Detecting Cyberattack Entities from Audit Data via Multi-View Anomaly Detection with Feedback

Md Amran Siddiqui
School of EECS
Oregon State University
Corvallis, OR, USA
siddiqmd@eecs.oregonstate.edu

Alan Fern
School of EECS
Oregon State University
Corvallis, OR, USA
afern@eecs.oregonstate.edu

Ryan Wright
Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
ryan@galois.com

Alec Theriault
Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
atheriault@galois.com

David Archer
Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
dwa@galois.com

William Maxwell
Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
wmaxwell90@gmail.com

Abstract

In this paper, we consider the problem of detecting unknown cyberattacks from audit data of system-level events. A key challenge is that different cyberattacks will have different suspicion indicators, which are not known beforehand. To address this we consider a multi-view anomaly detection framework, where multiple expert-designed "views" of the data are created for capturing features that may serve as potential indicators. Anomaly detectors are then applied to each view and the results are combined to yield an overall suspiciousness ranking of system entities. Unfortunately, there is often a mismatch between what anomaly detection algorithms find and what is actually malicious, which can result in many false positives. This problem is made even worse in the multi-view setting, where only a small subset of the views may be relevant to detecting a particular cyberattack. To help reduce the false positive rate, a key contribution of this paper is to incorporate feedback from security analysts about whether proposed suspicious entities are of interest or likely benign. This feedback is incorporated into subsequent anomaly detection in order to improve the suspiciousness ranking toward entities that are truly of interest to the analyst. For this purpose, we propose an easy to implement variant of the perceptron learning algorithm, which is shown to be quite effective on benchmark datasets. We evaluate our overall approach on real attack data from a DARPA red team exercise, which include multiple attacks on multiple operating systems. The results show that the incorporation of feedback can significantly reduce the time required to identify malicious system entities.

1 Introduction

The frequency of cyberattacks is rapidly increasing in all sectors of personal, enterprise, government, and medical computer systems. While there are many effective tools for detecting and mitigating known attacks, tools for detecting novel attacks are still unreliable. While fully automated detection of cyberattacks is a clear goal for the future, a more realistic and important near-term goals is to develop security systems that interact with security analyst in order to discover malicious entities as quickly as possible. In this paper, we move toward such a system by developing an approach for interactively ranking system entities, such as processes, files, and netflows (network sockets), according to their level of suspiciousness. In particular, our system analyzes streams of system-level audit data from a host machine in order to determine the most suspicious entities, which can be presented to an analyst for further investigation.

A traditional problem with such security systems is the high false positive rates, where many benign system entities are presented to the analyst. This is due to the fact that the event streams corresponding to an attack are usually buried under a huge number of other benign system events generated by normal host activity. This problem is made more difficult due to the fact that we would like a security approach to be effective across many host systems, which will each exibit unique behaviors. In order to address this challenge, we propose an adaptive approach, where feedback from a security analyst is used to accelerate the rate by which truly malicious entities are discovered.

In particular, we propose an approach based on multi-view anomaly detection with feedback. The approach is based on first designing a set of data views that each extract certain features from the audit data that may indicate malicious entities. We then use a state-of-the-art anomaly detector to extract signals from these views and later combine them to yield an overall suspiciousness ranking of the system entities. Unfortunately, the baseline view combination approach generates too many false positives due to unsupervised nature of the anomaly detectors. The main contribution of this paper is to show that by incorporating a small amount of expert feedback, which indicates whether proposed system entities are benign or malicious, the time required to identify malicious entities can be significantly accelerated. For this purpose, we develop a variant of the perceptron learning algorithm that adapts the multi-view anomaly detector to more effectively score entities in accordance with the feedback.

We demonstrate our approach on real attack data collected from a DARPA red team exercise, which contains multiple attacks on multiple operating systems, over multiple days.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The results show that feedback allows for malicious entities to be discovered significantly earlier than when no feedback is used by the system.

Related Work. Host based malicious system behavior or intrusion detection from audit data using anomaly detection is not a new concept. A lot of works, for example (Dong et al. 2017; Forrest et al. 1996; Gao, Reiter, and Song 2004; Sekar et al. 2001; Shu, Yao, and Ramakrishnan 2015), has been devoted to it. The major focus of these works were on learning normal behavior from sequences of system calls or execution control flow. The approaches show promise but are prone to high false positive rates, which hinders their use. Our proposed approach is fundamentally different in that we start by recognizing the fundamental difficulty of achieving low false positive rates using fixed detection schemes across a wide range of host system. By adapting the detection system based on analyst feedback, we aim to provide a more robust system that can quickly be tuned to be effective on any given host system. To the best of our knowledge there is no prior work that aims to incorporate analyst feedback into a machine-learning based detection system.

