiency; they may suffer from *mimicry attack* [20], since the target platform is widely available for study by attackers to generate exploits that appear normal when executed.

We have taken an alternative view of host-based anomaly detection. Anomalous process executions (possibly those that are malicious) may be detected by monitoring the trace of events as they appear on attempts to alter or damage the machine's permanent store. Thus, a malicious attack that alters only runtime memory would not necessarily be detected by this monitor, while the vast majority of malicious attacks which do result in changes to the permanent store of the host might leave a trace of anomalous file system events. In this case, the two very important host based systems to defend and protect are the Registry (in Window's case) and the file system (in both Window's and Unix cases). The file system is the core permanent store of the host and any malicious execution intended to damage a host will ultimately set its sights upon the file system. A typical user or application will not behave in the same manner as a malicious exploit, and hence the behavior of a malicious exploit is likely able to be detected as an unusual or unlikely set of file system accesses.

The File Wrapper Anomaly Detection System (FWRAP) is presented in this paper as an exemplary application of anomaly detection to computer security applications. FWRAP is a host-based detector that utilizes file wrapper technology to monitor file system accesses. The file wrappers implemented in FWRAP are based upon work described in [25] and operate in much the same fashion as the wrapper technology described in [2]. The wrappers are implemented to extract a set of information about each file access including, for example, date and time of access, host, UID, PID, and filename, etc. Each such file access thus generates a record describing that access.

Our initial focus here is to regard the set of file system access records as a database, and to model the *likely records* in this database. Hence, any record analyzed during detection time is tested to determine whether it is consistent with the database of training records. This modeling is performed by the **P**robabilistic **A**nomaly **D**etection algorithm (PAD) introduced in our prior work on the Windows registry [1]. We report on experiments using alternative threshold logic that governs whether the detector generates an alarm or not depending upon the scores computed by PAD. The PAD detector achieved impressive detection rates in some cases over 95% and about a 2% false positive rate when alarming on anomalous processes.

The rest of the paper is organized as follows. Section 2 briefly describes alternative anomaly detection algorithms and a brief description of the PAD algorithm. Section 3 discusses the architecture of the FWRAP sensor. Section 4 describes the audit data and features computed that are input to PAD to gener-

ate models of normal file accesses. We then detail results of various experiments running real malware against a Linux host. Section 5 describes the experimental setup and the results and findings. Section 5 describes the open problems in anomaly detection research and how this work can be extended.

## 2 Alternative Anomaly Detection Algorithms

Anomaly detection systems for computer security, and host-based intrusion detection specifically, were first proposed by Denning [3]. The concept was later implemented in NIDES [9] to model normal network behavior in order to detect deviant network traffic that may correspond to an attack against a network computer system.

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 [16, 13, 21], others specific to network protocols and LAN traffic for detecting denial of service attacks [12, 18] or Trojan execution, or application or system call-level data for malware detection [6], to name a few.

A variety of different modeling approaches have been described in the literature to compute baseline profiles. These include probabilistic models or statistical outlier detection over (temporal) data [4, 24]; comparison of statistical distributions (or histogram distance metrics) [22], one-class supervised machine learning [13, 21] and unsupervised cluster-based algorithms [5, 15]. Some approaches consider the correlation of multiple models [7, 23]. One-class Support Vector Machines have also been applied to anomaly detection in computer security [8]. W. Lee et al. [11] 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.

In general, in the case that an audit source is a stream or temporally ordered data, a variety of models may be defined for an audit source and a detector may be computed to generate an alarm if a violation is observed based upon volume and velocity statistics. Volume statistics represent the amount of data observed per unit of time, while velocity statistics model the changes in frequency of the data over time. In practice, a number of simple algorithms work surprisingly well. For example, computing “moving averages” over time to estimate the average state of a system, and detecting “bursts” by, for example, noting when the volume or velocity of events exceeds one or two standard deviations from the mean works effectively in detecting distributed denial of service attacks, or scanning/probing activities.

One of the most crucial design choices in designing an anomaly detector is the choice of algorithm, the feature sets and the training methodology. Fundamentally, the issue is whether the resultant models can effectively detect truly abnormal events that correspond to an attack. A fuller treatment evaluating anomaly detection algorithms and their coverage is given by Maxion [17].

