Towards Self-Adaptive Anomaly Detection Sensors

Gabriela F. Ciocarlie

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2009

ABSTRACT

Towards Self-Adaptive Anomaly Detection Sensors

Gabriela F. Ciocarlie

Spurred by the ever growing availability of online services and resources, threat models are constantly evolving. As a result, the same security techniques that were sufficient a decade, or even a few years ago, can prove inadequate today. In particular, recent advances in polymorphic attacks and the increasing volume of zero-day attacks threaten to overwhelm signature-based defense mechanisms. As attackers are finding new ways to gain access to networks and systems, so defense mechanisms must find new ways to protect them.

Anomaly Detection (AD) sensors provided a breakthrough in the defense against polymorphic and zero-day attacks by relying on models of normal behavior, rather than signatures of malicious input. However, as AD-based approaches are increasingly introduced as first-class defensive techniques, a number of open problems regarding their deployment and maintenance remain. The efficacy of AD sensors depends heavily on the quality of the data used to train them, and artificial or contrived training data may not provide a realistic view of the deployment environment. Most realistic data sets contain a number of attacks or anomalous events, their size making manual removal of attack data infeasible. As a result, sensors trained on this data can miss attacks and their variations. Another roadblock on the way to widespread adoption of AD sensors is that their deployment and maintenance often require significant intervention by a human expert, to manually optimize their performance and keep them up-to-date with changes in the system.

In this thesis we attempt to address these challenges by introducing a set of methods for self-sanitizing, self-calibrating and self-updating AD sensors. Our overall goal is to introduce a general framework for a class of AD sensors that can automatically adapt to the system under protection, combining detection performance with ease of deployment and operation.

We begin by extending the training phase for a class of content-based AD sensors to include a novel sanitization phase that significantly improves the detection performance of these sensors, in a manner agnostic to the underlying AD algorithm. This phase generates multiple models conditioned on small slices of the training data. We use these "micro-models" to produce provisional labels for each training input, and we combine the micro-models in a voting scheme to determine which parts of the training data may represent attacks. Our results suggest that this phase automatically and significantly improves the quality of unlabeled training data by making it as "attack-free" and "regular" as possible in the absence of absolute ground truth.

We then study the performance issues that stem from fully automating the AD sensors' calibration and long-term maintenance. Our goal is to remove the dependence on human operators using an unlabeled, and thus potentially dirty, sample of incoming traffic. To that end, we propose to enhance the training phase of AD sensors with a self-calibration phase that can be employed in conjunction with the sanitization technique resulting in a fully automated AD maintenance cycle. These techniques can be applied in an online fashion to ensure that the resulting AD models reflect changes in the system's behavior which would otherwise render the sensor's internal state inconsistent. We verify the validity of our approach through a series of experiments where we compare the manually obtained optimal parameters with the ones computed from the self-calibration phase. Modeling traffic from two different sources, the fully automated calibration shows performance comparable to that obtained using optimal parameters. Finally, our adaptive models outperform the statically generated ones retaining the gains in performance from the sanitization process over time.

We also introduce a set of specialized approaches for updating AD models, which can be applied in the case where the AD sensor directly monitors individual system components, such as the file system or database. If the sensor is informed of any changes applied to these components, it can derive and apply a "model patch" which translates the changes in terms of the AD behavioral model. We exemplify this approach by showing how information from the file system and database can be used to efficiently update models of non-dynamic HTTP requests. We also discuss the effect of software patches on AD models, and investigate the feasibility of deriving model updates directly from the contents of the patch.

The race between the attacker and defender for gaining access to the protected system is one of skill, information and resources. The methods that we have discussed so far improve the performance of local AD sensors through better data analysis and training methods. However, a well-equipped attacker, armed with intimate information on the protected system as well as extensive resources, can still attempt training attacks that change the normal input patterns. To cope with this possibility, we extend our methodology to support sharing models of abnormal traffic among multiple collaborating sites. We show that if one of the sites is able to capture an attack in its abnormal model, all of the collaborators can benefit via model exchange. Our framework makes this possible by defining a number of model operations which can be implemented for a wide range of AD sensors.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Dedication text

Chapter 1

Introduction

In recent years, network attacks such as flash crowds, denial-of-service attacks, port scans and the spreading of worms and botnets have evolved into a significant threat for large-scale networks. This problem is compounded by the fact that, as current research indicates, signature–based network intrusion detection systems, traditionally used by many anti-virus solutions, are quickly becoming ineffective at identifying malicious traffic [72; 12; 93]. In particular, Song *et al.* [93] demonstrate the relative ease with which polymorphic attack engines can overwhelm signature–based detection methods.

Modeling normal behavior or content represents one of a small set of promising alternatives that have been proposed in response to the threat of zero-day attacks. This approach is defined as *anomaly detection* (AD), and it implies the construction of learning-based models that characterize the normal behavior of entities (*i.e.* system, user, *etc.*). We find this argument reasonable, especially since application developers do not operate in an adversarial way: they do not purposefully allow their software to accept widely differing sets of strings. In short, relying on anomaly detection sensors to discover 0-day attacks has become a necessity rather than an option.

Anomaly sensors have wide applicability, ranging from systems that classify network traffic content [49; 109; 7; 108; 94] to those that focus on sequences of system calls

[92; 106; 24; 70]. Anomaly detection sensors also employ different learning methods: *supervised learning* techniques (*e.g.* [94]) which use labeled instances as training data in order to build the normality model that characterizes the behavior of a system, and *unsupervised learning* techniques (*e.g.* [108]) that aim to build normality models without requiring labeled data. However, anomaly–based approaches are not perfect [34; 25; 107]. One of the most commonly cited difficulties is that their effective application requires *highly accurate* modeling of normal traffic; despite the best efforts of the research community, this process remains an open problem. For example, Taylor and Gates [34] point to the problem of polluted or unclean training data sets as a key roadblock to the construction of effective AD sensors. Specifically, "ground truth" for large, realistic data sets is extremely hard to determine, rendering supervised learning techniques difficult to deploy and unsupervised techniques difficult to verify.

In a related problem, the intrusion detection community lacks a collection of significant, real-world data sets to test and validate new intrusion detection algorithms. Although an effort to assemble such a collection was made almost a decade ago [54], the resulting data set was flawed in a number of ways [64], and there is a growing consensus that future experimental results based on this data set should be ignored. The community, however, is left without any acceptable replacement. As a result, researchers and customers cannot validate the work of other researchers or vendors, especially since placing real, large data sets into wide circulation may reveal sensitive information belonging to the organization kind enough to donate the data. The next best solution involves every organization maintaining a private extensive data collection. Laying aside the challenges involved in addressing the privacy concerns of individuals within the organization, the technical challenge of keeping this data set pristine is currently an open problem.

Ideally, an anomaly detector should achieve 100% detection accuracy, *i.e.,* true attacks are all identified, with 0% false positives, *i.e.* no normal data classified as abnormal. Reaching this ideal is very hard due to a number of problems. First, the

generated model can under-fit the actual normal traffic. Under-fitting means that the AD sensor is overly general: it will flag traffic as "normal" even if this traffic does not belong to the true normal model. As a result, attackers have sufficient space to disguise their exploit, thus increasing the amount of "false negatives" produced by the sensor. Second, and equally as troubling, the model of normal traffic can over-fit the training data: non-attack traffic that is not observed during training may be regarded as anomalous. Over-fitting can generate an excessive amount of false alerts or "false positives." Third, as mentioned before, unsupervised AD systems lack a measure of ground truth to compare to and verify against. The presence of an attack in the training data "poisons" the normal model, thus rendering the AD system incapable of detecting future or closely related instances of this attack. As a result, the AD system may produce false negatives. This risk becomes a limiting factor of the size of the training set [101]. Finally, even in the presence of ground truth, creating a single model representing the normal behavior that includes all non-attack data can result in under-fitting and over generalization.

1.1 Problem Statement

The current reliance on a human expert for providing training data is symptomatic of the difficulties faced when deploying AD sensors in the real world. Depending on the system and the type of input it receives, the operator might have to provide not only recent training data sets, but also operational parameters depending on the characteristics of expected traffic. Even assuming the initial training and calibration stages are complete, the need for user supervision is hardly reduced. Real-world systems typically exhibit dynamic changes in their behavior patterns, which can render the AD model generated during deployment obsolete. An AD sensor that operates with an outdated model may incorrectly classify new behavior as malicious or assert that old, incorrect behavior is normal. Even though most work on anomaly detection ac-

knowledges the value of keeping a model updated, relatively little attention has been paid to the general framework or the question of summarizing and communicating these changes to a sensor in a manner enabling precise and automated model updates.

Other roadblocks on the way to the adoption of mainstream AD sensors are the often cited aspects of high false positive rates and susceptibility to training attacks. These are in fact related problems: a highly restrictive sensor typically generates a high number of alerts, not all of them representing true attacks. However, a more permissive sensor is also more susceptible to targeted training attacks, where the input stream is manipulated by a dedicated attacker causing the sensor to eventually accept malicious input as part of its normality model. This places an additional burden on the human operator, who must find an optimal "sweet spot" between these conflicting desiderata. Such significant user input dependence can quickly render an AD sensor unusable as a commercial off-the-shelf software. It seems that, apart from facilitating the deployment and maintenance of AD sensors, additional mechanisms are needed to insure their performance level in extreme adversarial environments.

The problem that we address in this thesis is the difficulty of building deployable, hands-free AD systems that can adapt to the normal behavior patterns of the protected host or application, derive their operational parameters from the intrinsic properties of observed host behavior, monitor and adapt to legitimate changes in this behavior, and successfully identify attacks without creating an inflation of false alerts. Throughout these stages, our goal is to greatly reduce, or if possible eliminate altogether, the need for human expert supervision.

1.2 Contributions

Our starting point for addressing the problems enumerated above is the following key insight: by improving and automating the AD training phase and the quality of the resulting normality model, we can improve the performance of AD sensors and

increase their robustness and deployability. In this thesis we present our results which validate our hypothesis. We extend the training phase with a new sanitization phase that uses our novel micro-models in a voting scheme to automatically eliminate attacks and anomalies from training data. We then investigate new techniques which can be applied in order to fully automate the training process and therefore the deployment of AD sensors in the real world. As part of this approach, we identify the *false* false positive problem and propose a shadow sensor architecture for consuming false positives (FP) with an automated process rather than human attention. We also develop methods for maintaining a high performance level for AD sensors when the system under protection exhibits dynamic changes in its behavior. In addition, we expand the notion of robust automated AD sensors to include collaboration between multiple entities: hosts, devices and application. We also propose a novel distributed architecture in order to cross-sanitize AD models and remove training attacks that might otherwise bypass detectors with a strictly local view of normal behavior patterns.

Our high level goal is to augment AD sensors, increasing their robustness while automating their deployment and operation. It is important to note that our core techniques make only the following assumptions about the inner workings of each AD sensor:

- the sensor can be trained on a specified set of continuous data (*i.e.* a time-delimited slice of a data stream);
- based on the training set, the sensor models the content of the stream, creating a self-contained AD model;
- using the knowledge encapsulated in an AD model, the sensor can classify a new data point as either normal or abnormal;

We note that this framework can be applied at either the network, host, service or application levels. In our experimental evaluation, we address mostly network-

based sensors, but discuss the application of the same technique to host-based AD sensors as well. Throughout our work, we refer to detection and false alert rates as rates determined for a specific class of attacks that we observed in the tested data sets since, as we noted before, discovering ground truth for any realistic data set is currently infeasible.

In order to produce concrete, verifiable results we have implemented our methods using a number of current sensors with the above properties. We treat these sensors as black-box entities with well defined input and output interfaces according to our framework. We strive for a general AD automation framework, applicable in a wide range of environments using multiple AD solutions. We believe that this approach can be instrumental in obtaining significant improvements in the detection of 0-day attacks that can be implemented outside of the lab and in the real world.

1.2.1 Training Data Sanitization

The first technique that we have developed extends the AD training phase of an AD sensor (without applying any changes to the AD algorithm itself) to successfully **sanitize training data**, while achieving both a high rate of detection and a low rate of false positives. Instead of using a normal model generated by a single AD sensor trained on a single large set of data, we use multiple AD instances trained on small data slices. Therefore, we produce multiple normal models, which we call *micro-models*, by training AD instances on small, disjoint subsets of the original dataset. Each of these micro-models represents a very localized view of the training data. Armed with the micro-models, we are now in a position to assess the quality of our training data and automatically detect and remove any attacks or abnormalities that should not be considered part of the normal model. The threat model for this approach consists of persistent and/or targeted attacks, or other anomalies that persist throughout the majority of the training set.

1.2.2 Automated Deployment of AD Sensors

Based on the proposed sanitization mechanism, we take additional steps towards self-calibrating and self-sanitizing AD sensors. To that end, we propose a method for automatically **determining the operational parameters** needed during the deployment process based only on the intrinsic characteristics of the system under protection. This novel self-calibration phase complements the data sanitization process, and can enable true hands-free deployment for future AD sensors.

1.2.3 AD Model Updates

Another important aspect of anomaly detection models is that they have to **reflect the dynamic changes** that are exhibited in the system's behavior. We propose a self-updating mechanism where an AD sensor can adapt to these changes by integrating them into its normality models. We differentiate between two types of model updates:

- *progressive model update*, for which the changes are caused by external factors which cannot be controlled but where the effect of their actions can be observed;
- *induced model update*, for which the changes are caused by factors internal to the protected system, such as code patches, changes to the file system or modifications of the database.

In the case of progressive model updates, we propose to continuously update the sanitized model to reflect the dynamic changes in the network or users' behavior. We achieve this goal by using an online learning approach combined with an aging mechanism for micro-models. Our results show significant performance gains compared to the case of "static", or non-updating AD sensors. For induced model updates, we investigate specialized approaches that depend on the monitored system components (*i.e.* file system, database and software patches) that effect changes in the system behavior. The ultimate goal is to automatically derive and apply a "model patch"

that describes the changes necessary to update an AD behavioral model based on direct notifications provided by the three components listed above. In this thesis, we explore the basic framework for this approach, and present a feasibility study that aims to avoid extensive retraining and regeneration of an entire AD model when only parts may have changed.

1.2.4 AD Models Exchange

We continue by extrapolating the problem of improving, automating and updating AD models to include the notion of **AD model exchange** between collaborating systems. AD models are essentially represented as first class objects that can be manipulated and communicated among collaborators. The first application the we present uses our AD sanitization methods in a novel, distributed strategy that leverages the location diversity of collaborating sites. This approach, which we refer to as *cross-sanitization*, implies the exchange of models of abnormal/malicious behavior between trusted peers that can use this information to further improve their local normality models. Second, we propose applying our AD model exchange framework at the application level, and analyze the feasibility of a modeling technique for application behavior based on return value distributions.

1.2.5 Feedback Systems

An important aspect of our work regards the efficiency of the resulting AD systems in the real world. Many papers comment on anomaly detectors having too high a false positive rate, thus making them less than ideal sensors. We believe that the false positive rate is the wrong metric, as service throughput and latency are more important. In this thesis, we describe the use of a heavily instrumented host-based "shadow sensor" system akin to a honeypot that determines with very high accuracy whether an event deemed an attack by the AD system is a false positive or a real attack. Such systems perform substantially slower, usually orders of magnitude

slower, than the native, un-instrumented application [1]. Therefore, when taking into account oracle constraints, we focus on producing a sensor that identifies a few "suspect data" items which are subjected to further but time-expensive tests. This way the false positives do not incur damage to the system under protection, and do not flood an operational center with too many alarms. Instead, the shadow server processes both true attacks and incorrectly classified packets to validate whether a packet signifies a true attack. These packets are still processed by the intended shadowed application and only cause an increased delay for network traffic incorrectly deemed an attack.

Aside from using the shadow sensor feedback for coping with false alarms, we also propose the use of a second type of feedback mechanism. We consider a filtering approach that is applied before our sanitization mechanism. To that end, we label as abnormal the input for which which the system under protection gives an *error response*. This abnormal data is not considered for the sanitization process, but it is directly encapsulated in the abnormal model, which in turn leads to a reduction in the number of false alerts.

1.3 Organization

This thesis is organized as follows:

- **Chapter 1** has provided the general background of our work, placing it in the larger context of defending large scale network systems against 0-day attacks. It has also established the motivation behind instrumenting deployable, hands-free anomaly detection sensors.

- **Chapter 2** discusses the relevant work in the field of anomaly detection systems. It focuses mainly on AD algorithms for detecting evasions at different levels, such as the network and host levels. It also presents existing work that aims to address problems such as calibrating AD parameters or handling concept drift.

While this line of work has produced promising results, they are often limited in scope: the problems mentioned above are addressed in separation, and often related to a specific type of AD algorithm, without providing a complete and general framework towards fully automated robust AD sensors.

- **Chapter 3** lays the cornerstone of our work, by presenting the details of the *training dataset sanitization* process, along with an extensive evaluation of the technique. The results confirm that, by employing the sanitization technique, the detection rate of the studied AD sensors is improved, while the false positive rate is kept at a low level. This chapter also introduces the shadow sensor architecture that addresses the false positive problem. Without inflicting a significant computational overhead on the system under protection, the shadow sensor determines with high accuracy when an alert is a false positive or a real attack. Parts of this chapter were originally published in [15; 16].

- **Chapter 4** introduces two *automated deployment* techniques, that complement and extend our dataset sanitization method, without sacrificing any of its generality. This work shows how the operational parameters of an AD sensor can adapt to the particular characteristics of the system under protection in order to achieve a fully automated training phase. It also presents a comparison between an AD system operating with empirically determined optimal parameters and one using our automated calibration methods. This work will appear in the proceedings of a conference [17].

- **Chapter 5** relates to the problem of *AD model updates* and proposes two different approaches. First, it demonstrates the use of an online learning technique which is applied in conjunction with the sanitization process. Then it proposes a specialized approach that depends on feedback from three integral parts of the system under protection: file system, database and software patching. A complete evaluation is conducted for the first approach, and feasibility stud-

ies along with a baseline evaluation for the file system and database feedback systems are presented for the second. This chapter is based on initial results published or under review in [14; 17; 56; 96]

- **Chapter 6** details the use of *AD model exchange* in a collaborative security setting. It presents two model exchange applications for which the security protection mechanism leverages the location diversity of collaborating sites in order to improve its performance. The first application, *cross-sanitization*, exchanges abnormal/malicious models in order to increase the resistance of local AD sensors to training attacks, while the second one uses application profiles to study the feasibility of a return value based modeling technique. This chapter also introduces a novel formalized set of AD model operations that can be applied to a wide range of algorithms based on the collaborative model exchange paradigm. Parts of this work originally appeared in [15; 58]

- Finally, the thesis concludes with **chapter 7**, which summarizes our contributions and also identifies some of the most important directions for future work.