2 System Overview

Figure 1 shows the high-level architecture of our overall system. The input to our system is an audit stream of system level events from the host being monitored. As described in Section 3, the data format is operating system independent, which means our system can be host agnostic. This data stream is processed in real-time by our data ingestor (Section 3), which produces a graph database representation of the data for later access by system analysts. The ingestor also computes a set of "data views", which are designed by domain experts as a way of capturing features of system entities that are potentially useful for detecting malicious activity. Intuitively, the view collection is designed with the intention that malicious activity will generally appear to be anomalous in one or more of the views. The views are sent to a multi-view anomaly detection algorithm (Section 4), which combines the information to produce a ranking of system entities according to their suspiciousness.

The ranking produced by the anomaly detector is fed to a user interface, which allows an analyst to explore events related to the high-ranked entities via a graphical interface connected to the graph database. Since the focus of this paper is on the anomaly detection component, we do not detail the graphical interface portion of the system. After exploring an anomaly, the analyst can provide feedback about whether the anomaly was determined to be benign and unlikely related to an attack, or whether it is indeed part of an attack or potentially part of an attack. This feedback is provided to the anomaly detector, which incorporates the new information in order to re-rank the entities (Section 5). The intent is for the feedback from the analyst to allow the anomaly detector to more quickly uncover entities that are truly of interest to the analyst. For example, when the anomaly detector receives feedback that an entity is benign, the detector will be less likely to assign high anomaly scores to similar entities. The feedback loop between the anomaly detector and analyst continues as long as analyst resources are available.

3 Data Ingestion

This section gives an overview of the data ingestion component of our system, which is responsible for real-time preparation of the audit data stream for further processing.

Data Format. The audit data under investigation is collected from one of several different operating systems supported by the auditing tools including Linux, Windows, FreeBSD, and Android. These tools were developed by several teams as part of a large DARPA project and used to process system data generated from multiple hosts under the control of a red team. Regardless of the originating source, the data is represented with a common syntactic format capable of representing system activity at a level of abstraction suitable for encoding operating system activity such as process forking, thread spawning, making system calls, creating network sockets, reading file descriptors, etc.

The data is collected on each host system and shipped off the machine as a stream of statements that can be structured as a heterogeneous property graph. Each node in the graph has a unique ID and a set of key/value pairs corresponding to properties. Edges are encoded as specially-named properties with the ID of the other node as its value.

Node types in the graph are either Subjects, Objects, or Events. Subjects represent processes, threads, or lower-level units of execution like loop iterations. Objects represent files, network communication, Windows registry entries, or memory regions. Events are also represented as nodes which have edges to other Subjects and Objects. For example, a process reading a file is represented by a READ event node with an edge to the Subject node performing the read, and another edge to the file Object node of the file being read.

Ingest Process. To ingest and process the data, we built a streaming system based on the Actor model (Haller and Odersky 2006). Each statement representing one node in the graph is read from the source and handed off to a lightweight actor which then routes that statement to other actors for further processing based on the data contained in that statement. Our system uses Akka[1] as the implementation of the actor model and its streaming DSL, Akka Streams, to define computation for each step in the ingest pipeline.

With this system we observed ingest rates in excess of 160k elements per second, with the system apparently IO bound as it was reading data from a single source. Average statement size varied with the data provider, and ranged between 116 and 509 bytes per statement when serialized with Avro into a binary format.

The ingest pipeline is back pressured so that all stream components are slowed down to the rate of the slowest component–and no component is overwhelmed with more data than it can handle. This approach allows identical stream logic to be attached to faster and slower downstream components interchangeably. Because of this capability we were able to ingest data from the source, then split the stream into one pipeline for writing in a graph database, and another pipeline for performing anomaly detection. Since the focus of this paper is on the anomaly detection component, we do not detail the construction of the graph database.

[1] http://www.akka.io

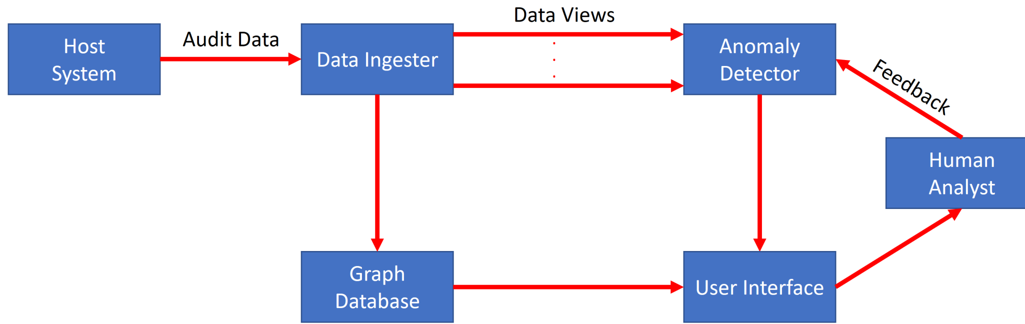


Figure 1: Overview of the multi-view anomaly detection system

Table 1: Summary of the views and entity counts in the three sets of data used in our experiments (last three columns). The counts for total number of events, processes, files, and netflows on each host is given along with the number of those entities that are malicious, i.e. involved in the attack.

