Detecting Insider Information Theft Using Features from File Access Logs

Christopher Gates[1], Ninghui Li[1], Zenglin Xu[1],
Suresh N. Chari[2], Ian Molloy[2], and Youngja Park[2]

[1] Purdue University
[2] IBM Research
{gates2,ninghui,xu218}@cs.purdue.edu,
{schari,molloyim,young_park}@us.ibm.com

Abstract. Access control is a necessary, but often insufficient, mechanism for protecting sensitive resources. In some scenarios, the cost of anticipating information needs and specifying precise access control policies is prohibitive. For this reason, many organizations provide employees with excessive access to some resources, such as file or source code repositories. This allows the organization to maximize the benefit employees get from access to troves of information, but exposes the organization to excessive risk. In this work we investigate how to build profiles of normal user activity on file repositories for uses in anomaly detection, insider threats, and risk mitigation. We illustrate how information derived from other users' activity and the structure of the filesystem hierarchy can be used to detect abnormal access patterns. We evaluate our methods on real access logs from a commercial source code repository on tasks of user identification and users seeking to leak resources by accessing more than they have a need for.

Keywords: file, access, insider threat.

1 Introduction

Theft of critical information by malicious insiders is a major threat. Companies may suffer critical damages when disgruntled employees steal intellectual property. There are ample evidences where organizations have suffered significantly due to leakage of large amount of sensitive information accessed by insiders who have legitimate access control privileges to access such information. Insider threats can be caused either by malicious employees or negligent employees who either have their credentials stolen or their devices compromised by malware.

Current access control paradigms, such as role-based access control [1], multi-level security [2], or originator access control [3], are insufficient to deal with this threat of malicious insiders [4]. It is often difficult to a priori predict future needs and configure access control to enforce least privilege in highly dynamic environments. For example, many organizations provide employees access to large data and source code repositories that make it easier to learn and build upon past success without having to reinvent the wheel; in healthcare environments, emergencies often dictate needs. In these cases it

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part II, LNCS 8713, pp. 383–400, 2014.
© Springer International Publishing Switzerland 2014

is thus desirable to allow broader access to resources and monitor for potential abuse later. Such systems expose themselves to risk from malicious insiders who can abuse their authorizations by making many access requests, building up aggregate risk.

In this paper we investigate the approach of using filesystem-derived features to detect such insider threats. We evaluate our techniques on a real dataset derived from a commercial source-code repository from a large organization to detect malicious insiders stealing sensitive information. The techniques developed are applicable to any large file repository or web-based information system, such as a wiki.

Most existing solutions for anomaly detection [5–8] only consider an individual resource or command, or aggregate statistics about file-related operations [9]. These systems often yield high false positive rates, especially for new users or resources. Building profiles based on past history or other resources accessed [10] often results in a blowup in the number of probability point estimates.

One existing approach is to adopt a risk-based access control model [4]. For example, in FuzzyMLS [11], one quantifies the risk entailed by the access to each file, and sums up such risk. The risk is independent of a user's access history—and the access history of other users—and depends on a risk-model on known user- and file-attributes. Such a model is not flexible in dynamic environments and has a high cost to deploy.

Our key insight is that the file system hierarchy, in addition to the behavior of other users, provides meaningful information regarding the relevance of a resource. For example, the location of a new file often indicates the project, component, revision, owner, or type of the file. We discuss how to extract and leverage this information for anomaly / normality testing. Our key idea is to detect abnormal accesses based on comparing a user's current accesses with the history. The hypothesis is that there is significant self-similarity in most user's accesses. That is, each user's accesses during the current time period will be similar to what the user has accessed in the past.

We approach the problem in two steps. In the first step, we define a scoring function that computes the score for the pair of a history (which we abstract as a set of files accessed during the history) and a file that is currently accessed. This scoring function can take into consideration extra information, such as file hierarchy and the histories of other users.

In the second step, we explore how to use such a scoring function to detect malicious insider behaviors. We propose and evaluate several models for scoring and aggregation, and evaluate how well each performs at identifying anomalous behavior. We evaluate our methods on real dataset of access logs, and find that filesystem hierarchy derived features are promising in the field of risk scoring and anomaly detection.

The rest of this paper is organized as follows. In Section 2, we present related work. In Section 3 we define the problem and provide an adversarial model for the scenarios we consider in this work. Details of our proposed approach are presented in Sections 4. We provide a description of the test system and experimental results in Section 5, and conclude in Section 6.

2 Related Work

Most related work is in the area of anomaly or intrusion detection. Denning [7] presented the first host-based intrusion detection system leveraging statistics (frequency,

inter-arrival time, etc.) of events for alerting. Javitz and Valdes [12] later implemented the concept as IDES. These works focus on building statistical profiles of past behavior and issuing alerts when new events exceed significant thresholds, such as a set number of standard deviations.

There has been an abundance of work on network-based intrusion detection [13–16], typically measuring anomalies in the rate or volume of the traffic [13], abnormal numbers of distinct hosts or ports [17], or similarity to known malicious behavior [18], such as blacklists[1] and signatures [19]. Salem et al. [5] present a survey of research on insider attacks for host and network intrusions.

There is relatively little work specifically looking at filesystem events for anomaly detection. Stolfo et al. [10] use filesystem features—primarily filename, working directory, and parent directory—to detect rootkits and other malicious activity. No features are derived from the relationship between two files in the filesystem, and each resource is treated as an opaque identifier. Deviations from first and second order density estimators [8] are used to score events. The system estimates the probability of a previously unseen event.

Huang and Wong [20] discuss the use of a Fuse virtual filesystem to monitor for filesystem anomalies. It uses a "baseline" library of profiles, but provides no details on how this library is generated or how filesystem requests are scored against the library. The TripWire File Integrity system[2] detects anomalies in filesystems by testing for unexpected changes in files using a file digest, similar to how ZFS detects file corruption [21], and cannot be used to detect violations of confidentiality.