Chapter 2

Related Work

2.1 Anomaly Detection

Anomaly-based classification provides a powerful method for security mechanisms in contrast to signature-based approaches which have been shown to be easily evaded. First, Polygraph [72] analyzed the case of polymorphic worms, showing that substring signatures are insufficient and proposing different types of signatures to detect such attacks. Then, Crandall *et al.* [12] performed an evaluation that revealed the fact that single continuous byte string signatures are not effective for content filtering, and token-based byte string signatures composed of smaller substrings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used. Finally, Song *et al.* [93] presented a quantitative analysis of the strengths and limitations of shellcode polymorphism along with empirical evidence that showed that modeling the class of self-modifying code is likely intractable by known methods, including both statistical constructs and string signatures.

In this chapter we will present the state-of-the-art in anomaly detection literature, focusing on computer security mechanisms and identifying related approaches that are most relevant to the work proposed in this thesis. There are two types of sensors

that are relevant to our work: those that classify network traffic content (network-based anomaly sensors) and those that classify sequences of system calls (host-based anomaly sensors).

2.1.1 Network-based Anomaly Detection

Network-based anomaly detection systems can operate at different levels: they can detect anomalies directly in the network packets, or they can reconstruct the content flows to extract more information about the protected network. Initial work towards anomaly detection at the packet level computed models related to header information. For example NIDES [40], SPADE [38] and ADAM [3] proposed models that characterized the normal distributions of IP addresses and ports, while later PHAD [61] learned the normal ranges of values for each packet header field at the data link (Ethernet), network (IP), and transport/control layers (TCP, UDP, ICMP). NETAD [62] introduced modeling of the first 48 byte values in a packet which, in the case of TCP data, also includes 8 bytes of payload.

The full transition to payload based anomaly detection at the packet level was done by PAYL [110; 108], which models 1-gram frequency distributions for a given packet length. Later Anagram [109] introduced the modeling technique of a mixture of high-order n-grams (n $>$ 1) designed to detect anomalous network packet payloads while exhibiting resistance to mimicry attacks. Anagram uses Bloom filters (BF)[6] to store n-grams of normal and malicious packet payloads. Depending on the ratio of normal and anomalous n-grams in the BF, incoming packets are deemed either normal or abnormal. The use of Bloom filters makes this method computationally efficient and more general, as opposed to the approach proposed by Rieck and Laskov [85], which stores the n-grams in a tree data structure for individual connections. In the detection phase they compare the test data against the pre-computed trees.

When network packets are reconstructed, more features can be extracted compared to packet content inspection. Systems such as Emerald [80], NetSTAT [104]

and Bro [78] reconstruct network packets to extract information related to session duration, service type, bytes transferred, *etc.* in order to detect network attacks. A similar approach to Payl, proposed by Kruegel *et al* [49], describes a service-specific intrusion detection system that works at the flow level. They combine the type, length and payload distribution of web requests as features in a statistical model to compute an anomaly score of a web request. Another proposed approach, Spectrogram [94], uses n-gram modeling in the form of mixture of Markov chains to defend against web-layer code-injection attacks.

2.1.2 Host-based Anomaly Detection

Host based AD sensors provide one mechanism for detecting the presence or activation of malware on a host by observing a shift in an application's execution profile. The seminal work of Hofmeyr, Somayaji, and Forrest [39; 92] helped initiate application behavior profiling at the system call level. This interface represents the services that malcode, once activated, must use to effect persistent changes and other forms of I/O. In particular, malware may begin to use system services that the application has not previously invoked, or may employ the set of already–used services in new ways (*e.g.,* via new arguments to those calls[70]). Such information is now easy to collect; the `strace` and `ltrace` tools for Unix perform exactly this task.

Wagner and Dean [106] proposed a static analysis approach to automatically derive a model of application behavior. Their approach assumes that a model of expected application behavior is pre-computed statically from the source code and the program is monitored to check its system call trace against the model at runtime. BlueBox [11] uses system call introspection to create an infrastructure for defining and enforcing fine-grained process capabilities in the kernel, specified as a set of rules. Systrace [81] also supports fine grained process confinement while eliminating the need to run programs in privileged process context. Policies are generated automatically in a training session or are generated during program execution. VtPath [24] utilizes return ad-

dress information extracted from the call stack to build an anomaly detection system. This approach generates the abstract execution path between two program execution points, and decides whether this path is valid based on what has been learned on the normal runs of the program. Another approach that builds an execution graph without requiring static analysis of the source code or binary and conforms to the control graph of the program was proposed by Gao *et al.* [32]. An execution graph model is defined as a model that accepts the same system call sequences as would be accepted by a model built from the control flow graph of the program, with the limitation that this type of models can train only on observed runs of the program which may miss entire branches of the program that static analysis would uncover. Giffen *et al.* [36] proposes a hybrid approach that incorporates in a model of a program's binary code knowledge of the environment in which the program runs, and increases the accuracy of the models with a dataflow analysis algorithm for context-sensitive recovery of static data.

An overall observation is that, for both network-based and host-based sensors, effective anomaly detection requires highly accurate modeling of normal data — a problem that is the focus of this thesis. Fogla and Lee [25] showed how some payload-based network anomaly detection systems can be evaded by polymorphic blending attacks, while Wagner and Soto [107] developed a theoretical framework for evaluating the security of host-based anomaly detection systems against mimicry attacks. Taylor and Gates [34] noted the problem of polluted or unclean training data sets as a key roadblock to the construction of effective AD sensors, as "ground truth" for large, realistic data sets is extremely hard to determine. For this reason we propose an unsupervised method for sanitizing training datasets both at local and distributed levels. In our experiments we focus mostly on network-based sensors, but discuss the application of the same techniques to host-based AD sensors as well.

2.2 Ensemble and Meta-learning Methods

Our approach for sanitizing training datasets shares elements with ensemble methods, which are reviewed by Dietterich [22]. It is important to note that, while most of these methods traditionally fall in the category of supervised learning algorithms, due the applications of our work (*e.g.* real network traffic), we are forced to use unlabeled training data. In particular, we construct a set of classifiers and then classify the new data points using a (weighted) vote. We generate AD models from slices of the training data, thus manipulating the training examples presented to the learning method. Another similar machine learning approach is that of Bagging Predictors [8], which uses a learning algorithm with a training set that consists of a sample of training examples drawn randomly for an initial data set. The *cross-validated committees* method [76] proposes to construct a training model by leaving out disjoint subsets of the training data. ADABoost [27] generates multiple hypothesis and maintains a set of weights over the training example. Each iteration invokes the learning algorithm to minimize the weighted error and returns a hypothesis, which is used in a final weighted vote.

Probably the most similar work to ours is MetaCost [23], which is an algorithm that implements cost-sensitive classification. Instead of modifying an error minimization classification procedure, it views the classifier as a black box, the same as we do, and wraps the procedure around it in order to reduce the loss. MetaCost estimates class probabilities and relabels the training examples such that the expected cost of predicting new labels is minimized. Finally it builds a new model based on the relabeled data. Our algorithm also labels the training examples in the voting process and uses them accordingly in the training phase.

The application of ensemble methods for computer security applications was introduced by JAM [97]; this seminal work focuses on developing, implementing, and evaluating a range of unsupervised learning strategies and combining techniques for fraud detection systems. It also presents methods for "meta-learning" or computing

sets of "base classifiers" over various partitions or sampling of the training data. The combining algorithms proposed are called "class-combiner" and "stacking" and they are built based on work presented by Chan and Stolfo [10] and Wolpert [111]. The objective of both "combiner" and "stacking" is to improve the overall prediction accuracy by exploring the diversity of multiple learning algorithms through meta-learning. In our work we use the same learning algorithm while exploring the diversity of multiple training example sets. The idea of stacked generalization was further improved by Ting and Witten [102] proposing the use of class distributions rather than class predictions.

Our voting technique also resembles a method for determining the density estimation of an AD model. Yeung and Chow [113] previously proposed the use of non-parametric density estimation based on Parzen-window estimators [77] with Gaussian kernels in order to build an anomaly detection system. Their sensor receives data labeled as normal to estimate the density of a model while our approach does not require any prior labeling. However, their technique requires essentially no training time, but introduces a high computational demand during testing.

2.3 Automated Calibration

We define the "automated calibration" as the process of automatically determining anomaly detection parameters based on the intrinsic characteristics of the data that is modeled. Most AD sensors presented in the literature determine some of these parameters empirically or require them to be user defined [26; 92; 94]. However, Tan and Maxion [101] show that the choice of such parameters has a significant impact on the detection performance, thus it is very important to make an "optimal", or at least "reasonably good", choice.

Anagram [109] determines the stability of an AD model automatically based on the rate at which new content appears in the training data. pH [92] proposes heuristics for

determining an effective training time, minimizing human intervention, but requiring user defined thresholds. Payl [108] has a calibration phase for which a sample of test data is measured against the model and a threshold is chosen. The threshold is updated through a subsequent round of testing. Kruegel *et al.* [50] propose a web-based anomaly detection mechanism, which uses a number of different models to characterize the parameters used in the invocation of server-side programs. For these models, dynamic thresholds are generated in the training phase, by evaluating the maximum score values given on a validation dataset. EMT [99], a data mining system that computes behavior-based profiles of user email accounts, also proposes a calibration phase for the thresholds used in determining viral propagation. Two types of threshold settings are used: a threshold proportional to the standard deviation of the histograms representing the user profile, and a threshold based upon the changing of trend, both conditioned on a window size of prior histogram values. Ringberg *et al.* [86] talk about the difficulty of tuning parameters for PCA-based techniques for detecting anomalous traffic in IP networks and discuss the capability of anomalies to pollute the normal traffic. We note that our work presents a solution for data pollution while also offering a generalized method for calibrating the sanitization parameters.

2.4 AD Model Update

As the systems under protection progress and evolve over time, what may be considered as "normal" behavior can legitimately change over time as well. As a consequence, the AD models that encapsulate this behavior need to adapt to this phenomenon, referred to as *concept drift*. We define the *model update* as the process of adapting the AD models to legitimate changes in order to maximize the accuracy of the AD sensors. Our online sanitization benefits from the advantages of both sanitization and updating processes, in an unsupervised learning environment.

Most publications which propose updating the model after significant changes to the environment, data stream, or application use supervised learning techniques, such as the approach proposed by Gama *et al.* [31]. Methods of this type maintain an adaptive time window on the training data [43], select representative training examples [45], or weight the training examples [44]. The key idea is to automatically adjust the window size, the example selection, and the example weighting, respectively, so that the estimated generalization error is minimized. Consequently, these methods assume the existence of labeled data which is not the case for the applications that we are interested in analyzing. It seems that anomaly detectors would benefit from an additional source of information that can confirm or reject the initial classification, and Pietraszek [79] suggests using human–supervised machine learning for such tuning.

Our second approach for model update proposes to use feedback information from three sources that are part of the system under protection: file system, database and patch installation. For the patch approach, our feasibility study was validated by the work proposed by Li *et al.* [53] for gray-box system-call-based anomaly detectors(*i.e.* trained with system-call traces of the program when processing intended inputs). They present an algorithm by which an execution graph can be converted from the program for which it was originally trained to a patched version of that program.

For the file system and database approach we correlate HTTP requests and responses along with the SQL queries and replies that correspond to each HTTP pair. Maggi *et al.* [60] introduce a technique that addresses the problem of web application concept drift, by modeling both the HTTP requests and responses using the *webanomaly* [51] sensor. They parse the HTML documents that are returned as responses to extract links and forms and then they pass them to the anomaly detection sensor in order to update the models. As a result of this analysis, the anomaly detector is able to adapt to changes in the session structure resulting from the introduction of new resources, and also to changes in the request structure resulting from

the introduction of new parameters. Our approach is more general as it considers the input directly from the source of change: the system components themselves, as the response of the web server can be compromised by a sophisticated attacker.

2.5 AD Model Exchange

Our distributed strategy for AD model sanitization leverages the location diversity of collaborating sites to exchange abnormal models that can be used to improve each site's AD model. The exchange of abnormal models was also used in JAM [97] for commercial fraud detection. Fraud detection systems can be substantially improved by combining multiple models of fraudulent transactions shared among banks. We apply a similar idea in the case of network traffic content-based anomaly detection in order to alleviate the effects of training attacks. If the exchanged models have similar features with the local normality model, then the local model is recomputed, excluding the similarities which, in this case, represent bad behavior.

We also used the idea of model exchange in previous work [13], where the AD model acts as a profile of device behavior in a MANET environment. The model can be utilized by peers to determine its trustworthiness by comparing their mutual models exchanged between the devices. Rather than proving one's trustworthiness via a certificate, a credential, or a representation of "reputation" MANET nodes are authenticated by their behavioral profile of how they typically interact. Other nodes may validate the node by conformance to their own profiles, and to ensure the new node subsequently behaves in conformance with its announced profile. For this application, when the exchanged models are similar, they are aggregated to produce one unified view of the current MANET and to reduce false suspicions of anomalous behavior. This technique was later expanded into a more complete system for MANET access control [28].

Frias-Martinez *et al.* [30] also used the concept of model exchange to reduce false

positive rate in anomaly detection sensors. Collaborating security approaches often use sharing of alerts to detect distributed attacks, reducing the false positive rate as a consequence [95; 75; 89; 48; 73; 103]. However, these approaches are not optimal in the presence of isolated attacks that do not affect the whole network. For that case, Frias-Martinez *et al.* [30] proposed a cluster-based AD sensor that relies on clusters of behavior profiles to identify anomalous behavior. The behavior of a host raises an alert only when a group of collaborative host profiles with similar behavior (cluster of behavior profiles) detect the anomaly, rather than just relying on the host's own behavior profile to raise the alert.

Chapter 3

Training Dataset Sanitization

To introduce our training data sanitization technique, which forms the backbone of our framework for self-adapting AD sensors, we will start from two key challenges for anomaly detection systems: an under-trained (overly generalized) sensor, or an over-trained (exceedingly strict) sensor. These situations are particularly hard to cope with in practice, as unsupervised AD systems often lack a measure of ground truth to compare to and verify against. An additional, and equally troubling aspect regards the presence of an attack in the training data: such an occurrence can "poison" the sensor, thus rendering it incapable of detecting future or closely related instances of this attack. These problems appear to stem from a common source: the quality of the normality model that an AD system employs to detect abnormal data. This single and monolithic normality model is the product of a training phase that traditionally uses all data from a non-sanitized training data set.

We conjecture that in order to generate an accurate normal model, researchers must utilize an effective sanitization process for the AD training data set. To that end, removing all abnormalities, including attacks and other traffic artifacts, from the AD training set is a crucial first step. Supervised training using labeled datasets appears to be an ideal cleaning process. However, the size and complexity of training data sets obtained from real-world network traces makes such labeling infeasible. In

addition, semi–supervised or even unsupervised training using an automated process or an oracle is computationally demanding and may lead to an over-estimated and under-trained normal model. Even if we assume that unsupervised training can detect 100% of the attacks, the resulting normal model may contain abnormalities that should not be considered part of the normal model. These abnormalities represent data patterns or traffic that are not attacks, but still appear infrequently or for a very short period of time. For example, the random portion of HTTP cookies and HTTP POST requests may be considered non-regular and thus abnormal. This type of data should not form part of the normal model because it does not convey any extra information about the site or modeled protocol. Thus, in practice, both supervised and unsupervised training might fail to identify and remove from the training set non-regular data, thereby producing a large and over-estimated normal model. We introduce a new unsupervised training approach that attempts to determine both attacks and abnormalities and separate them from the regular, normal model.

3.1 Data Sanitization Using Micro-Models

We observe that for a training set that spans a long period of time, attacks and abnormalities are a minority class of data. While the total attack volume in any given trace may be high, the frequency of specific attacks is generally low relative to legitimate input. This assumption may not hold in some circumstances, *e.g.,* during a DDoS attack or during the propagation phase of a worm such as Slammer. We can possibly identify such non-ideal AD training conditions by analyzing the entropy of a particular dataset (too high or too low may indicate exceptional circumstances). We leave this analysis for the future. Although we cannot predict the time of an attack in the training set, the attack itself will manifest as a few packets that will not persist throughout the dataset. Common attack packets tend to cluster together and form a sparse representation over time. For example, once a worm outbreak

starts, it appears concentrated in a relatively short period of time, and eventually system defenders quarantine, patch, reboot, or filter the infected hosts. As a result, the worm's appearance in the dataset decreases [67]. We expect these assumptions to hold true over relatively long periods of time, and this expectation requires the use of large training datasets to properly sanitize an AD model. In short, larger amounts of training data can help produce better models — a supposition that seems intuitively reasonable.

We must be cautious, however, as having a large training set increases the probability that an individual datum appears normal (the datum appears more frequently in the dataset; consequently, the probability of it appearing "normal" increases). Furthermore, having the AD system consider greater amounts of training data increases the probability of malcode presence in the dataset. As a result, malcode data can poison the model, and its presence complicates the task of classifying normal data. We next describe how we use micro-models in an ensemble arrangement to process large training data sets in a manner that resists the effects of malcode content in that data.

3.1.1 Micro-model Definition

Our method of sanitizing the training data for an AD sensor employs the idea of "ensemble methods." Dietterich [22] defines an ensemble classifier as "a set of classifiers whose individual decisions are combined in some way (typically by weighted or unweighted voting) to classify new examples." Methods for creating ensembles include, among other actions, techniques that manipulate the training examples. Given our assumption about the span of attacks in our training set (see beginning of section 3.1), it seems appropriate to use time-delimited slices of the training data.

We employ the following strategy: consider a large training dataset T partitioned into a number of smaller disjoint subsets (micro-datasets):

$$T = \{md_1, md_2, \ldots, md_N\}, \tag{3.1}$$

where md_i is the micro-dataset starting at time $(i-1)*g$ and, g is the granularity for each micro-dataset.

We can now apply a given anomaly detection algorithm. We define the model function AD:

$$M = AD(T), \tag{3.2}$$

where AD can be any chosen anomaly detection algorithm, T is the training data set, and M denotes the model produced by AD for the give training set.

In order to create the ensemble of classifiers, we use each of the "epochs" md_i to compute a *micro-model*, M_i. $M_i = AD(md_i)$. We posit that each distinct attack will be concentrated in (or around) time period t_j affecting only a small fraction of the micro-models: M_j may be poisoned, having modeled the attack vector as normal data, but model M_k computed for time period t_k, $k \neq j$ is likely to be unaffected by the same attack. In order to maximize this likelihood, however, we need to identify the right level of time granularity g. Naturally, epochs can range over the entire set of training data. Our experiments analyze network packet traces captured over approximately 500 hours. We find that a value of g from 3 to 5 hours was sufficient to generate well behaved micro-models. In chapter 4 we will present a method for determining the granularity automatically.