View Name	#Features	EntityType	Host1(FreeBSD)	Host2(FreeBSD)	Host3(Ubuntu)
All Process Features	61	Process	# total = 16818	# total = 52979	# total = 57512
Process Directory Scan	3				
Process Exec from Network	4				
Process File Events	17				
Process Memory Events	5		# malicious = 18	# malicious = 13	# malicious = 30
Process Netflow Events	8				
Process Process Events	12				
All File Features	33	File	# total = 119405	# total = 318898	# total = 18019
Downloaded File Execution	5				
Exfil Staging File	4				
File Executed Stats	9				
File MMap Stats	4		# malicious = 22	# malicious = 31	# malicious = 19
File Modify Event	8				
File Permission Event	3				
All NetFlow Features	18	Netflow	# total = 36	# total = 124	# total = 1955
Beaconing Behavior	3				
Netflow Read Stats	5				
Netflow Read Write Rate Lifetime	3		# malicious = 2	# malicious = 7	# malicious = 2
NetFlow-related File Anomaly	6				
Netflow Write Stats	5				
Number of system events			# total = 2.3M #malicious = 316	# total = 8.98M #malicious = 473	# total = 5.8M #malicious = 1139

Anomaly Detection Preprocessing. As described in Section 4 a key part of our system is the computation of multiple views of the event stream for use in anomaly detection. The primary computation involved in view construction is to compute a set of features for system entities that aggregate information about events related to the entity. To do this in a streaming fashion our ingestor aggregates all event data for each Subject and each Object, copying the event when necessary to become a member of both sets. The result is a collection of two-element tuples, each composed of a set of Events and either one Subject or one Object.

From these tuples, paths in the event stream can be built in a streaming fashion either by intersecting sets or by using IDs from events in sets to lookup new sets on the other end of the edge. Tuples are grouped together according to the view definitions (see Section 4) defined in advance and used to compute a collection of features for each item in the view.

To minimize the memory usage of this streaming aggregation, we exploited the fact that most events for a single Subject/Object arrive near each other in the stream. Each aggregated tuple was managed by a lightweight actor which serialized its data to disk after receiving no new events for a specified time period. We used a fast key-value store for this purpose which is able to keep the total memory usage of our JVM-based pipeline to only a few hundred megabytes of RAM. The anomaly component of our system was observed to run at approximately 50k statements per second.

4 View-Based Anomaly Detection

In this section, we describe our view-based anomaly detection approach to detect malicious system entities, which will be extended to incorporate analyst feedback in Section 5.

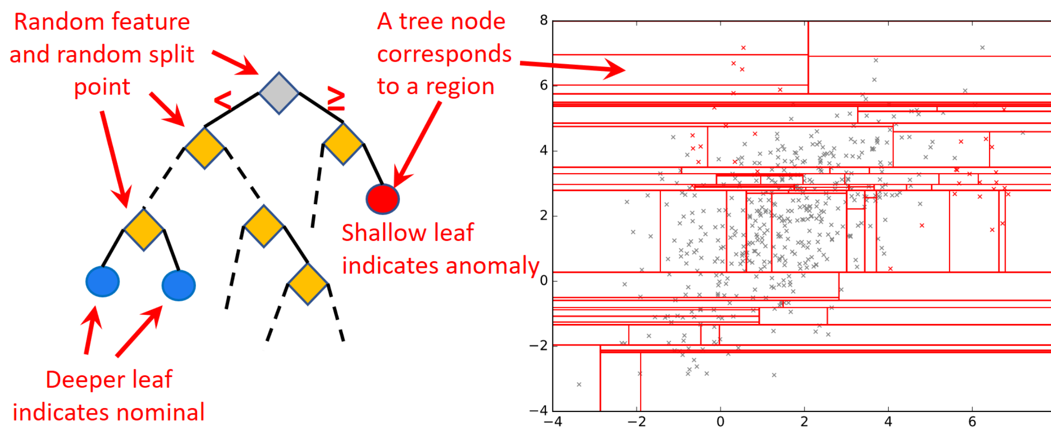


Figure 2: Isolation Forest tree structure and a set of corresponding region partitions for two dimensional space. Each node i.e. a region is associated with a weight that is adjusted after feedback.

Data Views. Our approach is based on defining a set of data views, which give a mechanism for system analysts to express their system/security knowledge. Each view computes a set of features for a class of system entities. Intuitively, we want to select a set of views such that malicious system entities will appear anomalous with respect to one or more of the views. By monitoring such views, anomaly detectors have the potential to identify the malicious entities.

More formally a view $v = (\phi_v, f_v)$ is a pair, where ϕ_v is an *entity filter*, which defines a class of system entities (e.g. all downloaded files) and f_v is a feature function, which returns a set of numeric features for any entity that passes the filter. For example, a simple view might use a filter that selects downloaded files and compute features of those entities that record statistics about certain event types such as permission changes and executions. A file that is downloaded and prepared for execution as part of an attack may appear anomalous under such a view.