The performance of the various anomaly detectors also varies with a number of tunable parameters including the amount and quality of training data (amount of noise present), the threshold settings and the particular decision logic used, and whether the detectors output is validated by correlating with other information. For example, probabilistic based algorithms typically score data by estimating the likelihood of that data. Cluster-based or SVM-based algorithms employ a distance metric or a density measure to estimate whether a datum is normal, or a member of a minority cluster. In either case, a threshold needs to be set to determine whether a data under test is or is not normal. The calibration of this tunable threshold greatly affects the performance of any anomaly detection system, either generating too few true positives, or too many false positives.

Furthermore, computer systems are highly dynamic and rarely reach a stable state. Any normal model computed for one period of time will undoubtedly go stale at some future time. Thus, anomaly detectors require updating and adaption to shifting environments. Detecting when retraining is necessary, or the time “epoch” when a detector ought to be retrained is very much a matter of the specific environment being modeled. In many cases, these issues, calibration, environment shift, and adaptive re-training, are core design requirements necessitating a lot of study, design and engineering for successful systems to be deployed. Bad choices will lead to faulty systems that are either blind to real attacks, or generate so many false alarms as to provide no useful information.

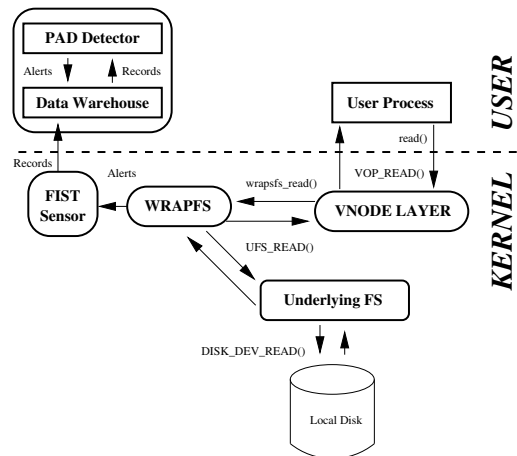
In our prior work, we proposed and developed a number of anomaly detection algorithms. Several cluster-based algorithms were explored [15, 5] as well as probabilistic modeling for sequence data [4], and for database records [1]. The PAD algorithm inspects feature values in its training data set, and estimates the probability of occurrence of each value using a Bayesian estimation technique. PAD estimates a full conditional probability mass function and thus estimates the relative probability of a feature value conditioned on other feature values and the expected frequency of occurrence of each feature. One of the strengths of the PAD algorithm is that it also models the likelihood of seeing new feature values at run-time that it may not have encountered during training. We assume that normal events will occur quite frequently, and abnormal events will occur with some very low probability. Our experiments and application of PAD have been shown to be robust and effective models are trainable in any environment where “noise” is a minority of the training data.

In the following sections we provide an exemplar anomaly detector. We present the FWRAP anomaly detector that audits host-based file systems during a training period, and detects abnormal file accesses at detection time using the PAD algorithm. Anomalous processes are noted by FWRAP when a sufficient number of anomalous file accesses are detected by PAD. We test the effectiveness of the detector by running real exploits against a Windows machine and measure the detector accuracy over varying threshold settings. We begin with a description of the FWRAP architecture.

### 3 FWRAP System Architecture

Several requirements drove the design of the FWRAP system. The file system sensor had to be lightweight, easily portable to different systems, and complete, in the sense that it is able to monitor all file system accesses without loss of information. Perhaps the most important requirement is that the system must be transparent to the user.

Previous work by Zadok [25] proposed a mountable file system for Unix and Windows which would allow additional extensions to the underlying operating system without having to modify kernel level functionality. The FiST technology developed in that work has been extended to provide a security mechanism via file system auditing modules via a Vnode interface. We implemented a FiST audit module that forms the basis of the FWRAP audit sensor. Figure 1 illustrates the architecture that we developed as a standalone real time application on a single Linux host.