3.1.2 Sanitized and Abnormal Models

Once the micro-models are built, they can be used, together with the chosen AD sensor, as a classifier ensemble: a given network packet, which is to be classified as either normal or anomalous, can be tested, using the AD sensor, against each of the micro-models. One possibility would be to apply this testing scheme to the same data set that was used to build the micro-models (we call this process *introspection*). Another option is to apply the micro-model testing to a second set of the initially available traffic, of smaller size. The ultimate goal is to effectively sanitize the training data set and thus obtain the clean training data set needed for anomaly detection.

Once again, we treat the AD sensor at a general level, this time considering a generic $TEST$ function. For a packet P_j part of the tested data set, each individual test against a micro-model results in a label marking the tested packet either as *normal* or *abnormal*:

$$L_{j,i} = TEST(P_j, M_i) \tag{3.3}$$

where the label, $L_{j,i}$, has a value of 0 if the model M_i deems the packet P_j normal, or 1 if M_i deems it abnormal.

However, these labels are not yet generalized; they remain specialized to the micro-model used in each test. In order to generalize the labels, we process each labeled dataset through a voting scheme, which assigns a final score to each packet:

$$SCORE(P_j) = \frac{1}{W} \sum_{i=1}^{N} w_i \cdot L_{j,i} \tag{3.4}$$

where w_i is the weight assigned to model M_i and $W = \sum_{i=1}^{N} w_i$. We have investigated two possible strategies: *simple voting*, where all models are weighted identically, and *weighted voting*, which assigns to each micro-model M_i a weight w_i equal to the number of packets used to train it. The study of other weighting strategies can provide an avenue for future research.

To understand the AD decision process, we consider the case where a micro-model M_i includes attack-related content. When used for testing, the AD may label as normal a packet containing that particular attack vector. Assuming that only a minority of the micro-models will include the same attack vector as M_i, we use the voting scheme to split our data into two disjoint sets: one that contains only majority-voted normal packets, T_{san} from which we build the sanitized model M_{san}, and the rest, used to generate a model of abnormal data, M_{abn}.

$$T_{san} = \bigcup \{P_j \mid SCORE(P_j) \leq V\}, \quad M_{san} = AD(T_{san}) \tag{3.5}$$

$$T_{abn} = \bigcup \{P_j \mid SCORE(P_j) > V\}, \quad M_{abn} = AD(T_{abn}) \tag{3.6}$$

where V is a voting threshold. In the case of unweighted voting, V is the maximum percentage of abnormal labels permitted such that a packet is labeled normal. Consequently, it must be the case that $1-V > N_p$, where N_p is the maximum percentage of models expected to be poisoned by any specific attack vector. We provide an analysis of the impact of this threshold on both voting schemes in the evaluation section (see the full architecture in figure 3.1).

After this two-phase training process, the AD sensor can use the sanitized model for online testing. Note that we have described a general approach to sanitization without resorting to the specific details of the AD decision process; it is enough that the AD sensor outputs a classification for each discrete piece of its input (*e.g.*, a network packet or message). Consequently, we believe that our approach can help generate sanitized models for a wide range of anomaly detection systems.

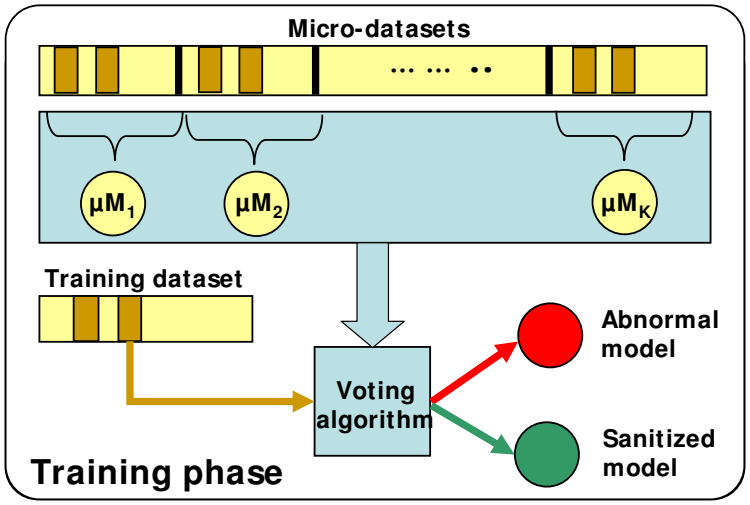


Figure 3.1: The training dataset sanitization architecture

3.2 Sanitization Effect on AD Performance

In the following experiments, we use two anomaly sensors: Anagram [109] and Payl [108; 110]. Both sensors are n-gram content-based anomaly detectors for network

packets (although we extend one of them to handle sequences of function identifiers). Although they both use an n-gram approach and train on *normal* unencrypted network packet content, these sensors have very different learning algorithms.

3.2.1 AD Sensors

The Payl [108; 110] AD sensor is a content-based anomaly detector developed in the Columbia University Intrusion Detection Systems(IDS) Lab. The sensor is based on the principle that zero-day attacks are delivered in packets whose data is unusual and distinct from all prior *normal content* flowing to or from the victim site. A Payl model contains 1-gram byte frequency distributions conditioned on the port/service and on the packet length, producing a set of statistical centroids that in total provides a fine-grained and compact model of a site's actual content flow. In the detection phase, the packets are compared against the model to check for abnormalities, computing the Mahalanobis distance between the frequency distribution of the tested packet and the correspondent model. When the value of this metric exceeds the threshold value, the packet is deemed abnormal. The sensor also allows the possibility of performing *ingress/egress* correlation, by analyzing the bidirectional traffic to detect a worm propagation and its signature.

Anagram[109] is an anomaly detection sensor that uses a mixture of high-order n-grams to model and test network traffic content and it was also developed in the Columbia University IDS Lab. The n-grams are generated by sliding windows of arbitrary lengths over a stream of bytes, which can be per network packet, per request session, or other type of data unit. The use of higher grams is more suitable for detecting significant anomalous byte sequences and their location in the data stream. The Anagram content models contain the set of *normal* n-grams and are implemented using highly efficient Bloom filters[6], reducing space requirements. In the testing phase the percentage of never-before-seen n-grams out of the total n-grams in the packet is computed and thresholding is applied to classify the packet.

3.2.2 Experimental Corpus

Our experimental corpus consists of 500 hours of real network traffic, which contains approximately four million content packets. We collected the traffic from three different hosts: *www*, *www1*, and *lists*. *www* hosts the Computer Science Department homepage and includes scripts for services such as a gateway to a tech-report database, student and faculty directory, search-engines *etc.* *www1* is a gateway to the homepages of students in the Computer Science Department, while *lists* hosts the Computer Science Mailing Lists. The three servers exhibit different content and volume of data. We partitioned this data into three separate sets: two used for training and one used for testing. We use the first 300 hours of traffic to build the micro-models and the next 100 hours to generate the sanitized model.

The remaining 100 hours of data was used for testing. It consists of approximately $775,000$ content packets (with 99 attack packets) for *www1*, $656,000$ packets (with 70 attack packets) for *www*, and $26,000$ packets (with 81 attack packets) for *lists*. Figure 3.2 presents both the total content packet and attack packet distributions for *www1*. Given that *www1* exhibits a larger volume of traffic, we chose to perform a more in-depth analysis on its traffic. In addition, we applied a cross-validation strategy: we used the last 100 hours to generate the sanitized model while testing on the other preceding 100-hour dataset.

3.2.3 Performance Results

We recall that throughout this thesis, **we refer to detection and false positive rates as rates determined for a specific class of attacks that we observed in these data sets**. We note that discovering ground truth for any realistic data set is currently infeasible.

Initially, we measured the detection performance for both Anagram and Payl when used as standalone AD sensors without sanitizing the training data. Then, we repeated the experiments with the same setup and network traces, but we included

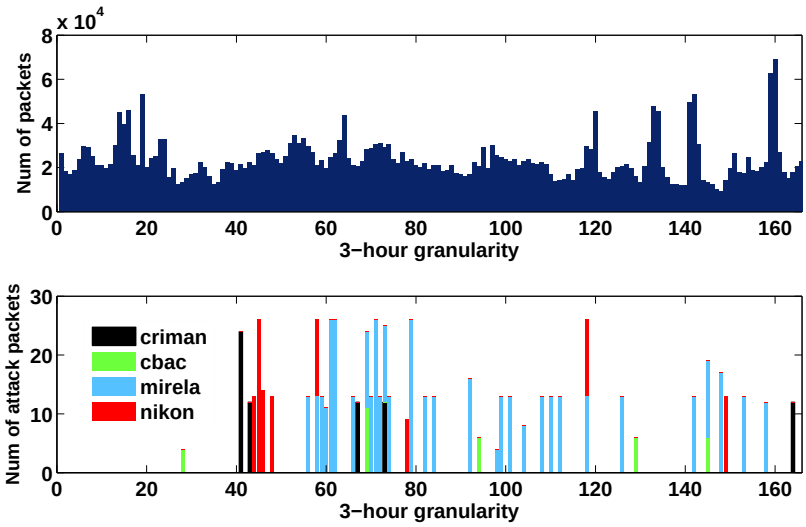


Figure 3.2: Total content packet and attack packet distributions for *www1*

the sanitization phase. Table 3.1 presents our findings, which show that sanitization boosts the detection capabilities of both sensors. The results summarize the average values of false positive (FP) and true positive (TP) rates. Both voting methods perform well. We used a granularity of three hours and a value of V that maximizes the detection performance (in our case $V \in [0.15, 0.45]$).

The optimal operating point is that which maximizes the detection of the real alerts and has the lowest FP rate. For Anagram, the sanitized and abnormal models were built to be disjoint (no abnormal feature would be allowed inside the sanitized model). If abnormal features were allowed in the sanitized model they could compromise the quality of the model given that the model contains only a set of normal grams. The traffic contains instances of phpBB forum attacks (mirela, cbac, nikon, criman) [84] for all three hosts that are analyzed.

Note that, without sanitization, the normal models used by Anagram are poisoned with attacks and thus unable to detect new attack instances appearing in the test data. Therefore, increasing AD sensor sensitivity (*e.g.* changing its internal detection threshold) would only increase false alerts without increasing the detection rate.

Table 3.1: AD sensors comparison

Sensor	www1		www		lists	
	FP(%)	TP(%)	FP(%)	TP(%)	FP(%)	TP(%)
Anagram	0.07	0	0.01	0	0.04	0
Anagram+Snort	0.04	20.20	0.29	17.14	0.05	18.51
Anagram+sanitization	**0.10**	**100**	**0.34**	**100**	**0.10**	**100**
Payl	0.84	0	6.02	40	64.14	64.19
Payl+sanitization	**6.64**	**76.76**	**10.43**	**61**	**2.40**	**86.54**

Table 3.2: Signal-to-noise ratio TP/FP: higher values mean better results

Sensor	www1	www	lists
Anagram	0	0	0
Anagram+Snort	505	59.10	370.2
Anagram+sanitization	**1000**	**294.11**	**1000**
Payl	0	6.64	1.00
Payl+sanitization	**11.56**	**5.84**	**36.05**

When using previously known malcode information (using Snort signatures represented in an "abnormal model"), Anagram was able to detect a portion of the attack packets. Of course, this detection model is limited because it requires that a new 0-day worm will not be sufficiently different from previous worms that appear in the traces. To make matters worse, such a detector would fail to detect even old threats that do not have a Snort signature. On the other hand, if we enhance Anagram's training phase to include sanitization, we do not have to rely on any other signature or content-based sensor to detect malware.

Furthermore, the detection capability of a sensor is inherently dependent on the algorithm used to compute the distance of a new worm from the normal model. For example, although Payl is effective at capturing attacks that display abnormal byte distributions, it is prone to miss well-crafted attacks that resemble the byte distribution of the target site [25]. Our traces contain such attacks: we observe this effect when we use the sanitized strategy on Payl, as we can only get a maximum 86.54% attack detection rate. We can see that the sanitization phase is a necessary but not sufficient process for reducing false negatives: the actual algorithm used by the sensor is also important in determining its overall detection capabilities.

Interestingly, the combination of Payl operating on the *lists* data set without sanitization shows a high FP rate compared to the same case where sanitization is used. After investigating this phenomena, we realized that for a specific packet length (161) the unsanitized model included a centroid that caused many false positives. The sanitized model did not contain a specific centroid created for length 161 (the packets with this length were considered abnormal in the sanitization phase) and the closest length centroid (178) was used for the testing phase.

Overall, our experiments show that the AD signal-to-noise ratio (*i.e.*, TP/FP) can be significantly improved even in extreme conditions, when intrinsic limitations of the anomaly detector prevent us from obtaining a 100% attack detection, as shown in Table 3.2. Higher values of the signal-to-noise ratio imply better results. There is

one exception: Payl used on the *www* data set. In this case, the signal-to-noise ratio is slightly lower, but the detection rate is still higher after using sanitization.

To stress our system and to validate its operation, we also performed experiments using traffic in which we injected worms such as CodeRed, CodeRed II, WebDAV, and a worm that exploits the nsiislog.dll buffer overflow vulnerability (MS03-022). All instances of the injected malcode were recognized by the AD sensors when trained with sanitized data. That result reinforced our initial observations about the sanitization phase: we can increase the probability of detecting both a zero-day attack and previously seen malcode.

3.2.4 Analysis of Sanitization Parameters

We have seen how our sanitization techniques can boost the performance of the AD sensors. Our results summarize the FP and the detection rates as averaged values obtained for the optimal parameters. We next explore these parameters and their impact on performance with a more detailed analysis using our Anagram implementation.

There are three parameters we need to fine-tune: the granularity of the micro-models, the voting algorithm, and the voting threshold. In order to determine a good granularity, we have to inspect the volume of traffic received by each site (given the characteristics of the chosen anomaly detector) such that we do not create models that are under-trained. In our initial experiments, we used 3-hour, 6-hour, and 12-hour granularity. We employed both the simple and weighted voting algorithms proposed in Section 3.1. The threshold V is a parameter that needs to be determined. It depends on the site/application modeled by the sensor. As we show, both the optimal values of V and the micro-model granularity have close values for all the sites in our experiments.

In Figures 3.3 and 3.4, we present the performance of the system when using Anagram enhanced with the sanitization method applied on the *www1* traffic. We

notice that the weighted voting algorithm appears to be a slight improvement over the simple voting algorithm. We seek a value for V that maximizes detection and achieves the lowest possible FP rate. We observe that the sanitized model built using the 3-hour micro-models shows the best performance, achieving a detection rate of 100% and minimizing the FP rate. The granularity and the voting threshold are inversely proportional because for the same dataset fewer models are built when the granularity is increased.

In Figures 3.5 and 3.6, we present the results for *www* and *lists* for a granularity of three hours and for both types of voting techniques. The best cases for these two sites are reached at almost the same value as the ones obtained for *www1*. We observe that the best case is where V has the minimum value 0.01.

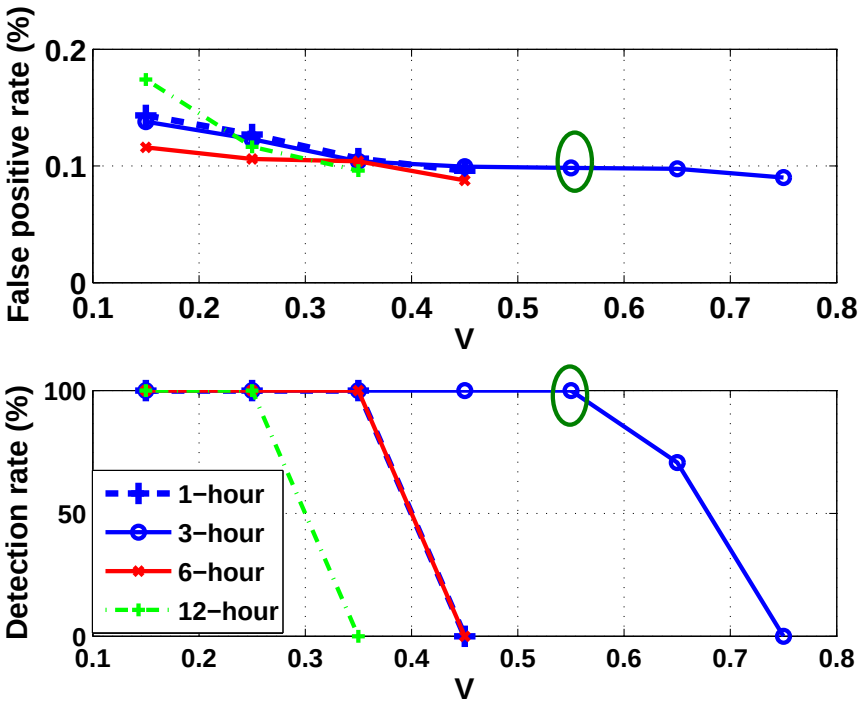


Figure 3.3: Performance for *www1* for 3-hour granularity when using simple voting and Anagram; the circled points are the optimal ones

To further evaluate our approach, we studied the impact that granularity has on

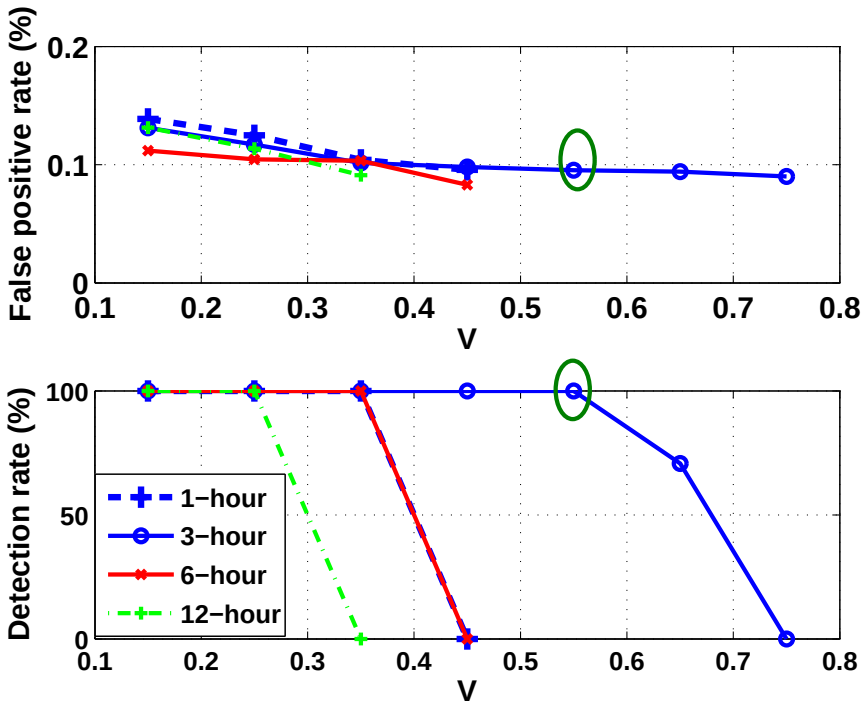


Figure 3.4: Performance for *www1* when using weighted voting and Anagram; the circled points are the optimal ones

the performance of the system. We fixed the voting threshold, and we sampled a large range of granularity values. This analysis allowed us to determine the best granularity. In Figure 3.7, we observe that the granularity of three hours performs the best, given the two threshold bounds *0.15* and *0.45* obtained from the previous experiments. For all other values of $V \in (0.15, 0.45)$, the granularity of three hours is the optimal choice. Notice that for $V = 0.45$, all values of granularity from 3 to 12 hours are optimal.