For the system described in this paper, we created a set of 20 views over three major type of entities: processes, files and netflows. A summary of these view is given in table 1. Each entity type has a master view that contains all the features related to that entity type and all other views for the entity type contain a subset of features from the master view based on the specific intent of the view.

Anomaly Detection for Individual Views. Our system monitors each view with its own anomaly detector. In this work, we employ the anomaly-detection algorithm *Isolation Forest (IF)* (Liu, Ting, and Zhou 2008), which has been shown to be a robust and state-of-the-art approach in recent benchmark studies (Emmott et al. 2013). The input to IF is a set of entities that are each described by a feature vector. The output is an anomaly score for each entity that allows entities to be ranked by their anomalousness, as described in Section 3, for each view the ingestor computes, in real-time, the set of feature vectors for each instance that is part of a view. IF is an extremely fast algorithm and can process millions of instances in seconds, so is not a bottleneck in the processing pipeline.

The key idea behind IF is to construct a forest of randomized trees that each divide the feature space of a view into hyper-rectangular regions. Figure 2 illustrates a single IF tree, where each internal node of the tree splits data using a random threshold for a randomly selected feature. In this way, each node of the tree represents an hyper-rectangular region of input space. Each IF tree is grown until the leaves contain individual entities, or in other words, until all the entities have been isolated from other entities into their own regions. The anomaly score assigned to an entity for a single tree is the *negative* depth of the leaf node at which it becomes isolated. Thus, nodes that become isolated at shallower leaves will be ranked as more anomalous. Intuitively, this makes sense since entities that are anomalous compared to other entities in a view will be easier on average to isolate from others via randomly selected tests in the tree. The overall anomaly score assigned to an entity by an IF is the average over anomaly scores of each IF tree. In our system, we currently use IFs with 100 trees.

View Combination. By applying IF to each view in our system, we get a set of anomaly rankings over the entities in each view. If the analyst knew beforehand which views were going to be most useful for detecting malicious entities in an upcoming attack, then only high-ranking entities under those views would need to be investigated. Unfortunately, such hindsight information is not available and it will typically be unclear to an analyst which of the many views to pay attention. For this reason, our system combines the anomaly rankings across views into a single overall ranking over entities that the analyst can use to focus his investigation.

There has been prior work on combining multiple anomaly detection results, e.g. (Rayana and Akoglu 2016; Aggarwal and Sathe 2015; Zimek, Campello, and Sander 2014; Zimek et al. 2013; Aggarwal 2013). These methods are based on different combination principles, often related to identifying rankings that maximally agree with the individual rankings in some sense. In our own benchmark evaluations, however, using the wide set of benchmarks from prior work (Emmott et al. 2013), we have found that no one method is consistently superior. Further, we found that simple averaging tends to be as good or better than more

complex mechanisms in terms of overall benchmark performance. Thus, the default combination method in our system is to use a simple averaging approach. In particular, for each system entity, we compute its overall anomaly score as the average of the anomaly scores it is assigned for each view that it is part of. We did find that averaging tends to be more robust than combining using the max over anomaly scores.

5 Combining Views via Analyst Feedback

We saw that using fixed combination strategies such as averaging or more complex methods can often result in relatively large false positive rates for certain hosts, which extends the time needed to identify attacks. We hypothesize that this is due to the fact that for a particular host with unique usage patterns, there will often be many views that are not useful for detecting malicious entities on that host. For example, certain views will be prone to generate many false alarms on a particular host due to the behavior of benign entities that appear anomalous in those views. This suggests that one approach to improve detection performance is to be able to customize a multi-view anomaly detector for a particular host system based on feedback from a system analyst.

More specifically our system will operate in rounds, where on each round the top ranked overall entity is shown to an analyst. The analyst provides feedback about whether the entity appears to be benign, or whether it is possibly (or definitely) malicious. After each round, the overall anomaly ranking is adjusted in a way that aims to move benign entities lower in the ranking and malicious entities to the top. The goal is to allow for the analyst to identify a malicious entity in as few rounds as possible as well as maximizing the number of malicious entities found over the feedback rounds. Below, we describe our approach to incorporating feedback into tree-based anomaly detectors such as IF. We first specify the feedback mechanism as it may apply to individual views/detectors and then trivially extend it to combine multiple views.

Feedback for Tree-based Anomaly Detectors. IF is one member of a class of state-of-the-art anomaly detectors(Liu, Ting, and Zhou 2008; Tan, Ting, and Liu 2011; Chen, Liu, and Sun 2015; Wu et al. 2014) based on building forests of randomized trees. Interestingly, as pointed out in recent work (Das et al. 2016; 2017), all of these methods can be captured by a generic tree-based anomaly detection framework. In particular, this framework builds a forest of randomized trees in the fashion of IF and for each tree node n in the forest assigns a numeric weight w_n. Given this forest, an entity e can be passed through each tree, yielding a path from the root to a unique leaf in each tree based on the e's feature values. Let $N(e)$ denote the set of tree nodes along those paths. The anomaly score assigned to e, denoted $S(e)$ is then given by summing the weights of nodes in $N(e)$:

$$S(e) = \sum_{n \in N(e)} w_n.$$

Using this definition of $S(e)$, we can replicate the anomaly rankings produced by IF by simply setting $w_n = -1$ for all tree nodes. To see this, note that for this weight setting, the anomaly score for an instance is equal to the negative sum of the path lengths followed in each tree, which is directly proportional to the average isolation depth. Thus the instance with highest anomaly score corresponds to the instance with lowest average isolation depth. By setting the weights to different values one can arrive at other types of tree-based anomaly detection algorithms.