Senator et al. [9] use statistical anomalies in file events, specifically the fraction of file events on removable media, to detect injected malicious activity into activity logs obtained from workstations at Raytheon.

Bowen et al. [22] suggest using decoy files to detect malicious insiders. Decoys are files that contain tainted information whose use can be tracked, e.g., credentials or account information. Their system relies on traceable or booby trapped resources rather to detect malicious insiders rather than by analyzing resources users typically require. It is more difficult to perform such attacks in software code repositories where use of the software is more difficult to detect.

Chen and Malin [6] propose an anomaly detection method that clusters weighted graphs of users' access to resources. Their clustering approach is similar to a hybrid of k-nearest neighbor and spectral analysis. If a user does not access similar resources to other users, or changes the cluster of similar users over time, they are considered an anomaly. This work is the most similar to ours in that it considers the resources other users access, but it does not consider the relationships between resources implicit in the filesystem or file names.

3 Problem Definition

At a high level, the problem we want to answer is how to detect malicious insiders who try to steal files they have the privilege or clearance to access. In this section, we

[1] `https://developers.google.com/safe-browsing/`

[2] `http://www.tripwire.com/it-security-software/scm/file-integrity-monitoring/`

introduce a concrete formulation of this problem. Note that while we are using the term "files" here, our approach is applicable to other types of resources where similarity can be measured.

We assume the following inputs. Let $\mathcal{F}$ denote the set of all files with associated information, e.g., the file hierarchy, the type of the files, etc. We are given A_j^t for $1 \leq t \leq T$ and $1 \leq j \leq N$, where N denotes the number of users, j ranges over all users, T is the index of the current time period, t ranges over all time periods, and each $A_j^t \subseteq \mathcal{F}$ denotes the set of files accessed by user j during the t'th time period. We use $A_j^{t_1:t_2}$ to denote $A_j^{t_1} \cup A_j^{t_1+1} \cup \cdots \cup A_j^{t_2}$. We use $\mathbf{A}$ to denote the matrix consisting of all A_j^t's.

For the output, we want to identify the users that are malicious in time period T. More specifically, we want a mechanism M that outputs a non-negative real number when given $\mathbf{A}, \mathcal{F}, j$, where an output of 0 denotes completely normal behavior, and the larger the value, the more suspicious the j'th user activity during the t'th time period. Then the j indices with the highest values are considered abnormal.

3.1 Adversary Model

We try to detect malicious insiders whose objective is to steal (i.e., download/check out) files. We assume that the attacker needs to steal a substantial number of files, and is aware of the mechanism that is being deployed. We consider two kinds of attackers.

- An *impetuous* attacker is one who turns malicious only at time T. An employee who turns malicious after learning that he will be fired soon belongs to this kind.
- A *patient* attacker is one who is malicious at a time earlier than T. Such an attacker can alter his normal access pattern over time to make the attacking activities in time T look benign.

3.2 Challenges and Evaluation Criteria

An important challenge is how to evaluate such a mechanism. While our formulation is close to classification problems in machine learning, one challenge is that we have very little labeled data, and the few labels we have are of limited reliability, containing both false negatives and false positives. To deal with this challenge of limited or missing labels, we treat this as an unsupervised learning problem, and build models using only the access data. We assume this data is *mostly* benign when training and testing. For data representing malicious activity, we inject file accesses representing attempts to steal files. Such data are not used in training, and are used only for testing the effectiveness of our approaches at detecting users with such injected accesses. We vary the number of file accesses injected into one user's access from around 500 to around 12,000, and evaluate the robustness of different approaches.

A second challenge is to be resilient to patient adversary who knows about the deployed mechanism, and may carry out evasion attacks. To evaluate effectiveness against such adversaries, we evaluate how the adversarial strategy of accessing a few files

among the files one plan to steal (i.e., among the files to be injected) impacts the effectiveness of different methods.

4 Proposed Approaches

Some currently deployed systems use the number of files that are accessed as a feature for detecting. This approach, however, is unlikely to be sharp enough to achieve the needed tradeoff between false positives and false negatives. Our intuition is that it should be possible to exploit more information between the files that are accessed in the current period, A_j^T, against the files that have been accessed during the history, $A_j^{T-\ell:T-1}$, which we use A_j^H as a short hand. We drop the subscript j when it is not important. First, if all files that one user currently accesses have been accessed in the recent past by the user, then this is unlikely to be a malicious theft. Second, even if many of the currently-accessed files have not accessed, if they are similar to the files that have been accessed, then this is less likely to be malicious theft. Many ways to measure similarity exist. One possibility is based on the hierarchical structure of the file. Files under the same directory may be viewed as more similar than files that are far apart in the hierarchy. Files that are accessed by essentially the same set of users may be viewed as more similar than files that are accessed by mostly disjoint sets of users. Files that have similar meta-data attribute values, such as file types (e.g., C source code files versus HTML files) may be viewed as more similar. In summary, we want to measure both the "amount" of accesses and the "similarity" of accesses.

At the center of our approach is a function that assigns a score for each file f when given an access history A^H, we use the notation $\mathsf{score}[f|A^H]$ to represent the scoring of f when given A^H. Intuitively, this function measures how "unexpected" a file f is, when given A^H as access history. We desire the following algebraic properties for such a score function:

1. $\mathsf{score}[f|A^H]$ is low when $f \subseteq A^H$. The intuition is that a file already accessed is considered quite normal.
2. $\mathsf{score}[f|A_1^H] \leq \mathsf{score}[f|A_2^H]$ when $A_1^H \supseteq A_2^H$, and as a corollary, $\mathsf{score}[f|\emptyset]$ should be high.