**Fig. 1.** The Architecture of FWRAP IDS

Once all subsystem file accesses are logged its a straightforward matter to provide the means of reading from the log, formatting the data and sending it to the PAD module for analysis. A typical snippet of a line of text sent to the PAD module is as follows.

```
Mar 9 19:03:14 zeno kernel:
snoopfs detected access by uid 0, pid 1010, to file cat
```

This record was generated when the root user accesses a file named 'cat' on a machine named 'zeno'. We modified a C program to format this data for PAD exemplified by the following (partial) record.

```
<rec><Month str>Mar</Month> <Day i>9</Day> <Time str>19:03:14</Time>
<IP str>zeno</IP> <UID i>0</UID> <PID i>1010</PID> <File str>cat</File></rec>
```

### 3.1 PAD Detector

The data gathered by monitoring each file access is a rich set of information that describes in great detail a single file access. Each record is treated as a feature vector used by PAD for training a normal model that describes normal file accesses.

PAD models each feature and pairs of features as a conditional probability. A single feature produces a “first order consistency check” that scores the likelihood of observing a feature value at run time. PAD also models the likelihood of observing a new feature value at run-time that was not observed during training. Second order consistency checks score the likelihood of a particular feature value conditioned on a second feature. Thus, given  $n$  features in a training record, PAD generates  $n$  first order consistency checks, and  $n * (n - 1)$  second order consistency checks. Although it is possible to use higher order consistency checks, the computational overhead and space constraints make it infeasible for the current implementation of PAD.

The feature vector available by auditing file accesses has 18 fields of information, some of which may not have any value in describing or predicting a normal file access. For example, one such feature may be the process identifier, PID, associated with the file access. PID’s are arbitrarily assigned by the underlying OS and in and of themselves have no intrinsic value as a predictor of a file access. As an expediency such fields may be dropped from the model. Only 7 features are used in the experiments reported in this paper as detailed in the next section.

After training a model of normal file accesses using the PAD algorithm the resultant model is then used at runtime to detect abnormal file accesses. Alerts are generated via threshold logic on the PAD computed scores. As shown in Figure 1 the detector runs on the user level as a background process. Having it run on the user level can also provide additional protection of the system as the sensor can be hard-coded to detect when it is the subject of a process that aims to kill its execution, or to read or write its files. (Self-protection mechanisms for FWRAP are beyond the scope of this paper.)

### 3.2 FWRAP Features

The FWRAP data model consists of 7 features extracted or derived from the audit data provided by the FWRAP sensor.

Several of the features describe intrinsic values of the file access, for example, the name of the file, and the user id. We also encode information about the characteristics of the file involved in the access, specifically the frequency of touching the file. This information is discretized into a few categories rather than represented as a continuous valued feature. We generally follow a strategy suggested by the Windows OS. Within the add/change applications function of control panel in Windows, the frequency of use of an application is characterized as “frequently”, “sometimes” and “rarely”. Since it is our aim to port FWRAP to Windows (exploiting whatever native information Windows may provide and to correlate FWRAP with RAD), we decided in this experimental implementation of FWRAP on Linux to follow the same principle for the Unix file system but

with a slightly finer level of granularity than suggested by Windows. Hence, we measured file accesses over a trace for several days, and discretized the frequency of use of a file into the four categories as described below.

The entire set of features used by in this study to model file system accesses are as follows:

**UID.** This is the user ID running the process

**WD.** The working directory of a user running the process

**CMD.** This is the command line invoking the running process

**DIR.** This is the parent directory of the touched file.

**FILE.** This is the name of the file being accessed. This allows our algorithm to locate files that are often or not often accessed in the training data. Many files are accessed only once for special situations like system or application installation. Some of these files can be changed during an exploit.

**PRE-FILE.** This is the concatenation of the three previous accessed files. This feature codes information about the sequence of accessed files of normal activities such as log in, Netscape, statx, etc. For example, a login process typically follows a sequence of accessed files such as `.inputrc`, `.tcshrc`, `.history`, `.login`, `.cshdirs`, etc.

**FREQUENCY.** This feature encodes the access frequency of files in the training records. This value is estimated from the training data and discretized into four categories:

1. NEVER (for processes that don't touch any file)
2. FEW (where a file had been accessed only once or twice)
3. SOME (where a file had been accessed about 3 to 10 times)
4. OFTEN (more than SOME).

Alternative discretization of course are possible. We computed the standard deviations from the average frequency of access files from all user processes in the training records to define the category ranges. An access frequency falls into the range of FEW or OFTEN categories often occurs for a file touched by the kernel or a background process.