When using Payl, the granularity of three hours again performs the best, given the two threshold bounds *0.15* and *0.55*. Payl behaves differently than Anagram due to its different learning algorithm. The way the models are built is more dependent on the number of training samples because models are created for each packet length.

As we mentioned previously, our technique assumes the use of a large training

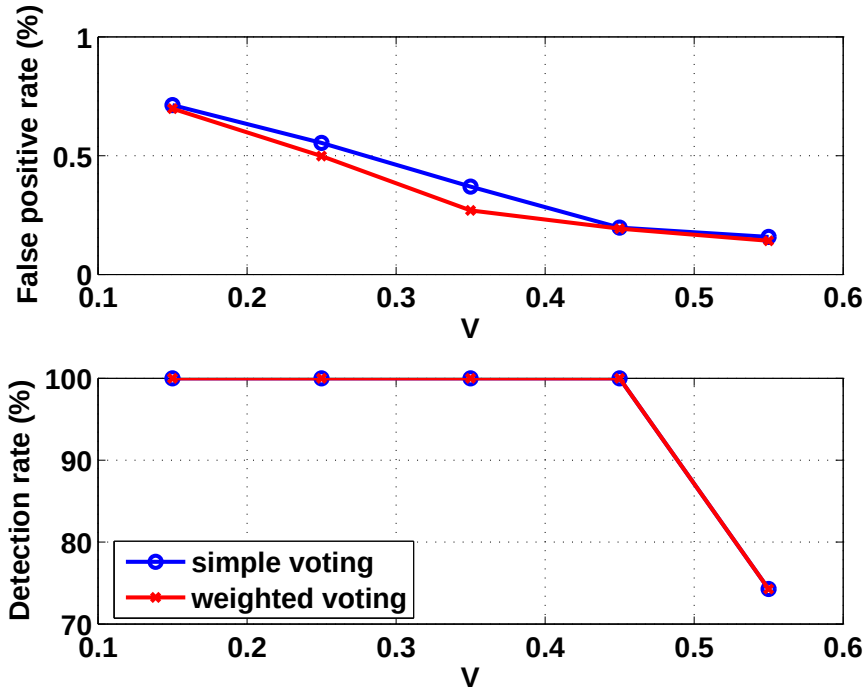


Figure 3.5: Performance for *www* for 3-hour granularity when using Anagram

dataset in order to increase the probability that an individual datum which is normal is not incorrectly deemed an anomaly. To analyze the impact that training data set size has on performance, we tested our methodology on Anagram using a certain percentage of the micro-models, starting from a randomly chosen position in the training dataset, as shown in Figure 3.9. This experiment uses 300 hours of training data, a granularity of three hours per micro-model, the weighted voting scheme, and a threshold of $V = 0.45$. The FP rate degrades when only a percentage of the 100 models is used in the voting scheme. The detection rate can vary for small numbers of micro-models, depending on the randomly chosen position in the training dataset.

Another factor is the relationship between the internal threshold of the sensor, τ, and the voting threshold, V, and the way it influences the performance of the system. Intuitively, if the anomaly sensor is more relaxed, the amount of data seen as anomalous by the micro-models will decrease. As a result, the sanitized model

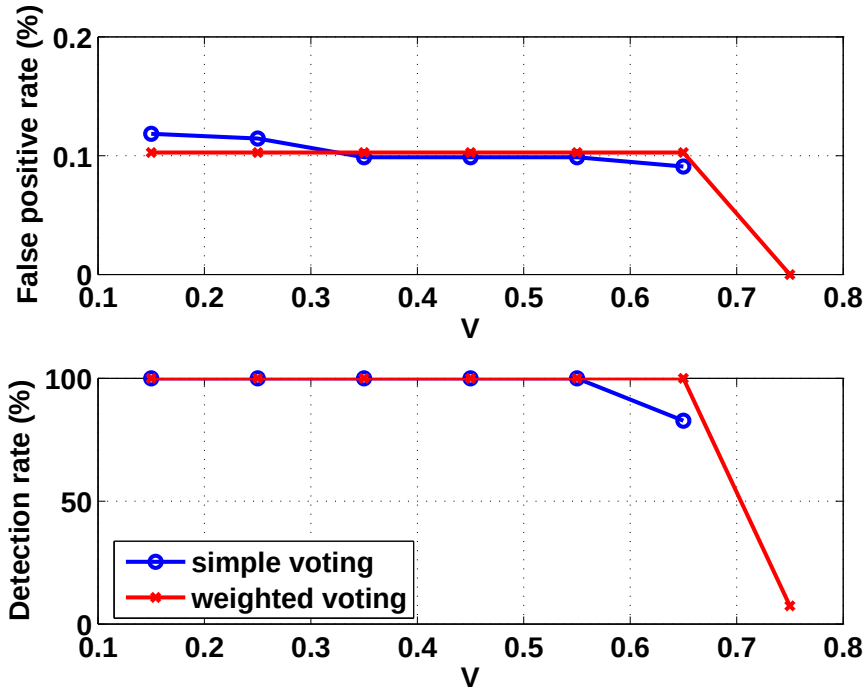


Figure 3.6: Performance for *lists* for 3-hour granularity when using Anagram

will actually increase in size and exhibit a smaller FP rate as shown in Figure 3.10. Although using a "relaxed" AD can improve the FP rate, we do not advocate such an approach to the extreme. In our experiments, the threshold for Anagram was set to $\tau = 0.4$, and we analyzed the effect of changing the internal threshold had over the performance of our system. We observed that if we increase the internal threshold, the FP rate decreases along with the detection rate.

3.3 Shadow Sensor Redirection

In order to further classify the alerts produced by the anomaly detector, we use a heavily instrumented host-based "shadow sensor" system akin to an "oracle". This type of systems perform substantially slower, usually orders of magnitude slower, than the native, un-instrumented application [1], which means we need to produce an

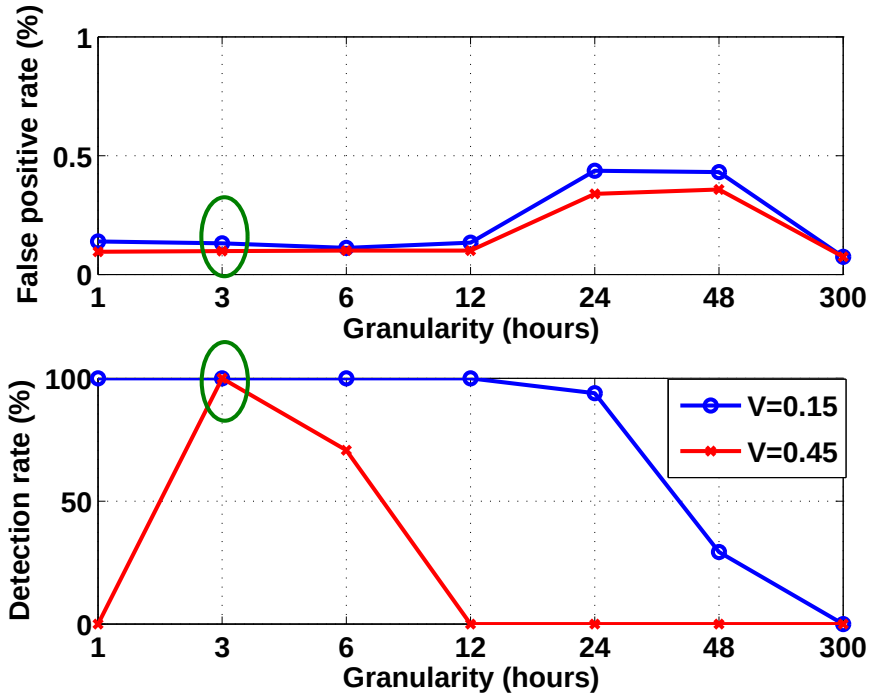


Figure 3.7: Granularity impact on the performance of the system for *www1* when using Anagram; the circled points are the optimal ones

AD sensor which exhibits a low enough FP rate. The shadow sensor processes both true attacks and incorrectly classified packets to validate whether a packet signifies a true attack. We evaluate our approach considering the computational costs involved in diverting each alert to a host-based shadow sensor. Both the feasibility and scalability of this scenario depend mainly on the amount of alerts generated by the AD sensor, since all "suspect-data" (data that causes the sensor to generate an alert) are significantly delayed by the shadow sensor.

We examine the average time it takes to process a request, and the impact that sanitization has on this time. In addition, we estimate the overall computational requirements of a detection system consisting of an AD sensor and a host-based shadow sensor. The AD sensor acts as a packet classifier that diverts all packets that generate alerts to the shadow sensor while allowing the rest of the packets to reach

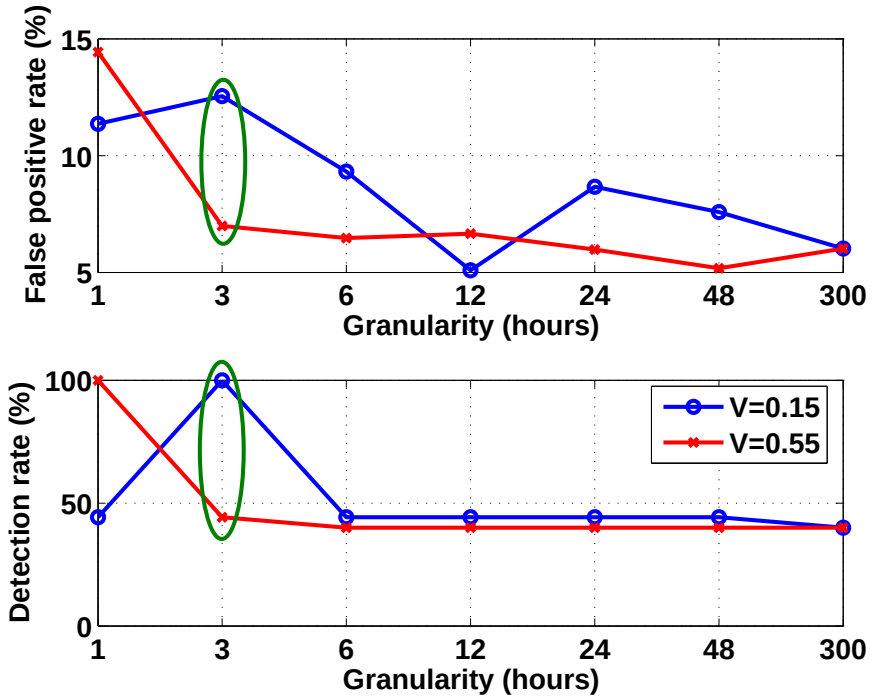


Figure 3.8: Granularity impact on the performance of the system for *www* when using Payl; the circled points are the optimal ones

the native service. This architecture effectively creates two service paths.

Our goal is to create a system that does not incur a prohibitive increase in the average request latency *and* that can scale to millions of service requests. Due to the overhead of the shadow sensor, we cannot redirect all traffic to it. Therefore, we would like the AD to shunt only a small fraction of the total traffic to the more expensive shadow. The shadow sensor serves as an oracle that confirms or rejects the AD's initial classification (see the full architecture in figure 3.11).

Although one could argue that using a shadow sensor alone is sufficient to protect a system from attack (and therefore we have scant need of a robust anomaly sensor in the first place), shadow sensors have significant shortcomings. First, they impose a hefty performance penalty (due to the instrumentation, which could include tainted dataflow analysis, a shadow stack, control-flow integrity, instruction set randomiza-

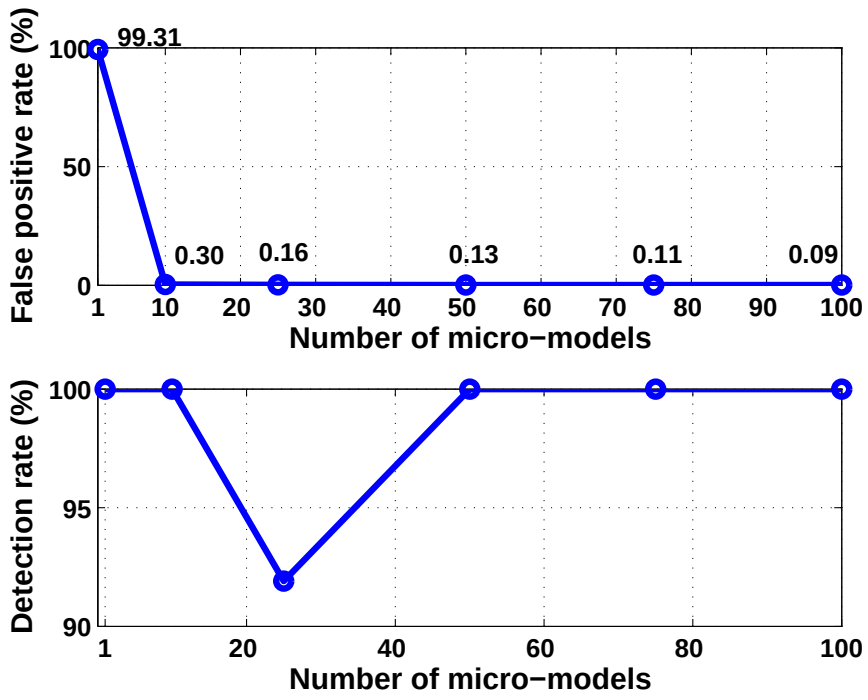


Figure 3.9: Impact of the size of the training dataset for *www1*

tion and other heavyweight detectors). Using a shadow sensor without the benefit of an AD sensor to pre-classify input would be unacceptable for many environments. Second, a shadow requires synchronization of state between itself and the shadowed "production" application. In many environments, this is a difficult task. Finally, shadow sensors have only an incomplete notion of what malicious behavior is: they use instrumentation aimed at detecting certain classes of attacks. Thus, a shadow sensor is not a perfect oracle. It serves only to offer a lower bound on the removal of attacks (and it completely misses abnormalities) if it were used to directly "sanitize" data sets.

For our performance estimation, we used two instrumentation frameworks: STEM [91] and DYBOC [1]. STEM exhibits a 4400% overhead when an application such as Apache is completely instrumented to detect attacks. On the other hand, DYBOC has a lighter instrumentation, providing a faster response, but still imposes at least

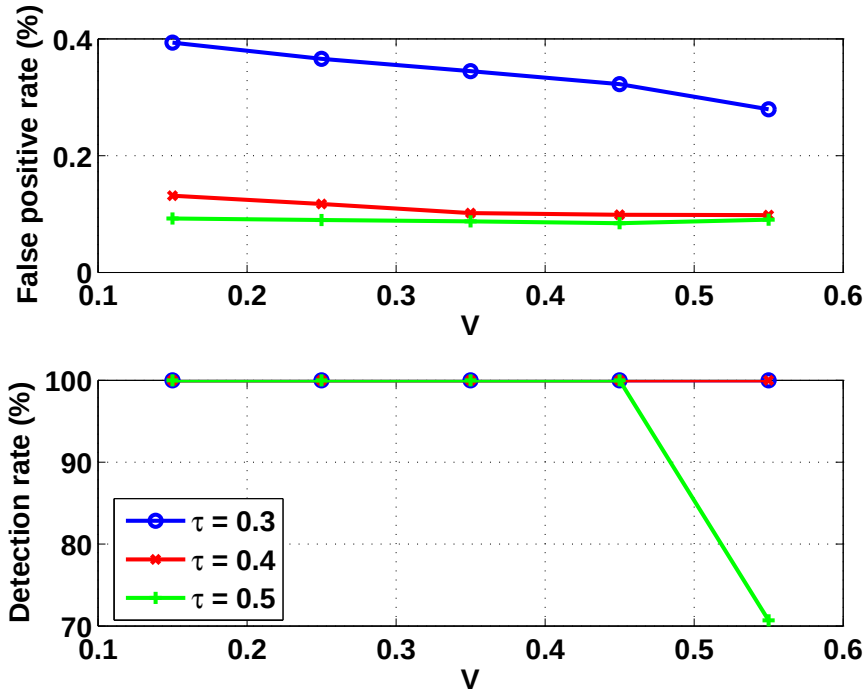


Figure 3.10: Impact of the anomaly detector's internal threshold for *www1* when using Anagram

a 20% overhead on server performance. Given that we know ground truth based on *the attacks these sensors detect*, we can estimate what the answers of the shadow servers would be. We can also estimate the overall overhead based on the reported performance of the frameworks in [91] and [1].

To compute the overall overhead, we use the method described in [109], where the latency of such an architecture is defined as following:

$$l' = (l * (1 - fp)) + (l * O_s * fp) \tag{3.7}$$

where l is the standard (measured) latency of a protected service, O_s is the shadow server overhead, and fp is the AD false positive rate.