Our system initializes the anomaly detectors with weights of -1 so that IF is the default anomaly detector. However, feedback is used to adjust the weights in a way that aims to improve the anomaly ranking over time. Recent work formulated the problem of adjusting the weights based on feedback as a non-convex optimization problem (Das et al. 2017). Unfortunately, solving this problem is computationally expensive and does not scale well with the number of entities under consideration. Thus, a key contribution of this paper is to design a novel feedback mechanism that is extremely light weight and at the same time performs as well or better than the prior optimization-based approach.

Our feedback algorithm is a variant of the classic perceptron algorithm for classification problems (Novikoff 1962). At each feedback iteration, the top ranked anomalous entity e^* is shown to the analyst based on the current weights in the forest. The analyst then labels e^* as either "benign" or "malicious" and we let the entity label be $l^* = -1$ and $l^* = 1$ respectively. After receiving the feedback, for each node in $n \in N(e^*)$ we update the weights of those nodes according to:

$$w_n := \begin{cases} w_n - 1, \text{if } e^* \text{ is "benign"}, \\ w_n + 1, \text{if } e^* \text{ is "malicious"} \end{cases} .$$

Intuitively, if e^* is a benign entity, then this update will decrease the overall anomaly score assigned to e^* and more importantly decrease the scores of similar entities that traverse tree nodes in common with e^*. Conversely, if e^* is malicious, then the update will increase the score of e^* and also increase the score of similar entities, making them more likely to rise to the top of the ranking. Unlike the traditional perceptron algorithm, which only updates weights upon a mistake (in our case a false positive), our algorithm updates the weights at each feedback iteration. We found that this approach is significantly more effective when anomalies occur in small clusters.

To illustrate our approach, Figure 3 shows results on several anomaly detection benchmarks from (Das et al. 2017) that compare our perceptron-style algorithm, the previous state-of-the-art AAD (Das et al. 2017) algorithm, and an IF baseline that does not take feedback into account. Each graph shows 100 feedback iterations where the top ranked entity is presented to a simulated analyst and the resulting label is used to update the anomaly detector (or ignored by the pure IF). Each graph illustrates the number of true anomalies found versus the number of feedback iterations. An ideal results would be a line of slope one, which indicates that the anomaly detector never shows the analyst a false positive. It it clear from Figure 3 that the perception update is at least as good as the AAD algorithm and often significantly better than the IF baseline, which does not use feedback. In addition the perceptron method is very simple, fast and doesn't

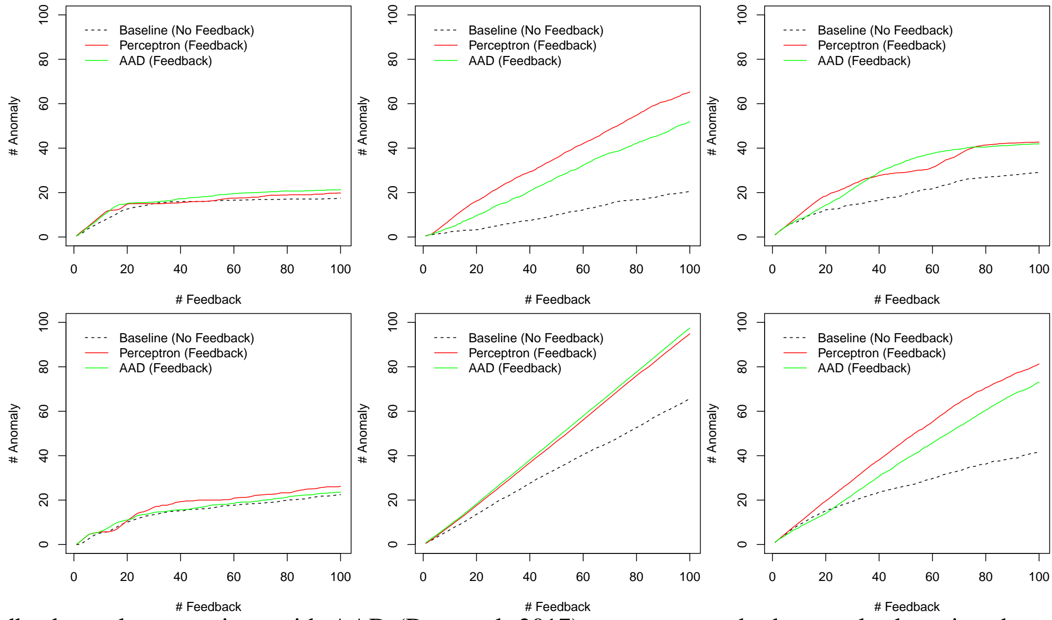


Figure 3: Feedback result comparison with AAD (Das et al. 2017) on some standard anomaly detection datasets (Das et al. 2017): Abalone, ANN-Thyroid-1v3, Cardiotocography, Yeast, Covtype and Mammography (left to right and top to bottom)

require any parameter tuning (unlike AAD). For this reason, for our cybersecurity experiments we focus exclusively on using the perceptron method for incorporating feedback.