Examples of typical records gathered from the sensors with these 7 features are:

```
500 /home/linhbui login /bin dc2xx10
725-705-cmdline Some      1205,Normal

500 /home/linhbui kmod /Linux_Attack kmod
1025-0.3544951178-0.8895221054 Never      1253,Malicious
```

The last items (eg., "1253,Malicious") are tab separated from the feature values and represent an optional comment, here used to encode ground truth used in evaluating performance of the detector. The first record with pid=1205 was generated from a normal user activity. The second was captured from an attack running the kmod program to gain root access. The distinction is represented by the labels "normal" and "malicious". These labels are not used by the PAD algorithm. They exist solely for testing and evaluating the performance of the computed models.



Another malicious record is

```
0 /home/linhbui sh /bin su meminfo-debug-insmod Some
1254,Malicious
```

This record illustrates the results of an intruder who gained root access. The working directory (WD) is still at /home/linhbui but the UID now has changed to 0. A record of this nature ought to be a low probability event.

## 4 Experiments

We deployed the FWRAP audit sensor on a “target host” machine in our lab environment, an Intel Celeron 800MHz PC with 256 RAM, running Linux 2.4 with an ext2 file-system. The data gathered by the sensor was logged and used for experimental evaluation on a separate machine. The latter test machine is far faster and provided the means of running multiple experiments to measure accuracy and performance of the PAD implementation. The target host was part of a Test Network Environment which allowed us to run controlled executions of malicious programs without worrying about noise from outside the network corrupting our tests, nor inadvertently allowing leakage of an attack to other systems. Data was not gathered from a simulator, but rather from runtime behavior of a set of users on the target machine.

We collected data from the target host for training over 5 days of normal usage from a group of 5 users. Each user used the machine for their work, logging in, editing some files on terminal, checking email, browsing some website, etc. The root user performed some system maintenance as well as routine sysadmin tasks.

The logged data resulted in a data set of 275,666 records of 23 megabytes which we used to build a PAD model on the other “test machine”. This model will be referred to as the “clean model”, although we note that PAD can tolerate noise. The size of the model was 486 megabytes prior to any pruning and compression.

Once the model was computed, one of the users on the target machine volunteered to be the “Attacker”, who then used the target machine for 3 experiments each lasting from 1 to 3 hours. The malicious user ran three different exploits and three Trojan exploits from their home account. These exploits are publicly available on the Internet. The user was asked to act maliciously and to gain root privileges using the attack exploits on hand. Once root control was acquired, the user further misused the host by executing programs which placed back-doors in the system. The system was monitored while the attacks were run. The resultant monitoring produced records from the FWRAP sensors. These records were then tested and scored by the PAD model.

The PAD analysis was run on the test machine, a dual processor 1500 MHz with 2GB of ram. The total time to build the model of the 23 MB of training data was three minutes, with memory usage at 14%. Once, the model was created, we ran the model against the test data from the 3 experiments, while varying the thresholds to generate a ROC curve. Each detection process took 15 seconds with 40% of CPU usage and 14% of memory.

These performance statistics were measured on the test machine, not on the target host where the data was gathered. This experimental version was implemented to test the efficacy of the approach and has not been optimized for high efficiency and minimal resource consumption for deployment on the target machine. Even so, the analysis of the computational performance of the sensor measured during the experiments indicates that although training the model is resource intensive, run-time detection is far less expensive. A far more efficient professionally engineered version of FWRAP would reduce resource consumption considerably. That effort would make sense only if the sensor achieves the goal of detecting anomalous events indicative of a security breach.

It should be noted that the RAD sensor/detector using PAD running on Windows has been upgraded to run exceptionally fast, with a very small amount of memory and CPU footprint. (The current Windows implementation of PAD requires less than 1 MB of space and the run-time detector consumes at most 5% of CPU, barely discernible.) This newer implementation of PAD is being ported to Linux so that FWRAP will also be far more efficient than the present prototype reported in this paper. The version reported here is the first proof of concept implementation without the performance enhancements implemented on the Windows platform.

## 5 Results

This section describes the results of a subset of experiments we ran. Space does not permit a full treatment and comparative evaluation of alternative alert decision logic.