Table 3.3 shows that, for all cases where an AD sensor filters the traffic, there is a significant improvement in the overhead over the case in which all the traffic

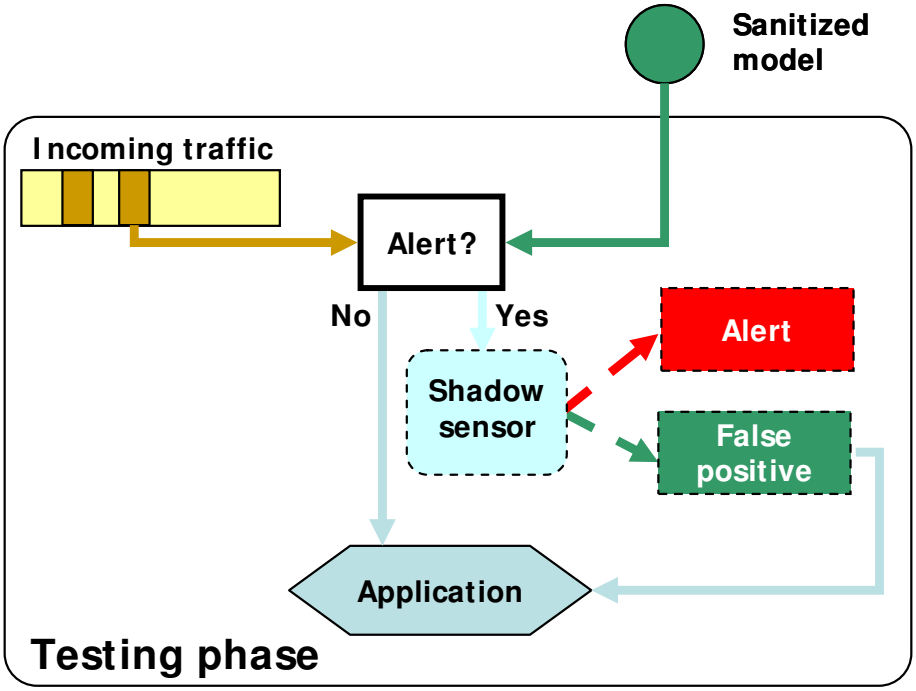


Figure 3.11: *The shadow sensor architecture.* The sanitized model is produced as shown in figure 3.1

is redirected to the shadow sensor (no AD sensor). To quantify the performance loss/gain from using the sanitization phase, we compare the average latency of the system when using Payl and Anagram with sanitized and non-sanitized training data. The difference is not significant and, more important, with sanitization we get a better detection rate, as discussed in previous sections.

3.4 Adversarial Scenarios

In this section we analyze how an attacker can attempt to evade our sanitization technique. We present the most important types of attacks which are of interest to our work and for which we try to find methods to alleviate their effect.

Table 3.3: Latency for different anomaly detectors

Sensor	STEM			DYBOC		
	www1	www	lists	www1	www	lists
no AD sensor	44	44	44	1.2	1.2	1.2
Anagram	1.0301	1.0043	1.0172	1.0001	1.0000	1.0000
Anagram+Snort	1.0172	1.1247	1.0215	1.0000	1.0005	1.0000
Anagram+sanitization	**1.0430**	**1.462**	**1.0430**	**1.0002**	**1.0006**	**1.0002**
Payl	1.3612	3.5886	28.5802	1.0016	1.0120	1.1282
Payl+sanitization	**3.8552**	**5.4849**	**2.0320**	**1.0132**	**1.0208**	**1.0048**

3.4.1 Polymorphic Attacks

Polymorphic attacks are of increasing concern, as they generate attack instances that do not share a fixed signature. That emphasizes even more the idea that anomaly detection is the appropriate solution, as opposed to signature based detectors. Fogla *et al.* [25] show that anomaly detection systems provide good defense because existing polymorphic techniques can make the attack instances look different from each other, but cannot make them look "normal". To test against such attacks, we used a popular polymorphic engine, CLET [21], to generate samples of polymorphic shellcode. In these experiments, we assume that an attacker tries to perform a training attack using a polymorphic vector (which implies that the exploit would include polymorphic shellcode). For the experiments, we used 2100 samples of shellcode generated with CLET. We used 100 micro-models with a three hour granularity derived from our dataset for *www1*. We poisoned each micro-model with 20 samples of shellcode. We also poisoned the data set from which the sanitized model was built with the remaining 100 shellcode samples.

We rebuilt the sanitized model with our system, using Anagram as the base sensor.

In the voting strategy, all of the micro-models found the 100 shellcode samples as being anomalous, given that, on average, 82% of the n-grams from 100 samples were found abnormal by the micro-models. After the sanitized model was computed, we tested it against the testing dataset of 100 hours. As expected, the performance results were identical with the ones given when the sanitized model was constructed without any shellcode samples. These experiments indicate that the problem of continuous polymorphic attacks can be handled by the local sanitization architecture. We are further interested in analyzing continuous attacks which are not polymorphic.

3.4.2 Long Lasting Training Attacks

A training attack, as defined in [109], "is one whereby the attacker sends a stream of data incrementally or continuously distant from the normal data at a target site in order to influence the anomaly detector to model data consistent with the attack vector". In this case, an attack may appear in all micro-models as well as the training dataset of the sanitized model. This type of scenario is not covered by our previous experiments, which used real traffic containing real attacks, which appeared in a small fraction of the micro-models. To test our methodology in such an extreme case, we injected a specific attack packet (in our case *mirela*) into every micro-model as well as the dataset from which the sanitized model was computed. Table 3.4 compares poisoned and "clean", or non-poisoned, sanitized models. The results were obtained using Anagram, weighted voting, a granularity of three hours, and $V = 0.35$. We can see that this method can evade our architecture.

For this reason, we want to investigate ways to alleviate the impact of long-lasting training attacks. This section presents different advents on how we can tackle this important problem. A complete solution to the training attack problem is presented in chapter 6, but it involves sharing information between collaborative sites. Here we suggest different approaches that could potentially be adopted locally, without using external information. However a complete exploration of their applicability is left for

Table 3.4: Long lasting training attacks

Sanitized	www1		www		lists	
model	FP(%)	TP(%)	FP(%)	TP(%)	FP(%)	TP(%)
non-poisoned	0.13	100	0.26	100	0.10	100
poisoned	0.10	29.29	0.26	38.27	0.10	35.80

future work.

A theoretical approach, presented by Barreno *et al.* [4] includes the use of a special test set that contains several known intrusions and intrusion variants, as well as some random points that are similar to the intrusions. After the learner is trained, misclassifying a disproportionately high number of intrusions could indicate compromises. This solution might not work in the case of 0-day attacks, which normally exhibit different features than known attacks do. Also it is not suitable for the applications that we are interested in because it assumes the knowledge of labeled data.

We consider two scenarios in which the attacker can pursue a training attack:

- If the attacker does not have any information about the targeted host, (*e.g.* micro-models granularity, number of micro-models used in the voting technique, volume of data received by the host, etc.) a training attack would have to be sent "as often as possible", with a higher rate than a normal data. That means that the probability of choosing an attack item when randomly selecting a training data point in a small time window increases with the volume of attack packets. We propose to redirect a number of randomly selected training data points (*e.g.* packets) to the shadow sensor in order to determine if the micro-models classified them correctly. The time window is chosen such that the percentage of data redirected to the shadow sensor is low (comparable to the false positive rate of an AD sensor). In case an attack is detected, the micro-models are rebuilt excluding the data similar to the attack instance. This approach assumes that

the datasets from which the micro-models were initially built are pre-stored.

- If the attacker has access to the training data used by the AD sensor and to other inside information such as the granularity of the micro-models and the number of micro-models, the attack vector can be crafted to appear at the same rate as normal traffic. This way it would not be easy to distinguish between normal and attack traffic. We recommend the use of active learning [66; 20; 87], for which the labels are provided by the shadow sensor. Active learning algorithms require labels for observation that are in the region of uncertainty, thus a small percentage of the data would be redirected to the shadow sensor. We envision augmenting this approach with mechanisms which determine if data is sent by a suspect source IP or subnet etc., and based on this information identify the moment when a training attack might occur.

3.4.3 Mimicry Attacks

Wang *et al.* [109] define a mimicry attack "as the willful attempt to craft and shape an attack vector to look normal with respect to a model computed by an anomaly detector". The attacker is assumed to have information about both the modeling algorithm and the training data. This kind of attacks have proved particularly difficult to prevent, as they target the AD sensor directly with intimate knowledge of the system. The sanitization methods described in this thesis aim to increase the performance of AD systems by improving the normal models that they use; as such they are intrinsically tied to the performance of the underlying sensor. Development of a new generation of AD sensors that are more robust to mimicry attacks is beyond the scope of this work; however, we will discuss here different types of mimicry attacks and some proposed solutions to alleviate their effect.

Fogla *et al.* [25] present a subclass of mimicry attacks called *polymorphic blending attack*. This is an attack that also has the ability to evade a payload statistics-based anomaly detector, by making each attack instance appear normal, transforming the

payload characteristics such that they fit the normal model used by the AD system. This attack was used against PAYL for both 1-gram and 2-gram distributions and it succeeded in evading it. For this type of attack, if the AD sensor by itself can be evaded, the training dataset sanitization will at least introduce another level of indirection for the attacker, while the long lasting training attack would still need to be addressed. As a response to the blending attack, Wang *et al.* [109] propose the use of randomized models/testing in order to defeat the mimicry attacks. Their *randomized models* technique assumes the use of a secret partition of the data streams for the modeling phase, which increases the overhead of the sensor. The *randomized testing* proposes to randomly partition packets into several substrings or subsequences and test each of them separately. Even if they use methods for coping with mimicry attacks, their sensor is still sensitive to non-sanitized data and long-lasting training attacks.

System call based anomaly detectors confront with *automatic mimicry attacks*[47], defined as a variation of the traditional mimicry attack which is able to hijack a program execution flow, execute malicious system call-free code, relinquish the execution flow to the diverted program to regain it later on. This attack was devised against host based AD systems like pH [92]. Bruschi *et al.* [9] propose a way to defeat this type of attack by using static analysis techniques (through the Interprocedural Control Flow Graphs) which can localize critical regions inside a program. The code is instrumented such that the integrity of dangerous code pointers is monitored and any unauthorized modification is restored with legal values. The problem of this defensive mechanism is represented by the accuracy of the static analysis phase performed on x86 binaries, which can lead to high false positive and negative rates.

3.5 Summary

Due to recent advances in polymorphic attacks, we believe that the research community should make a concerted effort to revive the use of content–based anomaly detection as a first-class defensive technique. To that end, we introduce a novel sanitization technique that significantly improves the detection performance of out-of-the-box AD sensors. We are the first to introduce the notion of micro-models: models of "normal" trained on small slices of the training data set. Using simple weighted voting schemes, we significantly improve the quality of unlabeled training data by making it as "attack-free" and "regular" as possible. Our approach is straightforward and general, and we believe it can be applied to a wide range of unmodified AD sensors (because it interacts with the training data rather than the AD algorithm) without incurring significant additional computational cost other than in the initial training phase.

The experimental results indicate that our system can serve both as a stand-alone sensor and as an efficient and accurate online packet classifier using a shadow sensor. Furthermore, the alerts generated by the "sanitized" AD model represent a small fraction of the total traffic. The model detects approximately 5 times more attack packets than the unsanitized AD model. In addition, the AD system can detect more threats both online and after an actual attack, since the AD training data are attack-free.

Chapter 4

Automated Deployment of AD Sensors

4.1 Why Automated Deployment for AD Sensors?

A major hurdle in the deployment, operation, and maintenance of AD systems is the calibration of these sensors to the protected site characteristics. Currently, AD sensors require human operators to perform initial calibration of the training parameters to achieve optimal detection performance and minimize the false positives. AD sensors can help counter the threat of zero-day and polymorphic attacks; however, the reliance on user input is a potential roadblock to their application outside of the lab and into commercial off-the-shelf software. In this chapter we take a number of steps towards AD sensors that enable true hands-free deployment and operation.

4.1.1 Overview

In chapter 3 we presented our novel sanitization technique that significantly improves the performance of AD sensors. The necessary parameters in the sanitization process were determined empirically in order to achieve the optimal operation points. Here, our aim is to automatically determine the values of the critical system parameters that

are needed for training using only the intrinsic properties of existing behavioral data from the protected host. To that end, we address the training stage and calibration of the AD sensor. We use an unlabeled, and potentially dirty sample of the training set to construct micro datasets. On one hand, these datasets have to be large enough to generate models that capture a local view of normal behavior. On the other hand, the resulting micro-models have to be small enough to fully contain and minimize the duration of attacks and other abnormalities which will appear in a minority of the micro datasets. To satisfy this trade-off, we generate datasets that contain just enough data so that the arrival rate of new traffic patterns is stable. The micro-models that result from each data set are then engaged in a voting scheme in order to remove the attacks and abnormalities from the data. The voting process is automatically adapted to the characteristics of the traffic in order to provide separation between normal and abnormal data.

4.1.2 Contributions

Our target is to create a fully automated protection mechanism that provides a high detection rate, while maintaining a low false positive rate. In chapter 3, we have explored the problem and proposed the sanitization techniques using empirically determined parameters. We also presented a shadow sensor architecture for consuming false positives (FP) with an automated process rather than human attention.

Here, we apply those insights to the problem of providing a run-time framework for achieving the goals stated above. While the sanitization process presented in chapter 3 did not require a manually cleaned data set for training, it relied on empirically determined parameters and human-in-the-loop calibration methods. Along these lines, the automated approach provides the following contributions:

- Identifying the intrinsic characteristics of the training data, such as the arrival rate of new content and the level of outliers (*i.e.* self-calibration)

- Cleansing a data set of attacks and abnormalities by automatically selecting an adaptive threshold for the voting method presented previously based on the characteristics of the observed traffic resulting in a sanitized training data set (*i.e.* automatic self-sanitization)

4.2 Self-Calibration using Time-based Partitions

In chapter 3, we focused on methods for sanitizating the training data sets for AD sensors. This resulted in better AD sensor performance (*i.e. higher detection rate while keeping the false positives low*). Here, we attempt to fully automate the construction of those models by calibrating the sanitization parameters using the intrinsic properties of the training data.

We recall that the inherent assumption of our work is that *attacks and abnormalities are a minority compared to the entire set of training data*. This is certainly true for training sets that span a long period of time. Therefore, we proposed the use of *time-delimited slices* of the training data as presented in section 3.1.1. Previously, all micro-datasets had the same empirically determined time granularity value. In this chapter the granularity value is automatically determined for each micro-dataset. Then, models are built using any chosen AD algorithm for each micro-dataset. We reinforce the fact that we treat the AD sensor as a black box whose first task is to output a normality model for a data set provided as input.

4.2.1 Model Stability

A key parameter of our automated sanitization method is the selection of the optimal time granularity. Intuitively, choring a smaller value of the time granularity g always confines the effect of an individual attack to a smaller neighborhood of micro-models. However, excessively small values can lead to under-trained models that also fail to capture the *normal* aspects of system behavior. One method that ensures that the

micro-models are well-trained is based on the rate at which new content appears in the training data [109]. This has the advantage of relying exclusively on intrinsic properties of the training data set. By applying this analysis, we can then identify for each md_i the time granularity that ensures a well-trained micro-model and thus attaining a balance between the two desiderata presented above.

We consider the training data set as a sequence of high-order n-grams (therefore a stream of values from a high-dimensional alphabet). When processing this data, for any time window tw_i, we can estimate the likelihood L_i of the system seeing new n-grams, and therefore new content, in the immediate future based on the characteristics of the traffic seen so far:

$$L_i = \frac{r_i}{N_i}, \tag{4.1}$$

where r_i is the number of *new unique* n-grams in the time window tw_i and N_i is the *total* number of *unique* n-grams seen between tw_0 and tw_i.

Assuming that the data processed by the system is not random, the value of L_i decreases much faster than the time necessary to exhaust the space of possible n-grams. We are interested in determining the stabilization point for which the number of new grams appears at a low rate, thus looking for the the knee of the curve. In order to detect the stabilization point, we use the linear least squares method over a sliding window of points (in our experiments we use 10 points) to fit a line, $L'_i(t) = a + b * t$. When the regression coefficient b approaches zero (0), we consider that the input has stabilized as long as the standard deviation of the likelihood is not significant. In our experiments, we discovered that we can relax the above assumptions to an absolute value lower than 0.01 for the regression coefficient b while the standard deviation of the likelihood is less than 0.1. The time interval between tw_0 and tw_i is then set as the desired time granularity for computing the micro-models as described above.

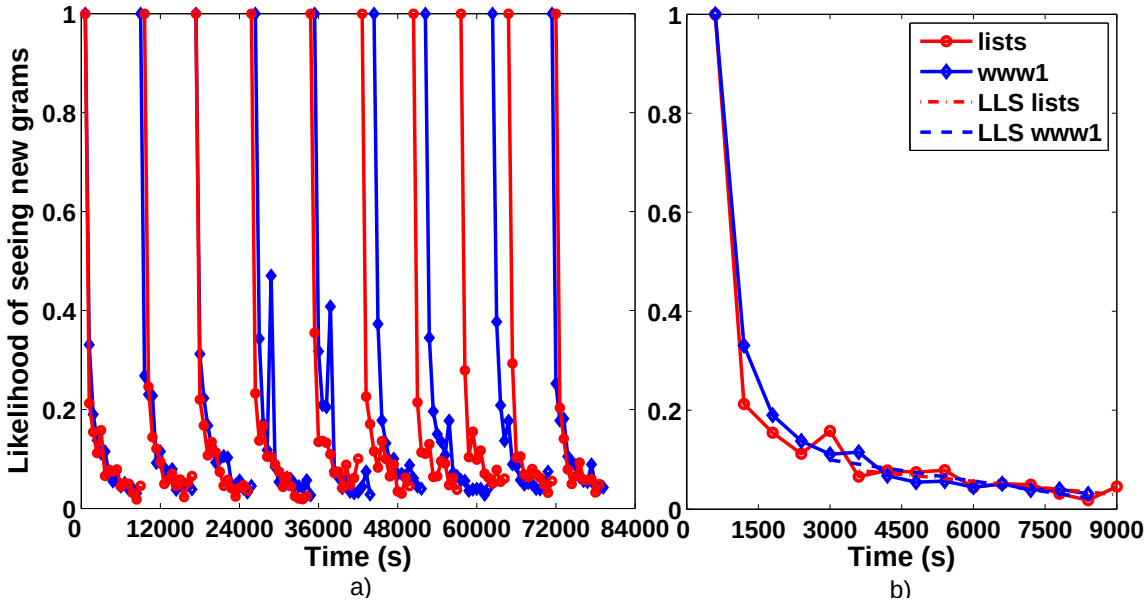


Figure 4.1: Time granularity detection($|tw| = 600s$): a) first 10 micro-models (after each model, L is reset); b) zoom on the first model

4.2.2 Analysis of Self-Calibration Parameters

Our experimental corpus consists of 500 hours of real network traffic from each of two hosts, *www1* and *lists* (details about the hosts were presented in chapter 3, section 3.2.2). Like before, we partitioned the data into three separate sets: two used for training and one used for testing. The first 300 hours of traffic in each set was used to build micro-models. Figure 4.1 shows the granularity detection method used to characterize both data sets. Figure 4.1 (a) presents the time granularity for the first ten micro-models. L is reset immediately after a stabilization point is found, and we begin to generate a new model. At a first glance, both sites display similar behavior, with the level of new content stabilizing within the first few hours of input traffic. However, they do not exhibit the same trend in the likelihood distribution, L_{www1} presenting more fluctuations. Figure 4.1 (b) presents a zoom on the first micro-model time granularity detection. The solid lines show the evolution of the L_i likelihood

metric over time(we use n-grams of size n=5). The dotted lines show the linear least squares approximation for the stabilization value of tw_i, which is used to compute the time granularity g_i.