We construct $\mathsf{score}[f|A^H]$ by using a composition of two functions, a similarity measure between two files, and an aggregation function. For each new access f the scoring function is mapped against f and each file in the user's history. The aggregation function reduces the result into a final score. The general function is:

$$\mathsf{score}[f|A^H] = \mathsf{agg}_{g \in A^H}\ \mathsf{score}(f, g)$$

To instantiate this, we need to define both the aggregation function and a scoring function. The $\mathsf{score}(f, g)$ function dictates how two files relate to each other while the agg expresses how a single file f relates to the entire history A^H. In the remainder of this section we present several possible instantiations, and methods to use the scoring function.

4.1 The Scoring Function: score(f, g)

The scoring function for two individual files defines how files relate to each other within the system. We will explore several different techniques to classify their relationships.

Binary Equality. The most basic method is to define a score to test for equality between two files:

$$\mathsf{score}(f,g) = \begin{cases} 0 & \text{when } f = g \\ 1 & \text{when } f \neq g \end{cases}$$

This method works well when users consistently access the same set of files, but cannot adequately handle new resources, such as new files in the same directory as previous requests.

Full Distance. This approach measure the distance between two files if one were to walk the hierarchy to the least common ancestor, lca, which is normalized by the worst case scenario where the lca is the root of the filesystem.

$$\mathsf{score}(f,g) = \frac{\text{length}(f, \text{lca}(f,g)) + \text{length}(g, \text{lca}(f,g))}{\text{length}(f, \text{root}) + \text{length}(g, \text{root})}$$

Lowest Common Ancestor (LCA). Another approach is to look at the lowest common ancestor between the two files. This gives a distance to the branch point, but does not consider how far away the other file is from that branch point. The full distance between two files can sometimes lead to longer than expected paths if there is a deeply nested structure where most of the accesses are occurring at the leafs. This approach evaluates the distance based on the branch of the filesystem being accessed as opposed to the exact files being accessed, and under some types of systems and hierarchies may be a more appropriate scoring technique.

$$\mathsf{score}(f,g) = \frac{\text{length}(f, \text{lca}(f,g))}{\text{length}(f, \text{root})}$$

Note that this is not symmetric, that is $\mathsf{score}(f,g) \equiv \mathsf{score}(g,f)$ is not necessarily true.

Log LCA. The previous method penalizes files near the root more than files deep within the hierarchy. The penalty incurred for being near the root may be too harsh, and so different ways of scaling the score may be applied. One way to scale the score is to take the log of the distances values, which adjusts the scores so files that are very shallow are not penalized as much as the previous technique.

$$\mathsf{score}(f,g) = \frac{log(\text{length}(f, \text{lca}(f,g)) + 1)}{log(\text{length}(f, \text{root}) + 1)}$$

Different scaling techniques also affect how the score of non-exact matches relate to the score of exact matches. In the case of this technique, exact matches still have a score of 0, while matches within the same directory or close directories will have a higher score relative to the non-scaled score.

Access Similarity. Given user sets U_f and U_g which contain the users who access files f and g in the history, we use the Jaccard Distance to define the score function between f and g.

$$\mathsf{score}(f, g) = 1 - \frac{|U_f \cap U_g|}{|U_f \cup U_g|}$$

The underlying hypothesis for this scoring method is that files dissimilar in the hierarchy may be similar for other reasons, and this association is elevated by user access patterns. For example, the specification and implementation files in a source code repository and their corresponding documentation. This is closely related to collaborative filtering, where new resources are suggested based on previous requests.

Discussion. The above techniques are the primary score functions that we examine in this work, but is not meant to be an exhaustive list of filesystem derived features. Future work will consider the order and frequency of file accesses, as well as other file metadata, such as the type. We also note that it is highly unlikely any single scoring function will be sufficient in all possible use cases or for all file requests. We will investigate how well each scoring function performs at discriminating abnormal activity in Section 5.

4.2 The Aggregation Function: $\mathsf{agg}_{g \in A^H}$

A single file $f \in A^T$ generates $|A^H|$ different scores, one for each $g \in A^H$. The agg function defines the way that the system aggregates the $|A^H|$ scores to create a single score for the specific f.

Min Score. One approach is to take $\min$, i.e., $\mathsf{agg}_{g \in A^H} \mathsf{score}(f, g) = \min_{g \in A^H} \mathsf{score}(f, g)$. The advantage of this approach is that it is simple, and in many cases captures the distance effectively. Even a single access in a certain area may be useful to predict where the next accesses are going to occur. The downside of this is that it is susceptible to seeding attacks by "patient adversaries" who may perform a single access in an area that they plan to later access much more broadly. That single access can hide many later accesses and undermine certain scoring functions.

Average. To mitigate the patient adversary attack described above, one can calculate the aggregate score as the average of all similarity score values. This increases the effort for an adversary to seed their history with files similar to the intended target, but may increase the aggregate risk scores for diverse users.

K-Nearest Scores. An alternative that balances the tradeoffs of the minimum and average aggregate functions is to compute the average of the k files in A^H that have the lowest score. This is also vulnerable to "patient adversaries", who can seed the past with a few files in different locations, however it takes more of an effort and some knowledge of k to be effective.

4.3 Feature Generation

The previous techniques produce a way to determine the score for any specific file, in this section we focus on how to use those scores to generate a feature set for a specific user. We look at how to use these scores for inner (to self) and outer (to others) approaches to feature generation.

4.3.1 Cumulative Score

Individual score(f, A^H) results taken as a single value are not able to provide any meaningful context as to possibly malicious behavior. Rather, the scores for all $f \in A^T$ taken together provide more information. We focus on two primary ways of accumulating the risk into a single feature, summing and averaging the scores.