The PAD algorithm evaluates each file access record output by the sensor by comparing the PAD scores to a threshold value. Each record produces 49 scores for each consistency check (7 first order + 7x6 second order). The minimum score over all consistency checks is then tested against the threshold. If the minimum score is below the threshold, an alert is generated for that record.

An example of the PAD output with threshold = 0.1 is as follows: 8.873115  
8.39732 7.69225 4.057663 0.485905 0.323076 6.527675 8.34453 7.464175 3.727299 0.0  
0.0 5.971592 8.344 7.464175 3.727299 0.0 0.0 5.79384 7.45713 7.454335 4.060443 0.0  
0.0 4.97851 3.732753 3.723643 4.039242 0.0 0.0 3.982627 0.458721 0.371057 0.439515  
0.14842 0.0 0.221132 0.302689 0.20546 0.258604 0.090151 0.0 0.067373 5.978326 5.81323  
5.015466 4.060443 0.0 0.0 :

1254,Malicious (Min score 0.0)

A process is identified as malicious if more than some minimum number of records it generates is scored as an anomaly. This number is a second threshold. We tested varying threshold levels applied to the PAD scores under different thresholds governing the number of anomalous records used to generate a final alert.

The decision process is evaluated by varying the percentage of anomalous records that are generated by a process in order to raise an alert. For example, a process might be considered malicious if it generates one anomalous record, or

all of its records are anomalous, or some percentage of its records are deemed anomalous. We vary this percentage in the following experiments from 10% to 80%.

We define “Detection Rate” as the percentage of all process labeled “malicious” that produced PAD scores below the threshold. The “False Positive Rate” is the percentage labeled “normal” that likewise produced records with PAD scores that were below the threshold. We illustrate the different detection rates and false positive rates over different threshold settings in tabular form. The results indicate that the per-process detection logic provides excellent detection rate and lower false positive rates. This leads to the observation that a malicious process typically generates a considerable number of anomalous events, while many normal processes occasionally generate a few anomalous events. This is not surprising as the same results were discovered in the RAD experiments [1].

**Table 1**, **Table 2**, and **Table 3** detail the results on a Per Process basis. The best results are 95% detection with about 2% false positive rates in experiment 1, 97% accuracy with 7% false positives in experiment 2 and 100% with 8% false positive in experiment 3. Note that the results from experiment 1 are relatively better with respect to false positive rates than those from experiments 2 and 3. The primary reason concerns that amount of training performed during the different experiments. Experiment 1 had far more training data establishing the perhaps obvious point that as the sensor models more events its detection accuracy increases.

In the first experiment implemented on the target machine, there were 5,550 processes generated during the 3 hour period. 121 processes were generated during the attack period (i.e. the time between the initial launching of the attacking exploits and the Trojan software execution after he gained root access). However, only 22 processes generated during this time were spawned by the attack.

Many of the false positives were from processes that were simply not run as a part of the training session but were otherwise normal file system programs.

**Table 1.** Experiment 1, per-process detection. Number of processes: 5550, number of malicious processes: 22

| Threshold | Detection Rate | False Positive |
|-----------|----------------|----------------|
| 1.1       | 1.0            | 0.027090       |
| 1.0       | 0.954545       | 0.02727        |
| 0.9       | 0.909091       | 0.026690       |
| 0.8       | 0.909091       | 0.026345       |
| 0.7       | 0.863636       | 0.025927       |
| 0.6       | 0.863636       | 0.025381       |
| 0.5       | 0.772727       | 0.023363       |
| 0.4       | 0.772727       | 0.021145       |
| 0.3       | 0.727273       | 0.020981       |
| 0.2       | 0.727273       | 0.020909       |
| 0.1       | 0.727273       | 0.020163       |

**Table 2.** Experiment 2, per-process detection. Number of processes: 1344, number of malicious processes: 37

| Threshold | Detection Rate | False Positive |
|-----------|----------------|----------------|
| 1.3       | 1.0            | 0.072485       |
| 1.2       | 0.972973       | 0.071741       |
| 1.1       | 0.972973       | 0.071145       |
| 1.0       | 0.972973       | 0.070104       |
| 0.9       | 0.972973       | 0.068616       |
| 0.8       | 0.972973       | 0.0675         |
| 0.7       | 0.972973       | 0.066532       |
| 0.6       | 0.945946       | 0.062961       |
| 0.5       | 0.945946       | 0.057553       |
| 0.4       | 0.945946       | 0.057328       |
| 0.3       | 0.918919       | 0.057276       |
| 0.2       | 0.918919       | 0.057180       |
| 0.1       | 0.918919       | 0.046897       |