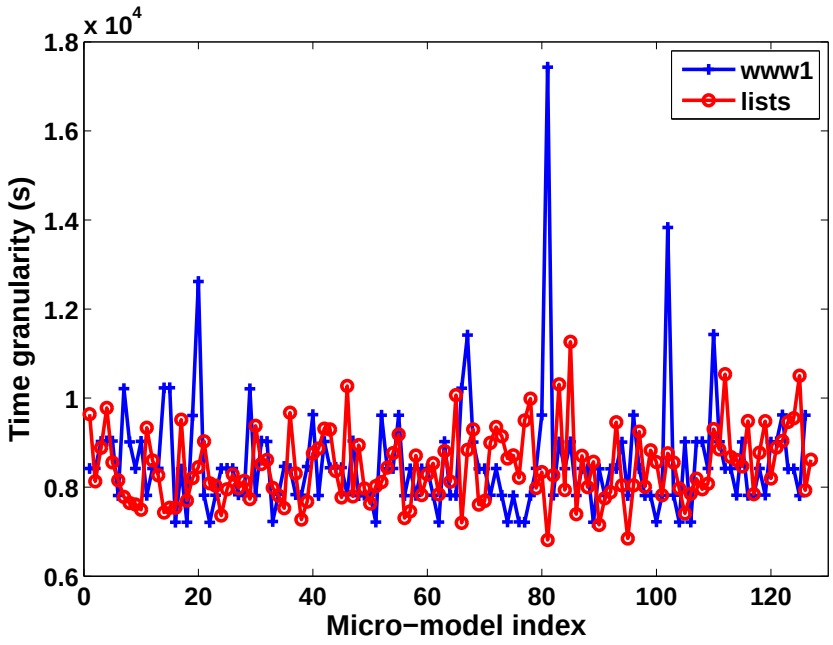


Figure 4.2: Automatically determined time granularity

Figure 4.2 illustrates the automatically generated time granularities over the first 300 hours of traffic for both *www1* and *lists*. The average value for *www1* is $g = 8562s$ ($\approx$ 2 hours and 22 minutes), while the standard deviation is $1300s$ ($\approx$ 21 minutes). For *lists* the average time granularity is $g = 8452s$ ($\approx$ 2 hours and 20 minutes), while the standard deviation is $819.8s$ ($\approx$. 13 minutes). In the next section, we will present an extensive comparison between the performance of the sanitized models that use the automated parameters versus the ones built using the empirically determined parameters.

4.3 Adaptive Training and Self-Sanitization

As presented in chapter 3, once the micro-models are built, they are used, together with the chosen AD sensor, as a classifier ensemble. A given network packet, which is to be classified as either normal or anomalous, can be tested, using the AD sensor, against each of the micro-models. Then, a voting scheme is invoked and a score is associated with the packet. In the previous chapter, we observed that the weighted voting strategy performs slightly better than the simple voting, so throughout this chapter we will use the weighted voting scheme. The final label of the packet is determined based on the value of the voting threshold, V(see section 3.1.2 for details).

4.3.1 Voting Threshold Detection

Our goal is to automatically determine the voting threshold, V. In order to establish an effective value for it, we must first analyze the impact of the voting threshold on the number of packets that are deemed normal. The extreme values have an obvious effect: a threshold of $V = 0$ (very restrictive) means that a packet must be approved by all micro-models in order to be deemed normal. In contrast, a threshold of $V = 1$ (very relaxed) means that a packet is deemed as normal as long as it is accepted by at least one micro-model. In general, for a given value V_i we define $P(V_i)$ as the number of packets deemed as normal by the classifier ($SCORE(P_j) < V_i$). The behavior of this function for intermediate values of V_i is highly dependent on the particular characteristics of the available data. For a particular data set, we can plot the function $P(V)$ by sampling the values of V at a given resolution; the result is equivalent to the *cumulative distribution of the classification scores over the entire data set*. This analysis can provide insights into three important aspects of our problem: the intrinsic characteristics of the data (number and relevance of outliers), the ability of the AD sensor to model the differences in the data, and the relevance of the chosen time granularity.

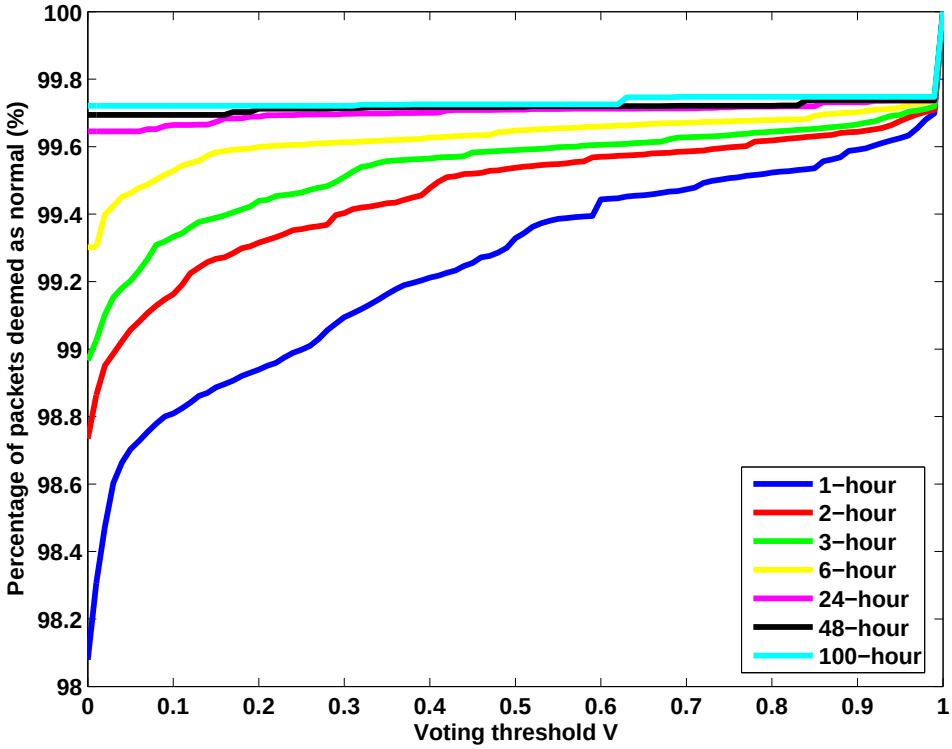


Figure 4.3: Impact of the voting threshold over the number of packets deemed as normal for different time granularities

To illustrate this concept, we will use as an example the *www*1 data set and the Anagram [109] sensor. Figure 4.3 shows the result of this analysis for time granularity ranging from 1 to 100 hours. We notice that, as the time granularity increases, the plot "flattens" towards its upper limit: the classifier loses the ability to discriminate as the micro-models are fewer in number and also more similar between themselves. We also notice that for V very close to 1, all the plots converge to similar values; this is an indicator of the presence of a number of packets that are highly different from the rest of the data in the set.

Intuitively, the optimal voting threshold V is the one that provides the best separation between the normal data class and the abnormal class. The packets that were voted normal for $V = 0$ are not of interest in the separation problem because they

are considered normal by the full majority of the micro-models and the choice of V does not influence them. So the separation problem applies to the rest data for which $V > 0$; thus, we normalize $P(V)$ as follows:

$$p(V_i) = \frac{P(V_i) - P(0)}{P(1) - P(0)} \tag{4.2}$$

The separation problem can be now considered as the task of finding the smallest threshold (minimize V) that captures as much as possible of the data (maximize $p(V)$). Therefore, if the function $p(V) - V$ exhibits a strong global maximum, these two classes can be separated effectively at the value that provides this maximum.

We have applied this method to both data sets considered in this chapter, using Anagram. The profiles of both $p(V)$ (solid lines) and $p(V) - V$ (dotted lines) are shown in Figures 4.4 and 4.5. In each case, we have marked the value of V that maximizes $p(V) - V$. In both graphs, the maximum of $p(V) - V$ corresponds to a "breaking point" in the profile of $p(V)$ (in general, any changes in the behavior of $p(V)$ are identified by local maxima or minima of $p(V) - V$). The value of the global maximum can be interpreted as a confidence level in the ability of the micro-model classifier to identify outliers, with larger values indicating a high discriminative power between the normal data and the abnormalities/attacks. A low value (and therefore a profile of $p(V)$ following the $x = y$ line) shows that the two classes are not distinct. This can be indicative of a poorly chosen time granularity, an AD sensor that is not sensitive to variations in the data set, or both. We consider this to be a valuable feature for a system that aims towards fully autonomous self-calibration: failure cases should be identified and reported to the user rather than silently accepted.

Once the value of the voting threshold V has been determined, the calibration process is complete. We note that all the calibration parameters have been set autonomously based exclusively on observable characteristics of the training data. The process can therefore be seen as a method for characterizing the combination of AD sensor - training data set, and evaluating its discriminative ability.

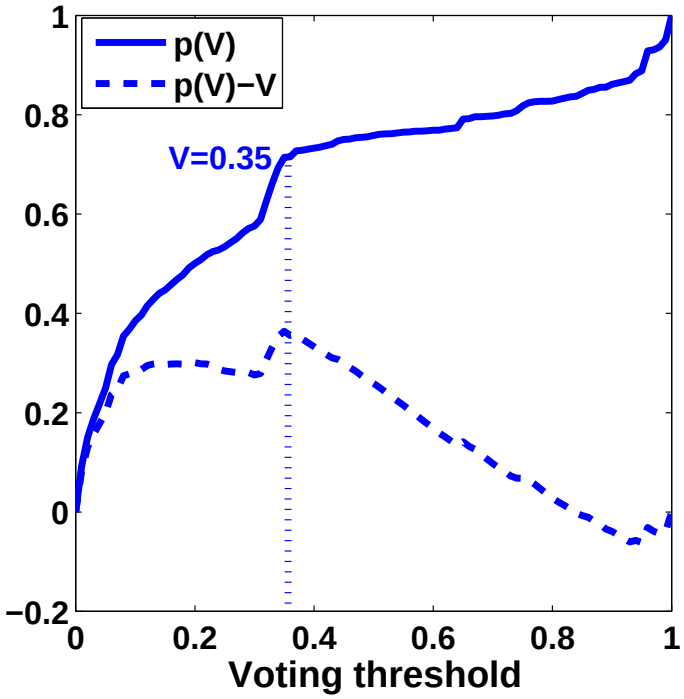


Figure 4.4: Determining the best voting threshold for *www1*

4.3.2 Analysis of Self-Sanitization Parameters

To evaluate the quality of the models built using the automatically determined sanitization parameters, we compare their performance against the performance of the sanitized models built using empirically determined parameters. There is a fundamental difference between the two types of models: for the first one the sanitization process is completely hands-free, not requiring any human intervention, while for the latter, exhaustive human intervention is required to evaluate the quality of the models for different parameter values and then to decide on the appropriate parameter values.

There are two parameters of interest in the sanitization process: the set of values for the time granularity and the voting threshold. We will therefore compare the models built using empirically determined parameters against the models built using:

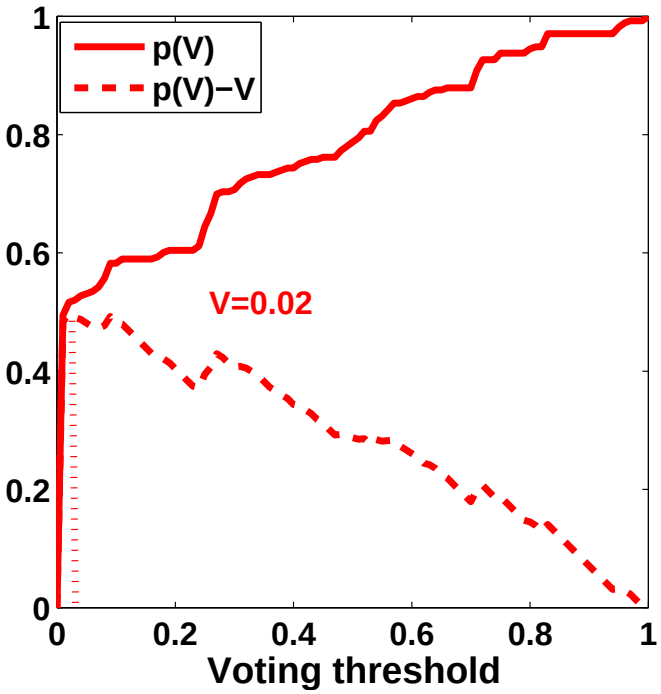


Figure 4.5: Determining the best voting threshold for *lists*

- a fixed time granularity and automatically determined voting threshold;
- automatically determined time granularities and fixed voting threshold;
- both time granularity and voting threshold determined automatically.

Figures 4.6 and 4.7 present the false positive and detection rates for models built using different sanitization parameters. The traffic contains instances of phpBB forum attacks (mirela, cbac, nikon, criman) [84] for both hosts that are analyzed. Each line shows the results obtained as the voting threshold was sampled between 0 and 1, with the granularity value either fixed at a given value (usually 1, 3 or 6 hours) or computed automatically using the method described earlier.

We note that the time granularity values empirically found to exhibit high performance were 1-, 3- and 6-hour for *www1*, respectively 3-hour for *lists*. For each of

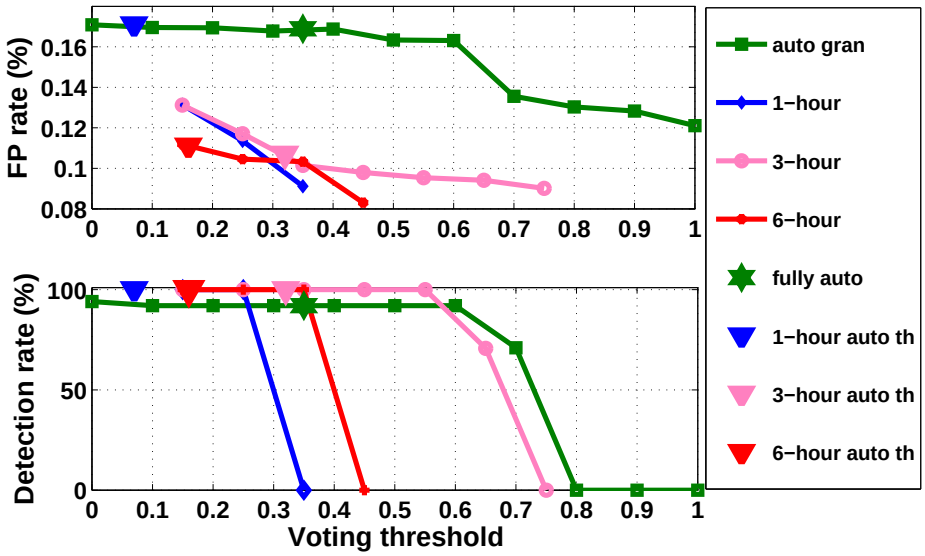


Figure 4.6: Model performance comparison for *www1*: automated vs.empirical

these values, we analyzed the performance of the models built with an automatically determined voting threshold. For each line representing a given granularity value, the triangular markers represent the results obtained with the automatically determined voting threshold. We observe that the voting threshold is placed in the safety zone for which the 100% detection rate is maintained for both *www1* and *lists*, while exhibiting a low false positive rate ($< 0.17\%$).

In the case of automated time granularity (the actual values are presented in figure 4.2), we initially explored the performance of the models determined for different values of the voting threshold, ranging from 0 to 1, with a step of 0.1. In figure 4.6, for the same fixed threshold, the detection rate is 94.94% or 92.92% compared to the 3-hour granularity (empirical optimal - 100%), while maintaining a low false positive rate ($< 0.17\%$). In figure 4.7, the results are almost identical to the empirically determined optimal (3-hour granularity).

When we use both the set of time granularities and the voting threshold determined automatically, the system is fully autonomous. In figures 4.6 and 4.7, this is

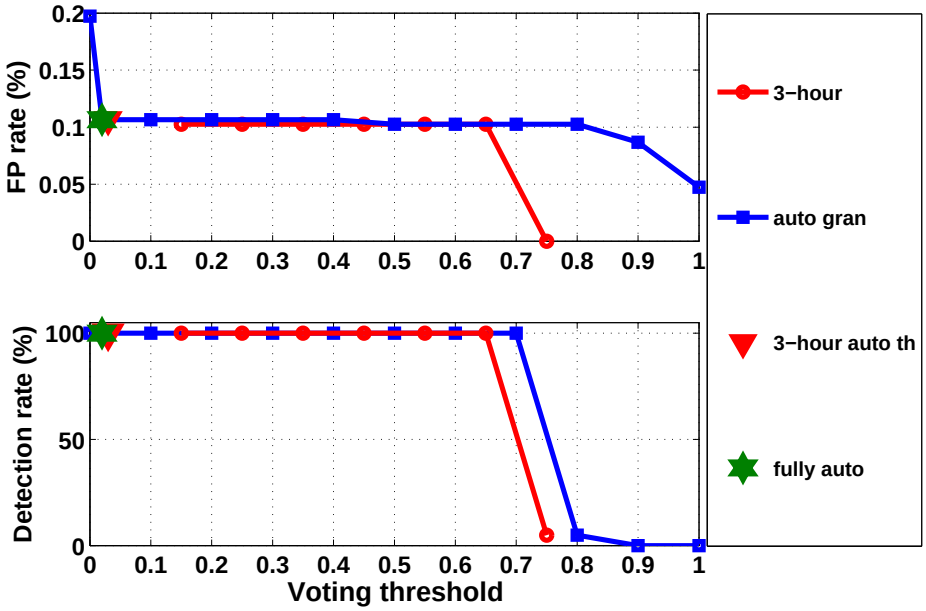


Figure 4.7: Model performance comparison for *lists*: automated vs.empirical

indicated by replacing the triangular marker with a star-shaped one. Table 4.1 also summarizes the values of false positive (FP) and true positive (TP) for the fully automated sanitized model, the empirical optimal sanitized model and the non-sanitized model. With automated parameters, for *lists* we achieve the same values as in the case of empirically determined parameters, while for *www1* the values differ, but we observe that in the absence of the sanitization process the detection rate would be 0. The most important aspect is that the fully-automated sanitization still significantly improves the quality of the AD models while setting its parameters based only on the intrinsic characteristics of the data and without any user intervention.

4.4 Summary

In this chapter, we studied the potential performance issues that stem from fully automating the AD sensors' day-to-day maintenance and calibration. Our goal is to

Table 4.1: Empirically vs. automatically determined parameters

Parameters	www1		lists	
	FP(%)	TP(%)	FP(%)	TP(%)
N/A(no sanitization)	0.07	0	0.04	0
empirical	0.10	100	0.10	100
fully automated	0.16	92.92	0.10	100

remove the dependence on human operator using an unlabeled, and thus potentially dirty, sample of incoming traffic.