Incorporating Feedback with Multiple Views. Incorporating feedback for combining multiple views is nearly identical to incorporating feedback into individual views. For each view, we create a dedicated forest of trees with their own weights and initialize all weights to -1 in order to simulate IF. Given an entity e, the anomaly score is simply equal to the average anomaly score assigned to it by views that it is part of. When we receive feedback on the top ranked entity e^*, we simply use the percptron algorithm to update the tree weights for views that e^* is part of. Intuitively, this updating approach will simultaneously adjust the way that views are combined and improve the anomaly detectors of individual views. In particular, if a view is generally not useful, in the sense that it produces many false positives, the weights in its corresponding trees will be continually decreased, making it less influential compared to other views.

6 Experiments on Red Team Attacks

Attack Datasets. The datasets for our experiments are based on audit logs that were generated as part of attack campaigns carried out by a red team as part of the DARPA Transperent Computing program. We collected data from three different hosts: two running the FreeBSD server operating system and one was running the Ubuntu desktop operating system. Data for two of the hosts (one FreeBSD and one Ubuntu) was collected over a three day period and the third over a five day period. During the data collection period all hosts were running benign activities resembling normal workloads. On each host, the red team executed an unknown attack campaign, which started at a time that was unknown to us. Table 1 reports the number of system entities of type event, process, file, and netflow that are observed on each host during the data collection and the number of those entities that were involved in the attack. It is clear that the number of malicious entities is a very small fraction of the total system entities.

After the attacks were completed and our results reported to the red team, the red team released a description of each attack scenario, which outlined the key entities and events. We used the descriptions in order to produce a ground truth data level encoding of the attacks, where each entity and event was labeled as being part of the attack or not. We use this ground truth to evaluate our approach and also to simulate feedback provided by a system analyst. In particular, when our simulated analyst is queried about an entity by our system, the simulated analyst used the ground truth in order to answer the query. We note that this is clearly an idealized simulation of an analyst, since analysts will not always be perfect. We consider evaluation of our approach under different noise models of analysts as interesting future work.

Because our tree based anomaly detection framework involves randomization, due to the randomized construction of the forests, we repeat each experiment on the collected data 10 times and report averaged results.

Result for Individual Views. We first consider applying our approach to each of the individual views. The goal is to observe the relative effectiveness of each view for detecting malicious entities. Figure 4 shows the result on a selected set of representative views from Host1 and Host2. Each graph the plot shows the number of malicious entities discovered by our approach versus the number of feedback iterations. Here we consider a malicious entity to be discovered if it was the top ranked entity that was presented to the analyst at a particular iteration. The dashed black line shows the baseline anomaly detection results for IF, which doesn't use any feedback i.e. a single ranking is computed at the start and

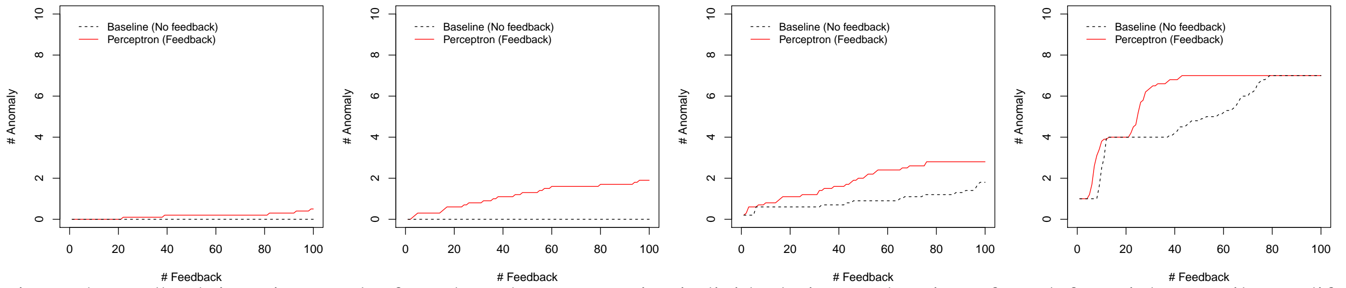
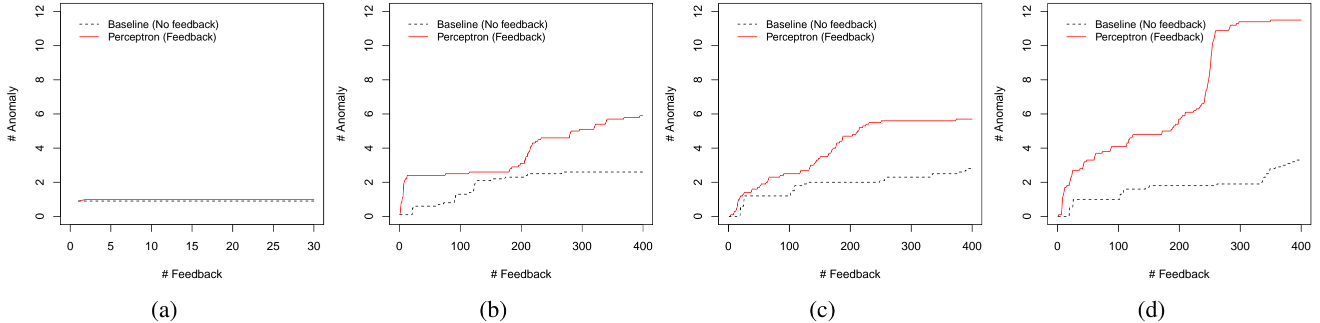