**Table 3.** Experiment 3, per-process detection. Number of processes: 1279, number of malicious processes: 72

| Threshold | Detection Rate | False Positive |
|-----------|----------------|----------------|
| 0.8       | 1.0            | 0.08889        |
| 1.7       | 0.98611        | 0.08850        |
| 0.6       | 0.97222        | 0.08647        |
| 0.5       | 0.86111        | 0.07732        |
| 0.4       | 0.80555        | 0.07544        |
| 0.3       | 0.79166        | 0.07498        |
| 0.2       | 0.79166        | 0.07357        |
| 0.1       | 0.77777        | 0.07107        |

False positives also occurred when processes were run under varying conditions. Command shell execution and file execution of a new application caused false positives to appear. Applications generate processes in different ways depending upon their underlying system call initiation. Furthermore, programs which require a network connection to run correctly caused a false alarm when executed without a network connection. These false alarms arise because the model has not seen behavior from all the different execution behaviors of a given program.

## 6 Conclusions

By using file system access on a Linux system, we are able to label all processes as either attacks or normal, with high accuracy and low false positive rates. We

observe that file system accesses are apparently quite regular and well modeled by PAD. Anomalous accesses are rather easy to detect. Furthermore, malicious process behavior generates a relatively significant number of anomalous events, while normal processes can indeed generate anomalous accesses as well.

The work reported in this paper is an extension of our research on anomaly detection. The PAD algorithm has been previously applied to network traffic, as well as the Windows Registry, as described earlier in this paper. There are a number of open research issues that we are actively pursuing. These issues involve calibration, pruning, feature selection, concept (or environment) drift, correlation and resiliency to attack.

Briefly, we seek automatic means of building anomaly detectors for arbitrary audit sources that are well behaved, and are easy to use. With respect to calibration, one would ideally like a system such as FWRAP to self-adjust its thresholding to minimize false positives while revealing sufficient evidence of a true anomaly indicative of an abuse or an attack. It is important to understand, however, that anomaly detection models should be considered part of the evidence, and not be depended upon for the whole detection task. This means anomaly detector outputs should be correlated with other indicators or other anomaly detection models computed over different audit sources, different features or different modeling algorithms, in order to confirm or deny that an attack is truly occurring. Thus, it would be a mistake to entirely focus on a well calibrated threshold for a single anomaly detector simply to reduce false positives. It may in fact be a better strategy to generate more alerts, and possibly higher numbers of false positives, so that the correlation of these alerts with other confirmatory evidence reveals the true attacks that otherwise would go undetected (had the anomaly detector threshold been set too low).

In the experiments run to date PAD produces fine grained models that are expensive in memory. There are several enhancements that have been implemented in the Windows implementation of PAD for the RAD detector to alleviate its memory consumption requirements. These include pruning of features after an analytical evaluation that would indicate no possible consistency check violation would be possible for a feature at run-time.

Finally, two questions come to most minds when they first study anomaly detectors of various kinds; how long should they be trained, and when should they be retrained. These issues are consistently revealed due to a common phenomenon, concept (or environment) drift. What is modeled at one point in time represents the “normal data” drawn from the environment for a particular training epoch, but the environment may change (either slowly or rapidly) which necessitates a change in model.

The particular features being drawn from the environment have an intrinsic range of values; PAD is learning this range, and modeling the inherent “variability” of the particular feature values one may see for some period of time. Some features would not be expected to vary widely over time, others may be expected to vary widely. PAD learns this information (or an approximation) for the period of time it observes the data. But it is not known if it has observed enough. RAD’s

implementation on Windows provides the means of automatically retraining a model under a variety of user controlled schedules or performance measures. RAD includes a decision procedure, and a feedback control loop that provides the means to determine whether PAD has trained enough, and deems when it may be necessary to retrain a model if its performance should degrade. The same techniques are easily implemented for FWRAP as well.