To that end, we proposed to enhance the training phase of AD sensors with a self-calibration phase, leading to the automatic determination of the optimal AD parameters. We showed how this novel calibration phase can be employed in conjunction with our training data sanitization method, resulting in a fully automated AD maintenance cycle. Our approach is completely agnostic to the underlying AD sensor algorithm. We verified the validity of our approach through a series of experiments where we compared the manually obtained optimal parameters with the ones computed from the self-calibration phase. Modeling traffic from two different sources, the fully automated calibration shows a 7.08% reduction in detection rate and a 0.06% increase in false positives, in the worst case, when compared to the optimal empirically determined parameters.

Chapter 5

Model Updates

In the previous chapters we presented different approaches for improving the performance of AD sensors while automating their deployment. In this chapter, we address another critical point that AD sensors are faced with: maintaining the quality of models in the presence of changes in the modeled system behavior. The way users interact with systems can differ and evolve over time, as can the systems themselves [26]. Consequently, as AD models capture the normal behavior of systems, they need to be updated when legitimate changes are introduced in this behavior. We define this process of adapting the model to behavior changes as *model update*.

To the best of our knowledge, the two main current approaches to the problem of AD model updates are: (1) fully retraining the AD sensor, and (2) incorporating a mechanism for gradual, online retraining into the AD algorithm itself. The first approach seems unsatisfactory because retraining the model may take significant amounts of time, and it represents an additional burden on system administrators. If a patch is generated and deployed automatically (due to an automatic defense mechanism), delays introduced by a long retraining period appear to defeat one of the main purposes of automated defense: the ability to respond with little or no human supervision at speeds comparable to that of the attack. This phase may simply re-learn large amounts of behavior that have *not* changed. Unfortunately, these problems may

discourage operators from employing an AD sensor in the first place.

The latter approach (*i.e.,* online, gradual retraining) incrementally updates the AD parameters (*e.g.*thresholds, smoothing window length) and instance selection (*i.e* the process of deciding whether or not to add a point to the model) to adapt to changes in the system behavior [52]. User behavior and access patterns can change in response to social demands not anticipated by the authors of the training phase. Thus, the AD algorithm needs to continuously incorporate new data (*e.g.,* input data such as network traffic or data summarizing behavioral patterns, such as sequences of system calls) into its "normal" model and to adjust its parameters and decisions. In section 5.1, we propose a novel online learning technique that incorporates our sanitization and automation methods, maintaining the performance level of the AD sensors over a long time horizon. We employ an aging mechanism for the micro-model selection while updating the sanitization parameters. In our complete validation study (section 5.1), we show that our online approach is suitable for changes that are caused by external factors (*e.g.* changes in the users' behavior) which cannot be controlled, but the effect of their actions can be observed; we refer to it by the name of *progressive model update*. Our approach is agnostic to the underlying AD sensor, making for a general framework.

We also introduce specialized approaches to the problem of AD model update, which alter the model only when the AD sensor is notified of possible legitimate changes in the modeled system behavior, as opposed to the continuous fashion approach. We call this type of update *induced model update*, as they are performed in response to a controlled change of the operating environment. We consider three entities that need to be monitored for changes as they are internal factors that determine the behavior of the system (unlike user actions discussed above, which we consider an external factor): file systems (FS), databases (DB) and software patches. We assume that the changes in these three entities are non-malicious (other security mechanisms might be necessary) and that the monitoring system has direct access to

them, and in some cases, can add information to them. Our goal is to harness the fact that patches, especially security–related ones, cause small, localized changes in the underlying AD model, thus only the affected areas need to be updated. On the other hand, the file system and database exhibit a certain granular structure (contain files, tables, *etc.*) that can be exploited in order to create *multi-granular* modeling techniques that can be updated only in the affected areas. Therefore, if we can provide an automated mechanism that efficiently incorporates changes into the existing model, we can avoid a lengthy retraining phase. For the FS/DB case, we introduce a monitoring systems that notifies the AD system of any changes that appear in the two entities. If the anomaly detector system models the data that resides in a system at a granular level, then a section that is changed implies only a small change in the overall model.

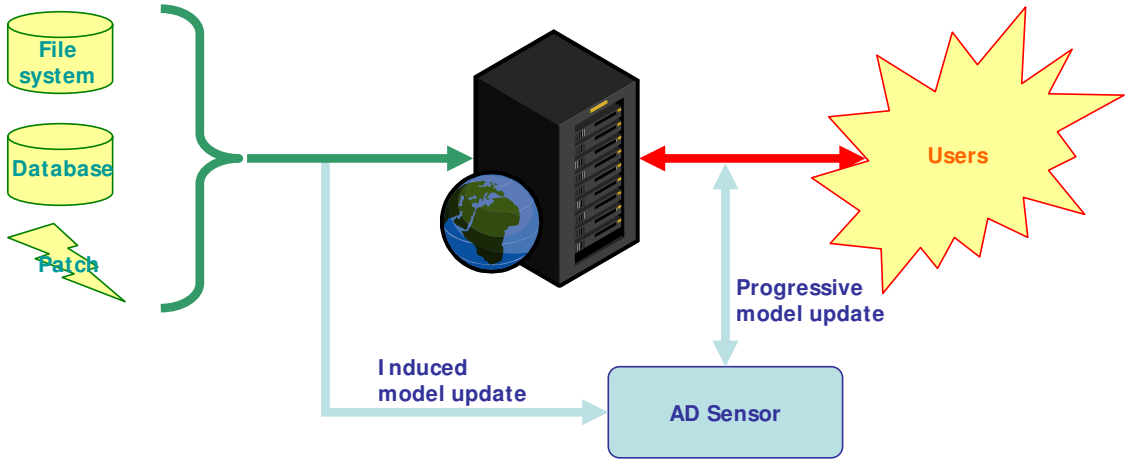


Figure 5.1: AD model update architecture

We analyze the case of patches (section 5.4) separated from the file systems and databases (section 5.3), because there is a very distinctive difference between the two approaches: for patches the changes are in the code section while for file systems and databases they appear in the data section. For the FS/DB case, we present a feasibility study along with a set of base-line results, while for the patch case, we

present a feasibility study, leaving room for future explorations.

5.1 Self-Update AD Models

The normal behavior of a system can be viewed as the set of possible legitimate inputs. Over time the inputs can change without control on the system side, while still being legitimate, and consequently the behavior of the system changes. For example, a web page or video suddenly becomes popular due to posting on slashdot or YouTube. This type of changes appear gradually without any internal control, but they can be observed. We need to find methods that allow the AD model to accommodate changes in the system's behavior *when there is no way of controlling the changes that occur in the system.*

In chapters 3 and 4, we presented a method that generates automatically self-calibrated and self-sanitized AD models. As the protected system evolves over time, the sensor's internal state becomes more and more inconsistent with the protected site. These discrepancies between the initial normality model and the current system behavior eventually render the AD sensor unusable. Therefore, the models need to adapt to this phenomenon, usually referred to as *concept drift*. As shown in [52], online learning can accommodate changes in the behavior of computer users. Here, we also propose to use an online learning approach to cope with the concept drift, in the absence of ground truth.

Our objective is to maintain the performance level of the automated AD sensors over a medium or long time horizon, as the behavior of the protected site undergoes changes or evolution. This is not an easy task [86] because of the inherent difficulty in identifying the rate of change over time for a particular site. However, we can "learn" this rate by continuously building new micro-models (as presented in chapters 3 and 4) that reflect the current behavior of the system: every time a new model is added to the voting process, an old model is removed in an attempt to adapt the normality

model to the observed changes. Without this adaptation process, legitimate changes in the systems are flagged as anomalous by the AD sensor leading to an inflation of alerts.

The main contribution presented in this section is maintaining the performance we gained by applying the sanitization methods (see chapters 3 and 4) beyond the initial training phase and extending them throughout the lifetime of the sensor by continuously updating the self-calibrated and self-sanitized model (*i.e.* self-update).

Our approach is to continuously create micro-models and sanitized models that incorporate the changes in the data. An aging mechanism is applied in order to limit the size of the ensemble of classifiers and also to ensure that the most current data is modeled. When a new micro-model, μM_{N+1} is created, the oldest one, μM_1, is no longer used in the voting process (see figure 5.2). The age of a model is given by the time of its creation.

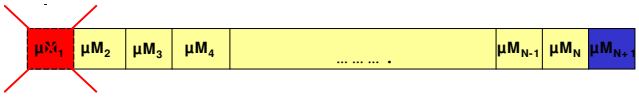


Figure 5.2: Online learning aging the oldest micro-model

Every time a new micro-model is generated, a new sanitized model is created as well. For the online sanitization we will use what we call *introspection*: the micro-models are engaged in a voting scheme against their own micro-datasets[1]. This alternative gives us the ability to apply the self-sanitization processes in an online fashion, without having to also maintain a second dataset strictly for model creation. When a new sanitized model is built, it is immediately used for testing the incoming traffic until a new sanitized model is built.

Concept drift appears at different time scales and our micro-models span a particular period of time. Thus, we are limited in observing drift that appears at scales

[1]We recall that we define a micro-dataset as the training dataset used for building a micro-model.

that are larger than the time window covered by the micro-datasets. Any changes that appear inside this time window are susceptible to being rejected by the voting process rather than being accepted as legitimate evolution of the system. In our online sanitization experiments we use 25 classifiers in the voting process (covering $\approx$ 75 hours of real time traffic) such that we can adapt to drifts that span more than 75 hours of traffic.

We cannot distinguish between a legitimate change and a long-lasting attack that slowly pollutes the majority of the micro-models. A well-crafted attack can potentially introduce malicious changes at the same or even smaller rate of legitimate behavioral drift. As such, it can not be distinguished using strictly introspective methods that examine the characteristics of traffic. However, the attacker has to be aware, guess, or brute-force the drift parameters to be successful with such an attack. In chapter 6, we will present a different type of information that can be used to break this dilemma: model data from a network of collaborative sites.

5.1.1 Self-Update Model Evaluation

To illustrate the self-update modeling, we first apply the online sanitization process for the first 500 hours of traffic (presented as our experimental corpus in chapter 3) using Anagram as the base sensor. Figures 4.2 and 5.3 present the fully automated sanitization parameters: the time granularity for each micro-model used in the creation of the new sanitized models, respectively the voting threshold for each newly created sanitized model.

If we didn't employ a model update mechanism, a sanitized model would be built only once. Thus, we refer to the first sanitized model as the *static sanitized model*. Because in the online sanitization process the models change continuously, we call them *dynamic sanitized models*. To analyze how the online sanitization performs, in figures 5.4 and 5.5 we compare the static sanitized model alert rate against the dynamic sanitized models alert rate for *www1*.

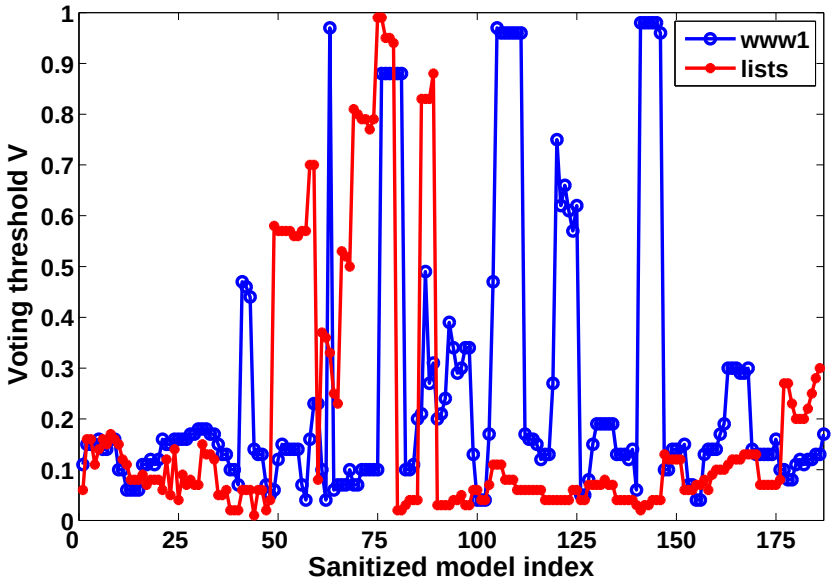


Figure 5.3: Automatically determined voting threshold for *www1* and *lists*

Figure 5.4 presents the total number of alerts for each micro-dataset tested with both the static and dynamic models. We first notice that, for a few micro-dates, the alert rate reaches levels up to 30% for both model types. After analyzing the alert data, we determined that the high alert rate was generated not by abrupt changes in the system's behavior, but rather by packets containing binary media files with high entropy. This type of data would be considered anomalous by AD sensors such as Anagram. Thus the recommendation is to divert all the media traffic to specialized detectors which can detect malicious content inside binary media files. Figure 5.5 presents the alert rate after ignoring the binary packets. We can observe that there is no significant difference between the alert rate exhibited by the static and dynamic sanitized models. Thus we can conclude that there are no fundamental changes over the 500 hour period.

In terms of performance, table 5.1 presents both the false positive rate (including the binary packets) and the detection rate for *www1* and *lists*. Abrupt changes in the

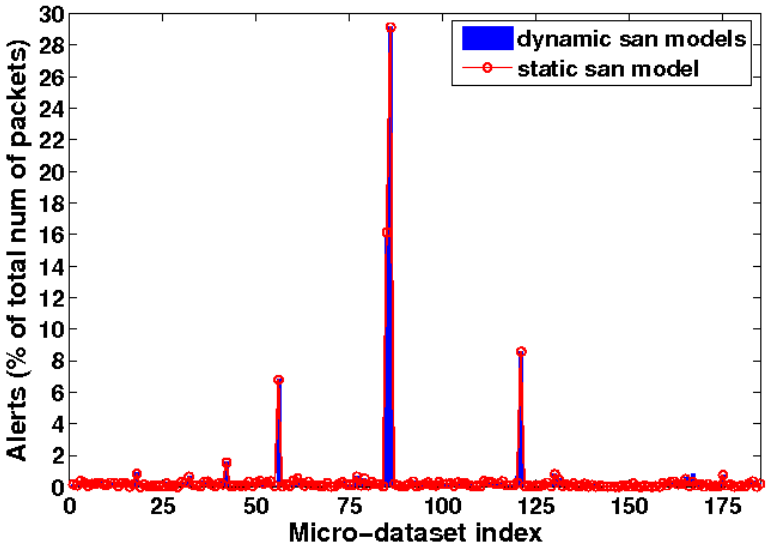


Figure 5.4: Alert rate for *www1*: both binary and ascii packets

voting threshold (as shown in figure 5.3) determine the creation of more restrictive models, thus the increase in the detection rate and/or the false positive rate. For *www1* the signal-to-noise ratio (*i.e.* TP/FP) is improved from 155.21 to 158.66, while for *lists* it decreases from 769.23 to 384.61.

Table 5.1: Static model vs. dynamic models alert rate

Model	www1		lists	
	FP(%)	TP(%)	FP(%)	TP(%)
static model	0.61	94.68	0.13	100
dynamic models	0.62	98.37	0.26	100

We then investigated concept drift appearing at larger scale such as weeks and months, as opposed to days. For this, we tested our method for traffic from the same site, collected at months difference. Figures 5.6 and 5.7 present the alert rate for both static and dynamic models, with and without the binary packets. Vertical lines

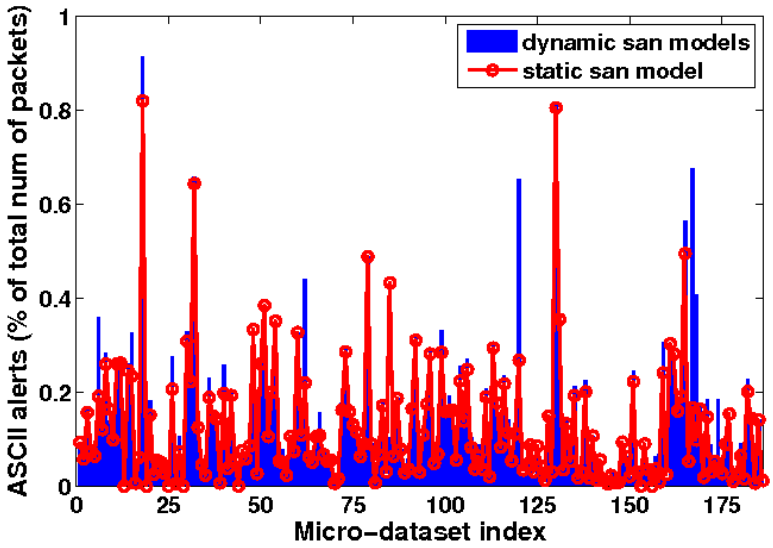


Figure 5.5: Alert rate for *www1*: ascii packets

mark the boundary between new and old traffic. We can observe that when changes happen in the system, the alert rate increases for both static and dynamic models. After the dynamic models start updating to the new data, there is a drop in the alert rate, back to levels below 1%. For the static model, the alert rate stays at about 7%, demonstrating the usefulness of a self-updating sanitization process.

Figure 5.8 presents the raw number of alerts that our system returns on an hourly basis. We note that spikes in the number of alerts can render manual processing difficult, especially when there are changes in the system under protection and the models gradually adapt to the new behavior. However, manual processing of alerts is not the intended usage model for our framework; our ultimate goal is to build a completely hands-free system that can further identify the true attacks from the false positives. Our study of computational performance presented in 3.3 shows that, with the shadow sensor architecture, the false positives can be consumed automatically and neither damage the system under protection nor flood an operational center with alarms.

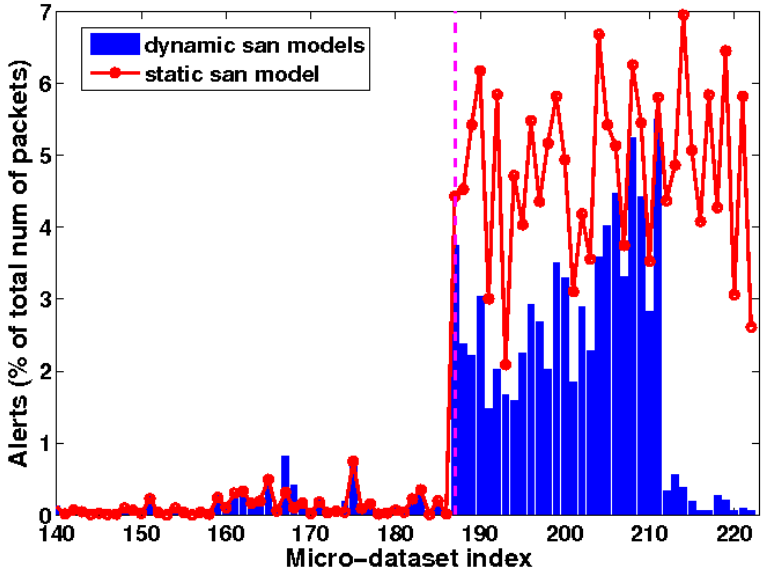


Figure 5.6: Concept drift detection for *www1* - alert rate for both binary and ascii packets. Vertical line marks the boundary between new and old traffic

5.1.2 Computational Performance Evaluation

To investigate the feasibility of our online technique we have to analyze the computational overhead that it implies. Ignoring the initial effort of building the first batch of micro-models and the sanitized model, we are interested in the overhead introduced by the model update process. Table 5.2 presents a breakdown of the computational stages of this process.