Since we are primarily concerned with a user stealing information, the method of a summing the scores together is one obvious choice since it will generate a higher value when more files are accessed, we define this as

$$sumScore = \sum_{k=1}^{M} \mathsf{score}[f_k | A_j^H]$$

where $f_k \in A_j^T$ and $M = |A_j^T|$ is the number of unique files accessed in the current period. A user who exceeds a risk budget could be flagged and their behavior reviewed. However, summing the scores will result in very unstable values, for instance in one period a user may perform 10x or 100x more accesses then they did in the previous or next period. Any technique which builds a model of the user's expected behavior would need to normalize the information or handle these drastically different cases accordingly.

Averaging the user's scores against the total number of unique accesses they performed is one natural way to normalize the data, $aveScore = \frac{sumScore}{M}$. In this way, activity between periods can be compared more naturally since all values will be in [0,1], and the overall score will be effected by the portion of files that receive a high or low score.

One way to use both the sum or average is to create a single score for each user as they relate to their own history. That is, for a set of users U, and $j \in U$, we calculate the $sumScore$ or $aveScore$ given A_j^T and A_j^H. However, this does not use all the information that is available.

4.3.2 Self Score vs. Relative Scores

Instead of taking a single score for user j, we can generate a matrix of scores $\mathbf{x}$ where $x_{i,j} = aveScore_{i,j}$ using A_i^T and A_j^H. Each row in the matrix represents a single user's current set of accesses A_i^T, and each column indicates how that user relates to the history of the j'th user, A_j^H. The advantage here is that instead of requiring a single score to be fixed above some threshold, we can instead evaluate how all the scores change in the same period. It may be that a user's behavior deviates from their own A^H by a relatively high degree, however if that user's behavior stays consistent to most of the other A^Hs, then this can be an indication that the new behavior reflects a user legitimately accessing new files. Conversely, if a user stays consistent to their own behavior, but deviates highly from all the other A^Hs then this can indicate other types of abnormal behavior.

The novelty here is that in most cases either a global model of user behavior is trained and used to detect abnormal behavior, or a specific profile given one user's history is trained. While the global model is capable of incorporating some of the more complex relationships within the data, it can be difficult to do on such high dimensional and sparse data.

4.4 Using the Features

Given the scores and features constructed in the previous subsection, we now turn to how to use this information.

4.4.1 Self Score Evaluation

The most basic way in which to use the similarity score is to look directly at the $sumScore$ or $aveScore$ for the user's own profile. There are two ways we may want to use this information during an evaluation, anomaly detection and profile identification.

Profile Identification This is also an effective way to associate an unknown $A_?^T$ to the actual user it belongs to. In this process we generate $\mathsf{score}[A_?^T|A_j^H]$ for all $j \in U$, and the A_i^H with the lowest scores generally help to identify the user that actually generated the accesses.

Anomaly Detection If the score for a particular set of accesses is above some threshold, then it is marked as abnormal.

We will see in our evaluation that even this simple metric on the scores can be effective. This is the only technique that we use which only looks at a user own score, the rest of this section discusses techniques that look at the all of the scores as a larger set of features, comparing a single access pattern back to all users' histories.

4.4.2 Mean Vector

This technique is similar to centroid based clustering techniques with known user labels for all points. Given the full features $\mathbf{x}$ where $x_{i,j} = \mathsf{score}[A_i^T|A_j^H]$ over many time periods, we find a mean vector to represent each user, which is essentially the center point for a cluster that will represent a specific user's expected behavior. The advantage of this technique is that each user's accesses will relate to other users in specific ways based on similar access patterns and job responsibilities so it adds more information into the system. Once we know the mean vectors for all users, we can compare any new feature vector to determine how close that vector is to the mean of each specific user. Cosine Similarity is used to measure the distance between the centroids and new feature vectors. This handles outliers in the training period well by smoothing the expectation out over all the training points. However this does not account well for cases where a user may be performing several different job functions over different periods but works well in general.

5 Experimental Results

The techniques presented can be applied to any system that manages sensitive resources, such as document repositories, online wikis, and source code repositories. While we extensively leverage the filesystem hierarchy, the presented techniques are general enough to be adapted to other domains. For example, the Wikipedia ontology or the shortest path between two URLs can be used as a substitute.

In our experiments we focus on a commercial source code repository for a large organization. Source code is an attractive target for malicious insiders and has an extremely high value, such as the theft of Goldman Sachs source code by Sergey Aleynikov [23],

and negative impact to the organization, for example the RSA SecureID[3] or Adobe breaches[4]. Source code is often organized into hierarchies, and access is often limited by job function and expertise. Source code has many of the characteristics of other file repositories that make anomaly detection difficult. Files are not consistently accessed, and become stable over time, e.g., libraries, while new files are constantly being added or removed. Further, many users require different levels of access. Those responsible for building code require broad read-only access, while many developers need narrow read-write access. Debugging may often require an employee to investigate how other components function to narrow down root causes. This all makes finding stable and consistent access patterns challenging.

The source code management system we use in our study is Configuration Management Version Control (CMVC). Each file in CMVC is associate with a filename and a location in a hierarchy, and files can be grouped into components orthogonal to the filesystem hierarchy. For example, not all files in a directory need to be in the same components, and a component can contain files from any directory. The components may be further nested, and users are granted access to check in and check out files by authorizing them to components. CMVC also includes extensive reporting, task and defect management, and release levels to make administration of large projects easier.

Lines from the log consist of a timestamp, userID, action performed on a resource, and the name of the resource. The logs contain additional lines which relate to other information such as reporting and defects, however we limit our view of the logs to the file activity. For our task, we consider only accesses that result in file reading. CMVC does not log which component a request pertains to, and we do not have access to the access control lists, historical or current.