## References

1. F. Apap, A. Honig, S. Hershkop, E. Eskin and S. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. Fifth International Symposium on Recent Advances in Intrusion Detection, RAID-2002. Zurich, Switzerland, 2002.
2. R. Balzer. Mediating Connectors. 19th IEEE International Conference on Distributed Computing Systems Workshop, 1994.
3. D. E. Denning, An intrusion detection model. IEEE Transactions on Software Engineering, 222, SE-13, 1987
4. E. Eskin. Anomaly Detection Over Noisy Data Using Learned Probability Distributions. In Proceedings of the 17th Intl Conf. on Machine Learning (ICML-2000), 2000.
5. E. Eskin, A. Arnold, M. Prerau, L. Portnoy and S. J. Stolfo. A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. Data Mining for Security Applications. Kluwer 2002.
6. S. Forrest and S. A. Hofmeyr and A. Somayaji and T. A. Longstaff. A Sense of Self for UNIX Processes IEEE Symposium on Security and Privacy, 120-128, 1996
7. A. K. Ghosh, A. Schwartzbard, M. Schatz. Learning Program Behavior Profiles for Intrusion Detection. Workshop Intrusion Detection and Network Monitoring 1999.
8. K. A Heller, K. M Svore, A. D. Keromytis, and S. J. Stolfo. One Class Support Vector Machines for Detecting Anomalous Window Registry Accesses. 3rd IEEE Conference Data Mining Workshop on Data Mining for Computer Security. November 19, 2003.
9. H. S. Javitz and A. Valdes. The NIDES Statistical Component: Description and Justification. Technical report. SRI International, 1993.
10. W. Lee and S. J. Stolfo and P. K. Chan. Learning patterns from UNIX processes execution traces for intrusion detection AAAI Workshop on AI Approaches to Fraud Detection and Risk Management, 50-56, 1997
11. W. Lee and S. Stolfo. 1999. A Framework for Constructing Features and Models for Intrusion Detection Systems. In Proceedings of 1999 IEEE Symposium on Computer Security and Privacy and the Proceedings of the 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. 1999.
12. M. V. Mahoney AND P. K. Chan. Detecting Novel Attacks by Identifying anomalous Network Packet Headers. Florida Institute of Technology Technical Report CS-2001-2, 1999.
13. R. Maxion, and T. Townsend. Masquerade Detection Using Truncated Command Lines. International Conference on Dependable Systems and Networks (DSN-02), Washington, D.C., 2002.
14. C. C. Michael and A. Ghosh. Simple, State-based approaches to Program-based Anomaly Detection. ACM Trans. on Information and System Security, TISSEC. Vol. 5., 2002.

15. L. Portnoy, E. Eskin and S. J. Stolfo. Intrusion detection with unlabeled data using clustering'. Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001). Philadelphia, PA, 2001.
16. M. Schonlau, W. DuMouchel, W. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):58-74, February 2001.
17. K M.C. Tan and Roy A. Maxion. Why 6? Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector. *IEEE Symp. On Security and Privacy*. 2002.
18. C. Taylor and J. Alves-Foss. NATE: Network Analysis of Anomalous Traffic Events, a low-cost approach. In *Proceedings New Security Paradigms Workshop*, 2001.
19. G. Vigna, F. Valeur and R. Kemmerer. Designing and Implementing a Family of Intrusion Detection Systems. *Proc. 9th European software engineering conference*, 2003.
20. D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. *Ninth ACM Conference on Computer and Communications Security*, 2002.
21. K. Wang and S. Stolfo. One-Class Training for Masquerade Detection. *3rd IEEE International Conference on Data Mining, Workshop on Data Mining for Security Applications*, Florida, Nov., 2003.
22. K. Wang, S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. *RAID2004*, France, September 2004.
23. C. Warrender, S. Forrest, and B. Pearlluter. Detecting Intrusions Using System Calls: Alternative Data Models. *IEEE Computer Society*, 1999.
24. N. Ye. A Markov Chain Model of Temporal Behavior for Anomaly Detection. In *Proceedings of the 2000 IEEE Workshop on Information Assurance and Security*, United States Military Academy, West Point, NY. 2000.
25. E. Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *Usenix Technical Conference*. June 2000