Figure 4: Feedback iteration results for selected representative individual views. The views from left to right are File Modify Event (Host1), Exfil Staging File (Host2), Process Process Events (Host1), and Beaconing Behavior (Host2).



(a) (b) (c) (d)

Figure 5: Feedback iteration results when applied to selected view combinations. From left to right: (a) combining only netflow views (Host1), (b) combining all 20 views (Host1), (c) combining all process views (Host2), (d) combining all 20 views (Host2).

the analyst is shown entities in the order of that fixed ranking. The red curve shows the performance achieved when using analyst feedback via the perceptron-style weight updates. Recall that these curves are averaged over 10 runs of the algorithm. We can see that in some views, for example, Process Process Events, Beaconing Behaviour and Netflow-related File Anomaly, the feedback improves significantly over baseline and in others both of the baseline and feedback performs poorly. This indicates, as we should expect, that some views are not useful for detecting entities in these attacks, or at least none of our approaches are able to use them effectively. We have no way to tell beforehand which view will be effective on a particular host system. Thus, without combining views one would need enough analyst resources to monitor each view, which is unlikely in practice.

Combining Views via Feedback. We now evaluate our view combination approach with and without feedback, using different grouping of views to be combined. We considered four sets of views to be combined: 1) All 20 views over all system entities, 2) Only Process views, 3) Only Netflow views, and 4) Only File views. By considering these groups we can observe the relative importance of each entity type to the detection and whether we are able to effectively combine views across entity types. Figure 5 shows graphs for selected combinations and hosts. With the exception of the "Only Netflow" results on Host1 we see that using feedback is able to significantly increase the rate that malicious entities are discovered. We also see that the results for "All views" produces higher discovery rates compared to "Only Netflow" or "Only Process", which indicates that combining across views can be effective.

Table 2 provides a more comprehensive set of results. In this table we give results for each of the 4 view groupings on each of the hosts. In particular, we report the average number of iterations until the first attack entity is discovered and also the average number of attack entities that were discovered after 50, 100, and 400 rounds of feedback. In each case, we give results for both the IF baseline, which uses no feedback, and our approach that uses feedback.

The first observation is that incorporating feedback performs significantly better than the IF baseline in almost all cases. Feedback allowed for the first attack entity to be discovered significantly faster and also discovered more attack entities for each number of feedback iterations.

A second observation is that we see our feedback approach is quite effective at combining views. In particular, the approach is able to discover more anomalies for each number of iterations by combining all views, compared to just combining views related to a single entity type. We also see that when restricted to a single entity type, the number of feedback rounds needed to find the first attack entity varies significantly. For example, on Host2 by combining just netflow views we are able to find the first attack entity in 1.6 rounds on average, versus not being able to find any attack entity using a combination of just the file views. By combining all views, we see that the time required to find the first anomaly is much closer to the minimum time across the individual entity types compared to the maximum time. This indicates that even when some views are quite bad for detection on a particular host, our combination approach is not significantly hurt by their presence and can exploit the views that are more effective.

Table 2: Summary result for all view combinations (averaged over 10 runs). H1 refers to Host1 and similar for H2 and H3. We run feedback upto 1000 iterations maximum, so in the case of file entity type 1001 means no attack entities were discovered.