For our task we analyze one year of log data consisting of approximately two-thousand users. There are $\sim$512k unique files and $\sim$133k unique directories in the filesystem. Since there must be some history for all of our techniques to be useful, there is a single period of learning the initial history. Then there are 10 meaningful periods of training data, and a final period used for testing.

5.1 Portion of Files Accessible

One way to measure the effectiveness of each scoring technique is to measure how well it scores files that the user does access compared to files that the user does not access in a given time period. We generate a set of uniformly sampled files to represent the 'All Possible Access' group, while using each user's actual accesses for the other group.

Figure 1 plots t at every increment of .05 between 0 and 1, the x and y values are generated by the following formulas :

[3] http://www.darkreading.com/attacks-and-breaches/rsa-securid-breach-cost-$66-million/d/d-id/1099232?

[4] http://arstechnica.com/security/2013/10/adobe-source-code-and-customer-data-stolen-in-sustained-network-hack/

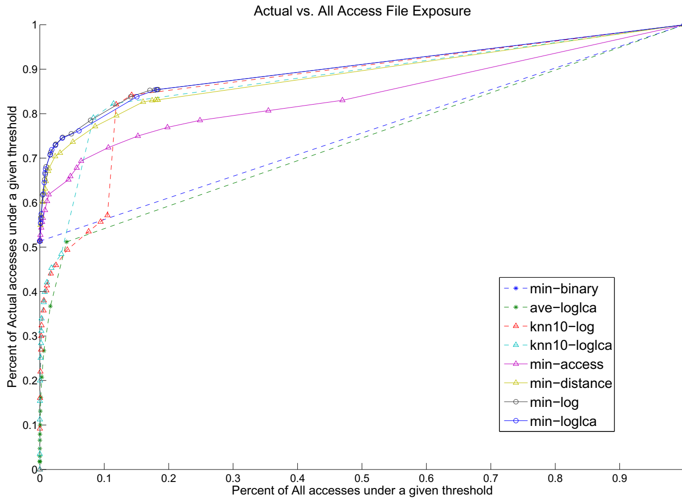

Fig. 1. Actual Access vs. All Possible Accesses

$$x(t) = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{\# files in hierarchy for user i under score t}}{\text{\# files in hierarchy}}$$

$$y(t) = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{\# files access by user i under score t}}{\text{\# of files accessed by user i}}$$

The x value represents, averaged across all users, how many files in the complete hierarchy have a score under threshold t. The y value represents, averaged across all users, the number of actual accessed files in a given period that have a score under threshold t.

In an ideal scenario, with knowledge of the future, all legitimate accesses made in a given period would receive a score less than all of the files from the group the user did not access. This would create a line from the upper left, (0,1), to the upper right, (1,1), of the graph. Given unpredictable human behavior and shifting responsibilities, this is of course impossible, and so we desire a scoring function which gets closer to the upper left but also allows for changing behavior.

All techniques which use min as the aggregation function start their curve at $\sim$52% since on average a user accessed around half of their files in a previous time period, and exact matches get a score of 0 for all min techniques. The min-loglca and min-log stand out as performing the best among the techniques across all thresholds. The knn10-lca and knn10-loglca show a marked difference between the lca vs loglca techniques.

Averaging the distance between all files in the history to the current file does not perform well overall in this task. Originally this seemed like one possibly useful technique since it naturally weights branches of the hierarchy with more accesses, however, this potential advantage seems to be overwhelmed by other properties of the access behavior and file hierarchy.

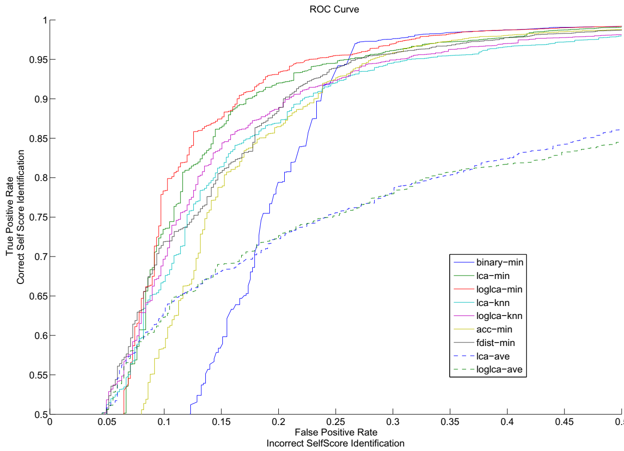

Fig. 2. ROC curve to compare the overall relative performance of each technique

The binary score function in Figure 1 is represented by two points and creates the line from (0,.52) to (1,1). The binary technique is useful given the nature of our target system, a source code repository where roughly half of accesses have already been performed in the past. However, under different conditions where most accesses are new and unique, such as classified documents or medical records, then repeat accesses to the same data across multiple time periods could be less common and this signal would become weaker.

5.2 Profile Identification

Another task we explore is how effective each technique is at identifying which user generated a specific set of access. That is, given a random A^T, how well can we predict which user generated that set of accesses. For this task we compute $\mathbf{x}$ where $x_{i,j} = aveScore_{i,j}$ as specified in Section 4.3.2. We denote the cases where $i = j$ as the "SelfScore", and the cases where $i \neq j$ as "OtherScore". The percent of self scores and other scores that are assigned a score in a specific range for a subset of techniques are presented in the Appendix in Figure 5 and the ROC curves for all techniques are presented in Figure 2 in this section. This gives an indication of how well each technique performs in identifying a user's own behavior. One thing to note here is that user accesses can be highly correlated, and so it is not necessarily abnormal for some "OtherScores" to have low values.

The generic scaling for depth, taking the log of the lowest common ancestor and log of the distance to root, has a slight performance impact for our data, seen in the difference between Figures 5(a) and 5(b) and in Figure 2.