EntityType	Algorithm	# of Feedback untill 1st attack entity			# of attack entities discovered within								
					50 feedback			100 feedback			400 feedback		
		H1	H2	H3	H1	H2	H3	H1	H2	H3	H1	H2	H3
All	Baseline	57.7	90.3	42.2	0.6	1	1.4	1.3	1	2.9	2.6	3.3	3.9
	Perceptron	3	7.4	22.2	2.4	3.3	3	2.5	4.1	4	5.9	11.5	5
Netflow	Baseline	4	19	31	0.9	0.4	0	NA					
	Perceptron	1.1	1.6	24.1	1	1.3	0.7	NA					
File	Baseline	13.5	1001	1001	1	0	0	2	0	0	3	0	0
	Perceptron	12.9	1001	1001	2	0	0	2	0	0	3	0	0
Process	Baseline	63.5	55.2	1001	0.4	1.2	0	1.3	1.2	0	2.8	2.8	0
	Perceptron	4.8	34.1	571.3	1.6	1.7	0.1	1.6	2.5	0.3	4.8	5.7	0.5

7 Summary

We developed a system that can work across different platforms to detect malicious system entities using audit data of a target host system. We employed view based anomaly detection along with feedback and showed that it helps to find attack entities more quickly using feedback than when feedback is ignored. To the best of our knowledge this is the first time that such feedback has been incorporated into an anomaly detection system for reducing the false positive rate in a cybersecurity setting. Moving forward we intend to continue evaluating our approach on ever more complex attacks, which will likely require development on new views and other analysis techniques. Another critical component moving forward is to study interfaces that best support a system analyst in investigating proposed entities, which will involve providing the analyst with clear explanations about why our system believes an entity is potentially malicious.

Acknowledgement

This work is supported by DARPA under contract number W911NF-11-C-0088. Any opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the DARPA.

References

Aggarwal, C. C., and Sathe, S. 2015. Theoretical foundations and algorithms for outlier ensembles. *SIGKDD Explor. Newsl.* 17(1):24–47.

Aggarwal, C. C. 2013. Outlier ensembles: position paper. *ACM SIGKDD Explorations Newsletter* 14(2):49–58.

Chen, F.; Liu, Z.; and Sun, M.-t. 2015. Anomaly detection by using random projection forest. In *2015 IEEE International Conference on Image Processing (ICIP)*, 1210–1214.

Das, S.; Wong, W.-K.; Dietterich, T. G.; Fern, A.; and Emmott, A. 2016. Incorporating expert feedback into active anomaly discovery. In *Proceedings of the IEEE ICDM*, 853–858.

Das, S.; Wong, W.-K.; Fern, A.; Dietterich, T. G.; and Siddiqui, M. A. 2017. Incorporating feedback into tree-based anomaly detection. *arXiv preprint arXiv:1708.09441*.

Dong, B.; Chen, Z.; Wang, H. W.; Tang, L.-A.; Zhang, K.; Lin, Y.; Li, Z.; and Chen, H. 2017. Efficient discovery of abnormal event sequences in enterprise security systems. In *The ACM International Conference on Information and Knowledge Management (CIKM). Pan Pacific, Singapore.*

Emmott, A. F.; Das, S.; Dietterich, T.; Fern, A.; and Wong, W.-K. 2013. Systematic construction of anomaly detection benchmarks from real data. In *Proceedings of the ACM SIGKDD workshop on outlier detection and description*, 16–21. ACM.

Forrest, S.; Hofmeyr, S. A.; Somayaji, A.; and Longstaff, T. A. 1996. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, 120–128. IEEE.

Gao, D.; Reiter, M. K.; and Song, D. 2004. Gray-box extraction of execution graphs for anomaly detection. In *Proc. of the 11th ACM conference on Computer and communications security*, 318–329.

Haller, P., and Odersky, M. 2006. Event-based programming without inversion of control. In Lightfoot, D. E., and Szyperski, C. A., eds., *Lecture Notes in Computer Science*, volume 4228. Springer.

Liu, F. T.; Ting, K. M.; and Zhou, Z.-H. 2008. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, 413–422. IEEE.

Novikoff, A. B. 1962. On convergence proofs on perceptrons. In *In Proceedings of the Symposium on the Mathematical Theory of Automata, volume 12*, 615622.

Rayana, S., and Akoglu, L. 2016. Less is more: Building selective anomaly ensembles. *ACM TKDD* 10(4):42.

Sekar, R.; Bendre, M.; Dhurjati, D.; and Bollineni, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, 144–155. IEEE.

Shu, X.; Yao, D.; and Ramakrishnan, N. 2015. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 401–413. ACM.

Tan, S. C.; Ting, K. M.; and Liu, T. F. 2011. Fast anomaly detection for streaming data. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence - Volume Two*, 1511–1516.

Wu, K.; Zhang, K.; Fan, W.; Edwards, A.; and Philip, S. Y. 2014. Rs-forest: A rapid density estimator for streaming anomaly detection. In *ICDM, 2014 IEEE International Conference on*, 600–609.

Zimek, A.; Gaudet, M.; Campello, R. J.; and Sander, J. 2013. Subsampling for efficient and effective unsupervised outlier detection ensembles. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 428–436.

Zimek, A.; Campello, R. J.; and Sander, J. 2014. Ensembles for unsupervised outlier detection: challenges and research questions a position paper. *Acm Sigkdd Explorations Newsletter* 15(1):11–22.