Given the outcome from Figure 1 and 2 we focus the remaining experiments on the min-loglca technique to generate features. The min-binary technique is used as a baseline that only counts unique accesses in the current period given all previous periods. This is different from counting the unique accesses in the current period, which we also use as another baseline since this is the most commonly used statistic in related work.

5.3 Attacker

The primary goal for the attacker we model is on data exfiltration of varying degrees. While there are other potential attacks, such as targeted insertion or deletion, for this evaluation we only focus on the general problem of stealing information. Due to the nature of the target application domain there are several assumptions that we make about the data and the attacker.

Arbitrarily Self Control. An attacker using account u_i can arbitrarily control file accesses in A_i^T and A_i^H.

Restricted Overall Control. An attacker is not able to control another user's activity on the system.

Targeted Knowledge. The attacker has knowledge of the files or directories they are targeting, and does not have to perform a read all on the root of the repository.

Location Stability. For the purposes of this evaluation, we assume that files are stable in their location in the hierarchy. It is possible that files can be moved, but given the log data, this is very infrequent. Additionally, if an attacker wants to directly move a file, then the action is captured in the logs and will count as both a read from the source and write to the destination, which would translate to the same general information as just reading or writing a file directly.

With this in mind we discuss two attack types that we test against, impetuous and patient attackers.

5.3.1 Impetuous Attacker

An impetuous attacker is a user who does not have the time or ability to create a crafted attack. It represents an employee who is suddenly laid off or leaving the company, a naive user who is unaware of the protections in place, or an attacker who is afraid of being detected and so they grab as much data as quickly as possible.

To model an impetuous attacker, we generate injections that consist of randomly selected directories that contain file counts in various ranges, we then inject all files under that directory into the accesses in A^T to simulate that the user accessed all information under a specific directory. We generate data for 3 ranges to capture the effect that different access counts have on detection: 500-1000 accesses contains 10 unique attacks, 1000-2500 accesses contains 12 unique attacks and 5000+ accesses contains 2 unique attacks. We compare the detection rates for all injections in a given range against the actual accesses to determine the true positive and false positive rate for various techniques.

Figure 3 demonstrates detection when a user has abnormal access activity. SumScore and AveScore for each user, as defined in Section 4.3.1, are generated with the loglca-min technique. NewUnique is equivalent to taking the SumScore of binary-min for a user's own profile.

Unique accesses in the current period and NewUnique access are a baseline, AveScore and SumScore are calculated against the user's own profile. The MeanDistance is calculated as discussed in Section 4.4.2, using the historic $\mathbf{x}$ vectors from the previous periods as training data to learn an expected profile for each user. Multiplying the MeanDistance by the NewUnique gives a higher score to abnormal behavior that

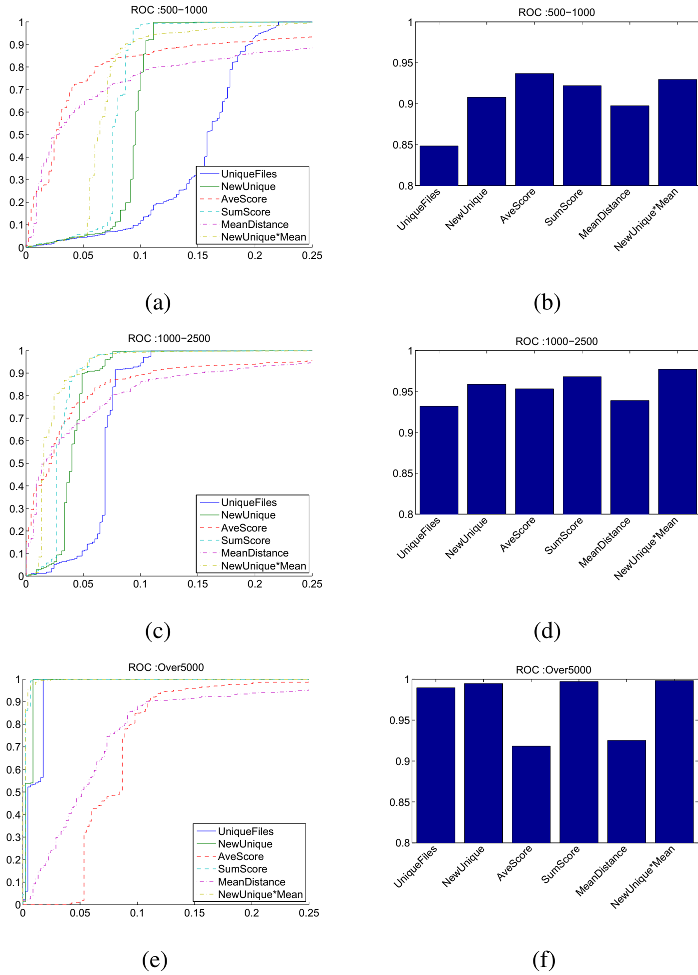

Fig. 3. Demonstrates the performance of detecting various quantities of injected files into otherwise normal behavior

also accesses many files compared to abnormal behavior that only accesses a few files, and is similar to the difference between AveScore and SumScore.

The NewUnique value generally performs well in our tests, however this is in part due to the nature of our dataset where many accesses are not unique in the actual data. On average, users access ∼600 unique files in the test period, and ∼300 of those are new files the user has not previously accessed.

The AveScore performs best in the lowest range, since it is able to differentiate between expected and unexpected behavior, while the SumScore performs better as the injected file counts increase since this more strongly penalizes larger sets of unique accesses. The MeanDistance alone does not perform best overall, but scaled by the number of NewUnique accesses performs well in the 500-1000 range, and best in the 1000-2500 and 5000+ ranges.

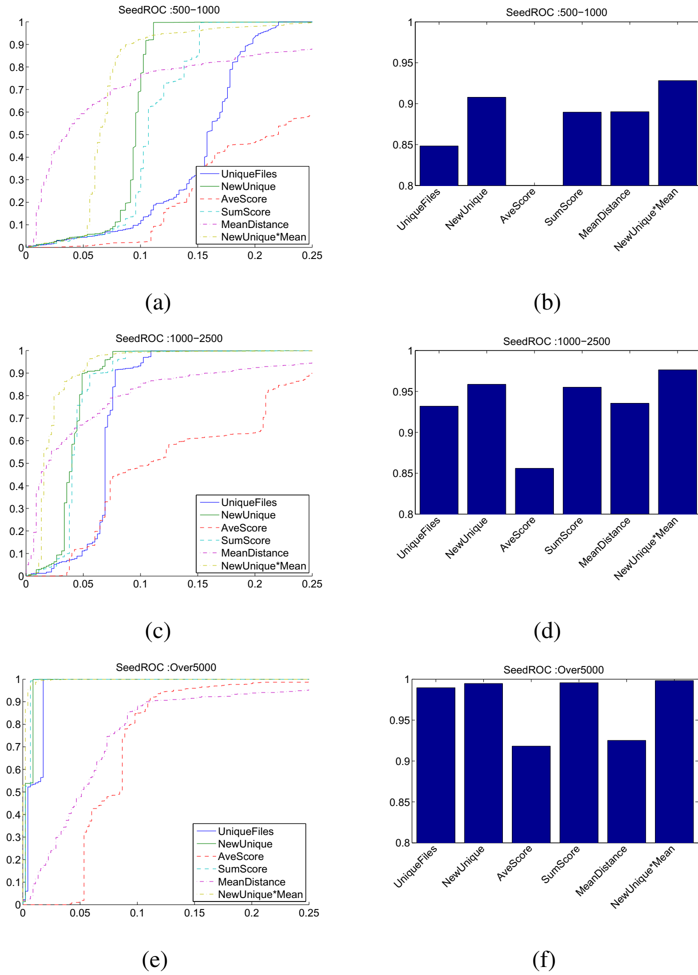

Fig. 4. Demonstrates the performance of detecting various quantities of injected files when the attacker also seeds an attack ahead of time

5.3.2 Patient Attacker

Our attack model assumes that the attacker is capable of injecting accesses into their own training periods, we model this as a patient attacker who has both the time and knowledge to craft a more meaningful attack in order to manipulate the detection techniques. Under this model, an attacker has arbitrary control over their own score, but less control over their relation to other profiles.

We use the same set of injections as the impetuous attack evaluation. To model the patient attacker we seed file accesses from the injection into the A^H, and then generate the features vectors $\mathbf{x}$ from A^T based off of the seeded A^H. We injected seeds for various access counts, but find that a single seed access is nearly as effective as most strategies, and so we present only single access seed results in Figure 4.

The features are generate in such a way that $x_{i,i}$ uses the seeded A^H, while $x_{i,j}$ for $i \neq j$ uses the non-seeded A^H. Each row in the feature vector $\mathbf{x}$ reflects the outcome of a seeded attack by the user represented by that row, and not the case when all users spontaneously decide to inject their own profiles with the same seed.

Any techniques that only use a the self score will be affected since the self score is easy to manipulate under the hierarchy based similarity techniques. The SumScore is less affected since the injection always increases the score over the normal activity. More aggressive seeding techniques would cause a bigger drop in the SumScore AUC.

The techniques that use the entire feature vector for a user as they relate to all other users are more robust to seeding attacks as seen in Figure 4 and particularly they are stable when compared against Figure 3. They also get the highest AUC value and lead to fewer false positives in the early part of the ROC curve. For the 1000:2500 case we can detect about 80% of attacks with 2.5% of normal accesses as false positives using the NewUnique*MeanDistance method, compared to 80% detection at just below 5% FP for the NewUnique method. While 2.5% may be too high depending on the number of users in the system and the resources of the organization to investigate alerts, this is still a meaningful improvement over the baseline.

6 Conclusions

The techniques we propose in this paper are a first step to better use hierarchy and similarity information to understand a user's behavior and detect behavior that is most likely malicious. While we have shown that the detection rates can be improved using our proposed methods, this is just a first step. There seems to be potential in collaborative learning on this complex data that contains rich relational information. The end goal is to utilize this information more effectively to achieve even better detection with fewer false positives and to take the burden away from the incident response teams who have to deal with alerts from any such system.

References

1. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer 29(2), 38–47 (1996)
2. Bell, D.E., LaPadula, L.J.: Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation (March 1976)
3. Park, J., Sandhu, R.: Originator control in usage control. In: Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks 2002 (2002)
4. Horizontal integration: Broader access models for realizing information dominance, JASON Report JSR-04-132 (2004)
5. Salem, M., Hershkop, S., Stolfo, S.: A Survey of Insider Attack Detection Research. In: Insider Attack and Cyber Security, pp. 69–90 (2008)
6. Chen, Y., Malin, B.: Detection of anomalous insiders in collaborative environments via relational analysis of access logs. CODASPY 2011: Proceedings of the First ACM Conference on Data and Application Security and Privacy (February 2011)
7. Denning, D.E.: An Intrusion-Detection Model. IEEE Transactions on Software Engineering SE-13(2), 222–232 (1987)

8. Apap, F., Honig, A., Hershkop, S., Eskin, E., Stolfo, S.J.: Detecting malicious software by monitoring anomalous windows registry accesses. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, p. 36. Springer, Heidelberg (2002)
9. Senator, T.E., Goldberg, H.G., Memory, A., Young, W.T., Rees, B., Pierce, R., Huang, D., Reardon, M., Bader, D.A., Chow, E., Essa, I., Jones, J., Bettadapura, V., Chau, D.H., Green, O., Kaya, O., Zakrzewska, A., Briscoe, E., Mappus, R.I.L., McColl, R., Weiss, L., Dietterich, T.G., Fern, A., Wong, W.K., Das, S., Emmott, A., Irvine, J., Lee, J.Y., Koutra, D., Faloutsos, C., Corkill, D., Friedland, L., Gentzel, A., Jensen, D.: Detecting insider threats in a real corporate database of computer usage activity. In: KDD 2013: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM Request Permissions (August 2013)
10. Stolfo, S.J., Hershkop, S., Bui, L.H., Ferster, R., Wang, K.: Anomaly detection in computer security and an application to file system accesses. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 14–28. Springer, Heidelberg (2005)
11. Cheng, P.C., Rohatgi, P., Keser, C., Karger, P.A., Wagner, G.M., Reninger, A.S.: Fuzzy MLS: An Experiment on Quantified Risk-Adaptive Access Control. In: IEEE Symposium on Security and Privacy (2007)
12. Javitz, H.S., Valdes, A.: The SRI IDES Statistical Anomaly Detector. Research in Security and Privacy (1991)
13. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23-24), 2435–2463 (1999)
14. Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 305–316 (2010)
15. Mahoney, M.V., Chan, P.K.: Learning nonstationary models of normal network traffic for detecting novel attacks. In: KDD 2002: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Request Permissions (July 2002)
16. Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, S&P 2001, pp. 130–143 (2001)
17. Lakhina, A., Crovella, M., Diot, C., Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions, vol. 35. ACM (October 2005)
18. Mathur, S., Coskun, B., Balakrishnan, S.: Detecting hidden enemy lines in IP address space. In: NSPW 2013: Proceedings of the 2013 Workshop on New Security Paradigms Workshop (December 2013)
19. Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K.: Kargus: a highly-scalable software-based intrusion detection system. In: CCS 2012: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM Request Permissions (October 2012)
20. Huang, L., Wong, K.: Anomaly Detection by Monitoring Filesystem Activities. In: 2011 IEEE 19th International Conference on Program Comprehension (ICPC), pp. 221–222. IEEE (January 2011)
21. Bonwick, J.: Zfs end-to-end data integrity (December 2005)
22. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Chen, Y., Dimitriou, T.D., Zhou, J. (eds.) SecureComm 2009. LNICST, vol. 19, pp. 51–70. Springer, Heidelberg (2009)
23. Glovin, D., Harper, C.: Goldman trading-code investment put at risk by theft (2009)

A Appendix

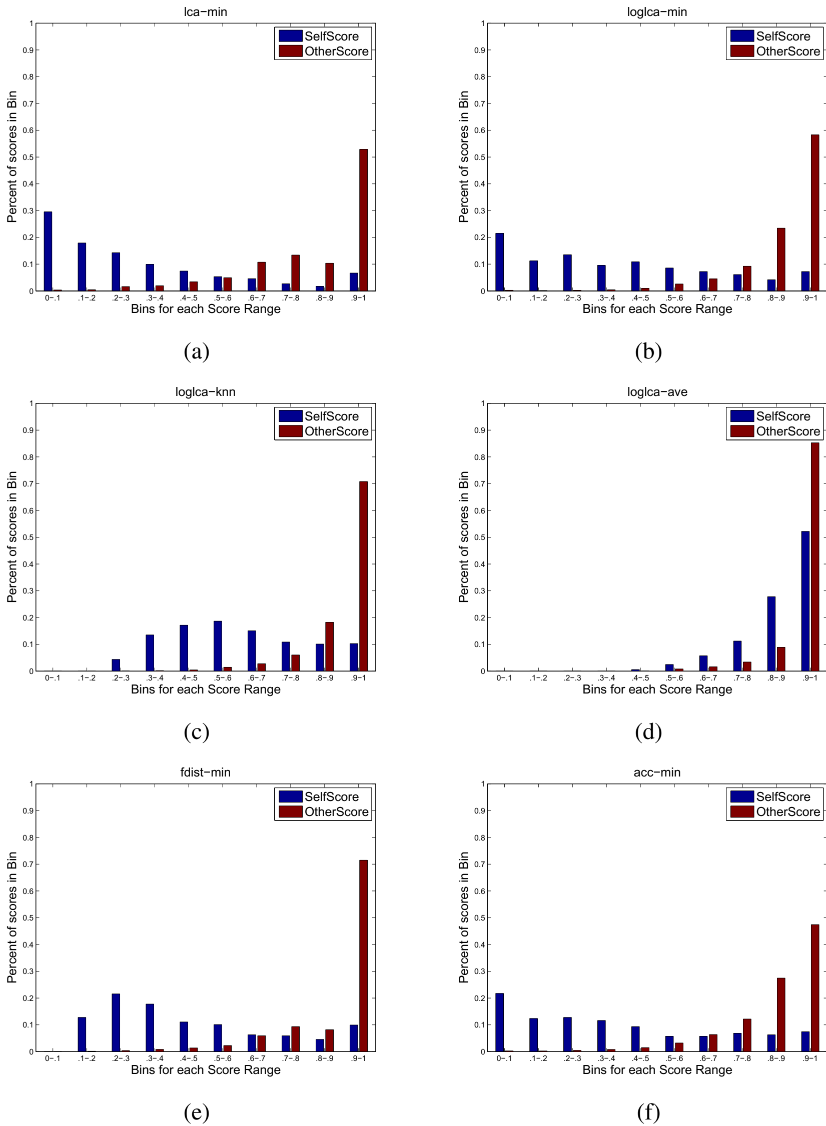

Fig. 5. The percent of Self Scores vs Other Scores in each score range for various techniques using the aveScore output