The overhead has a linear dependency on the number and the size of the micro-models. For *www1*, we used 25 micro-models per sanitization process and the size of a micro-model was on average 483 KB (trained on 10.98 MB of HTTP requests). The experiments were conducted on a PC with a 3GHz Intel(R) Xeon(R) CPU with 4 cores and 16G of RAM, running Linux. This level of performance is sufficient for

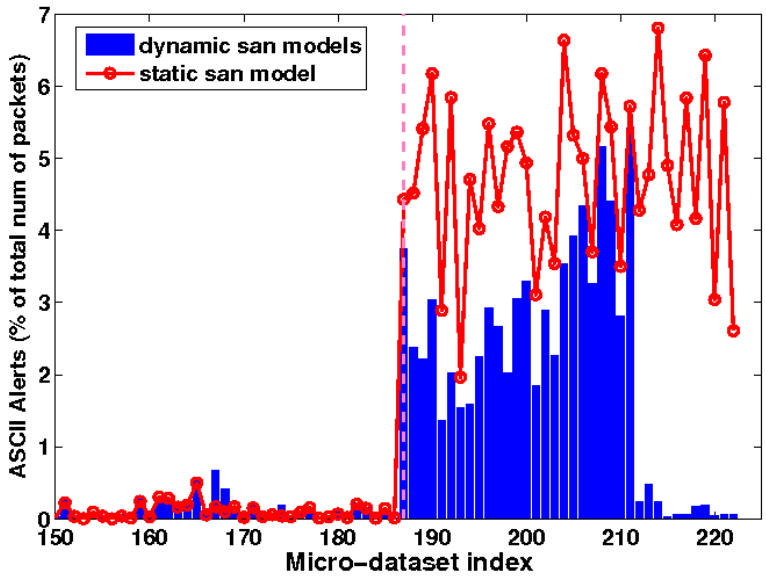


Figure 5.7: Concept drift detection for *www1* - alert rate for ascii packets. Vertical line marks the boundary between new and old traffic

Table 5.2: Computational performance for the online automated sanitization for *www1*

Task	Time to process
build and save a new micro-model	7.34 s
test its micro-dataset against the older micro-models	1 m 12 s
test the old micro-datasets against the new micro-model	1 m 58 s
rebuild and save the sanitized model	3 m 03 s

monitoring and updating models on the two hosts that we tested in this section, as it exceeds the arrival rate of HTTP requests. In the case of hosts displaying higher traffic bandwidth, we can also exploit the intrinsic parallel nature of the computations in order to speed up the online update process: multiple datasets can be tested against multiple models in parallel, as the test for each dataset-model pair is an independent

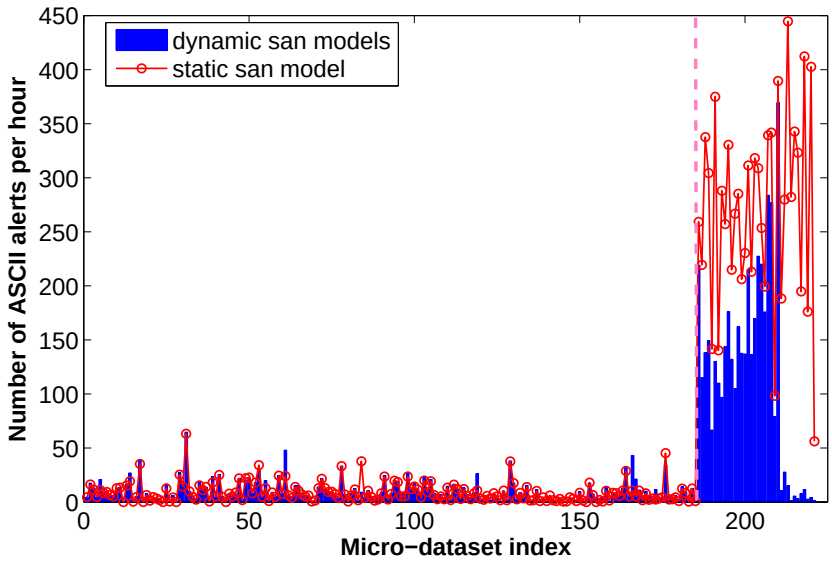


Figure 5.8: Number of ASCII alerts per hour for *www1*. The vertical line marks the boundary between new and old traffic

operation. In future work, we will implement a parallel version of this algorithm to test these assumptions.

5.1.3 Micro-Models Clustering

As mentioned above, our sanitization technique can introduce a delay until the sanitized model accurately represents the new behavior of a system when this changes. In order to minimize this delay, we propose for future work the use of fewer micro-models in the voting technique, by clustering similar micro-models. After clustering the micro-models, the model that is most recent in a specific cluster is used in the voting process. For example, as shown in figure 5.9, if models μM_1, μM_3 and μM_N were clustered in cluster c_1, we propose to use in the voting process, the most recent model, *i.e.* μM_N. We will call the most recent model in a cluster the *cluster representative micro-model*.

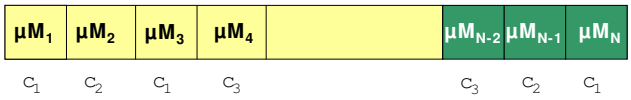


Figure 5.9: Clustering the micro-models

This approach implies also the fact that we will use a different voting technique. The two techniques presented in chapter 3 will be augmented with two new ones, which will take into account similarities between the micro-models. In order to create the sanitized model, labels are generated for each packet tested against each cluster representative micro-model and then used in the voting scheme, which generates a final score per tested packet.

$$SCORE(P_j) = \frac{1}{W} \sum_{i=1}^{N} w_i^c \cdot L_{j,i} \tag{5.1}$$

where w_i^c is the weight assigned to the cluster representative micro-model M_i^c and $W = \sum_{i=1}^{N} w_i^c$. Here the previously presented *simple voting* method becomes a *weighted voting* method which assigns to each cluster representative micro-model M_i^c a weight w_i^c equal to the cardinality of the cluster c_i. The previously presented *weighted voting* method assigns to each cluster representative micro-model M_i^c a weight w_i^c equal to the sum of numbers of packets used to train all the micro-models in cluster c_i.

5.2 Error-Response Feedback for Self-Update

As presented above, our self-update technique allows the AD sensor to adapt to changes in the protected system without any access to ground truth information. In this section we discuss a mechanism that uses partial labels of a given training datatset. This mechanism identifies as *abnormal input data* the input for which which the system under protection gives an *error response* (ER). This is data for

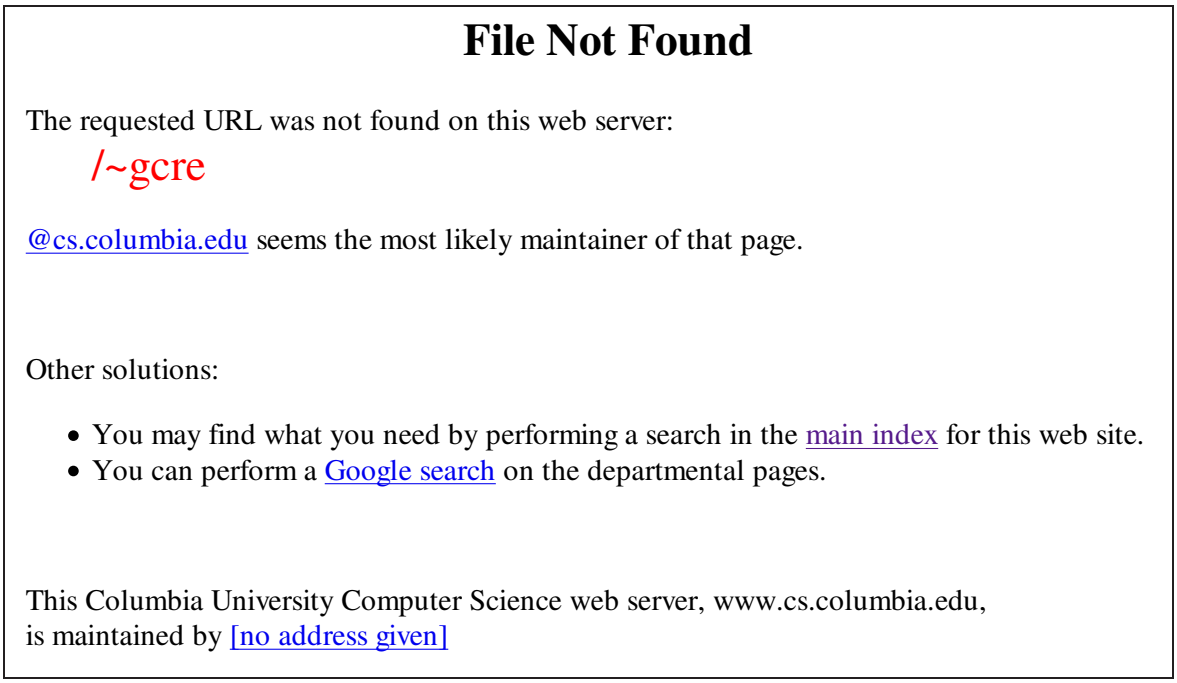
which the system acknowledges the inability to respond "normally". It is then labeled as abnormal, not used in the sanitization process and encapsulated in the abnormal model.

Our approach aims to address error response feedback for web servers, but it can also be applied to other services as well. A web server returns an error for a malformed request or a request that it can not locate. It is important to note that this technique is not foolproof. As Reynolds *et al.*[83] remarked, a server that is successfully attacked can respond with success, while a server which is not vulnerable to the same attack will respond with an error code. Based on this observation, they leveraged the software diversity to detect attacks based on disagreements between the responses of diverse servers. However, by encapsulating the requests for which the server returns an error in abnormal models, we can still gain better view of the traffic the server expects to handle, and thus in turn improve its normality model.

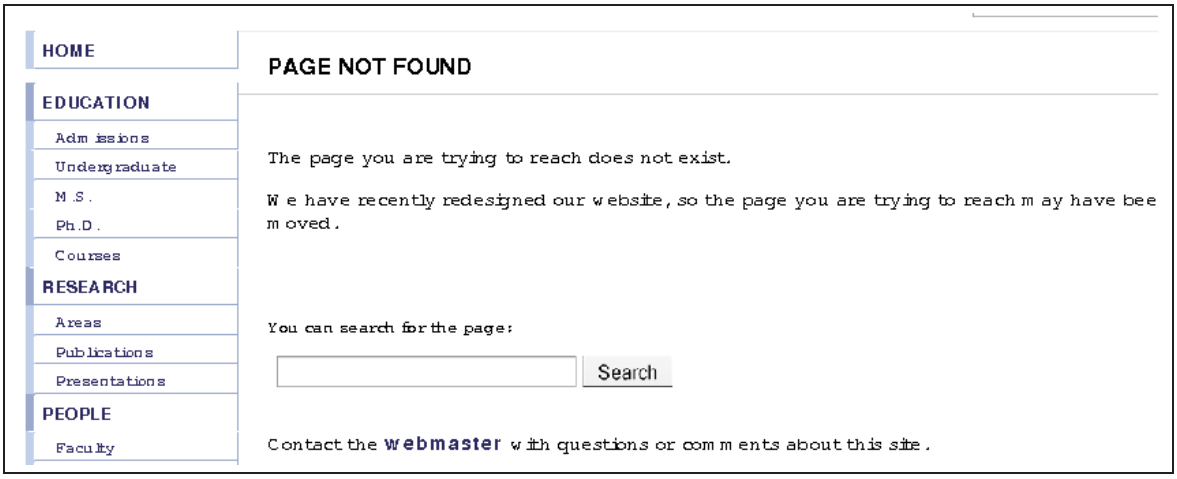
Before we proceed to the implementation details and the experimental evaluation for the online sanitization in conjunction with the error response feedback mechanism, we present the type of error codes that an HTTP server can return. We distinguish between two types of error responses:

- HTTP error 4xx, 5xx: the 4xx codes are intended for client errors and they are the most common errors. The 5xx indicate the cases in which the server is aware that it encountered an error [105]. The error pages themselves can be customized;

- application specific errors: the error pages that originate with the web application when one of the above errors appear or other specific formatting/specifications enforced by the application itself are violated. Figure 5.10 exemplifies both types of error responses.



(a)



(b)

Figure 5.10: Examples of error pages: (a) HTTP server error page customized for this application; (b) application-specific error page

5.2.1 Implementation Details

We had two alternatives to identify the HTTP requests that return error codes: to instrument the application in order to communicate them directly to the AD sensor or to compare the server responses against a pre-determined set of possible error pages. We chose the second solution as it is less invasive for the application. First, we had to identify the possible error pages for any given web application. Again we considered the case of manual vs. automatic processing. For small sites, it is usually possible to determine the error responses by manually targeting the server with bad requests. For larger sites, it becomes unfeasible to determine the error pages manually. For this purpose, we developed an Error Response (ER) WebCrawler that extracts all possible cases of error responses for a given web server. The ER WebCrawler starts from the main web page and stores all the links in the page and then uses a breadth first search algorithm to extract all possible pages. Every link is modified using a fuzzing process [65; 90]:

- for static pages, the URL is appended with a random string;
- for dynamic pages, the fuzzing is done at two different levels: the name of a variable is randomly changed while its value is kept unmodified, and the name of a variable is kept unchanged and the value is randomly changed.

The next step was to implement a TCP flow reconstruction tool to extract HTTP request/response pairs. Given an HTTP request, its correspondent HTTP response is compared against the extracted error responses using the longest sub-sequence algorithm. Depending on the comparison result, the incoming traffic is further redirected to the sanitization process or is used as training data for the abnormal model.

5.2.2 Experimental Evaluation

In this section we analyze the impact that the error response filtering mechanism has on the online sanitization process. For our experiments we used approximately 85

hours of real traffic for *www1* (*i.e.* the second dataset used in section 5.1.1), for which we had access to both incoming and outgoing traffic.

In order to perceive the significance of traffic reduction introduced by the error response filtering process, we analyzed the volume of error responses that *www1* exhibited. We considered a 24-hour window time (380,052 network packets) for a detailed analysis. Figure 5.11 considers an hourly basis approach illustrating an approximate 6% rate of error responses out of the total number of HTTP responses. The ER WebCrawler detected six types of error pages for *www1*, two of them being "File Not Found" and "Access Forbidden"

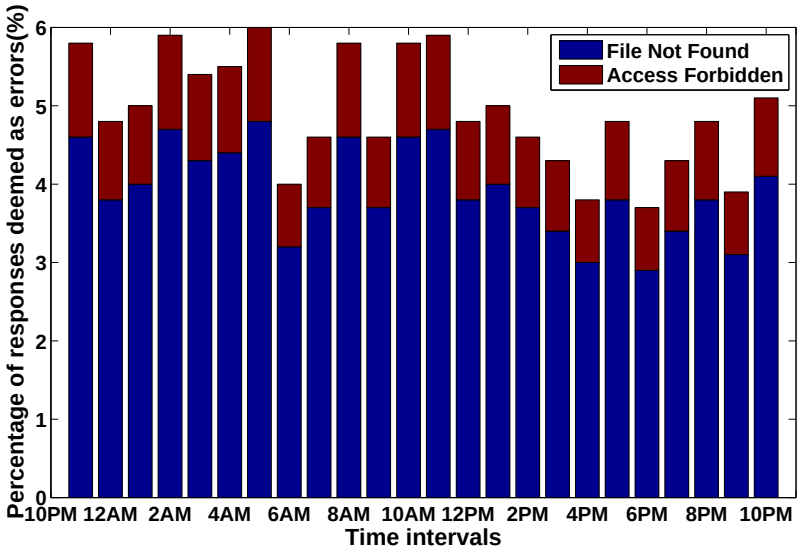


Figure 5.11: Percentage of error responses out of total number of responses for 24 hours on *www1*

To continue our investigation, we applied the online sanitization process on the given dataset with and without the filtering process. Figures 5.12 and 5.13 present the automatically determined time granularity and voting threshold for the online sanitization with and without the error response filtering. We can observe that, for both sanitization parameters, there are a number of differences. We conjecture that

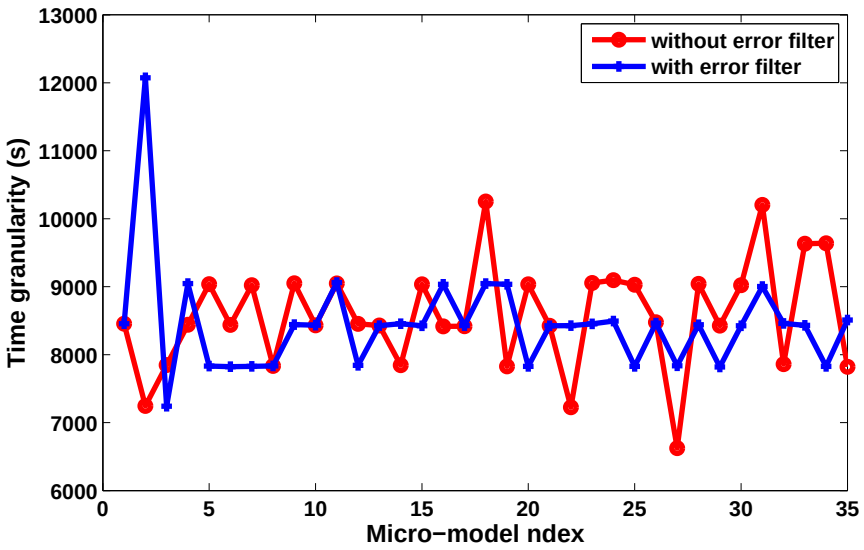


Figure 5.12: Automatically determined time granularity with and without error filtering for *www1*

the malformed requests changed the distribution of normal data in the micro-models, as they are not necessary similar to normal requests, and they constantly present in the traffic, as figure 5.11 shows. Table 5.3 presents the number of false alerts raised in both cases. With the error filtering in place, the sanitized models exhibited a smaller false alert rate as the models became denser and better characterize the normal content that the HTTP server receives.

Table 5.3: Error response filtering vs. no error response filtering

	w/o error response	w. error response
false alerts	181	120

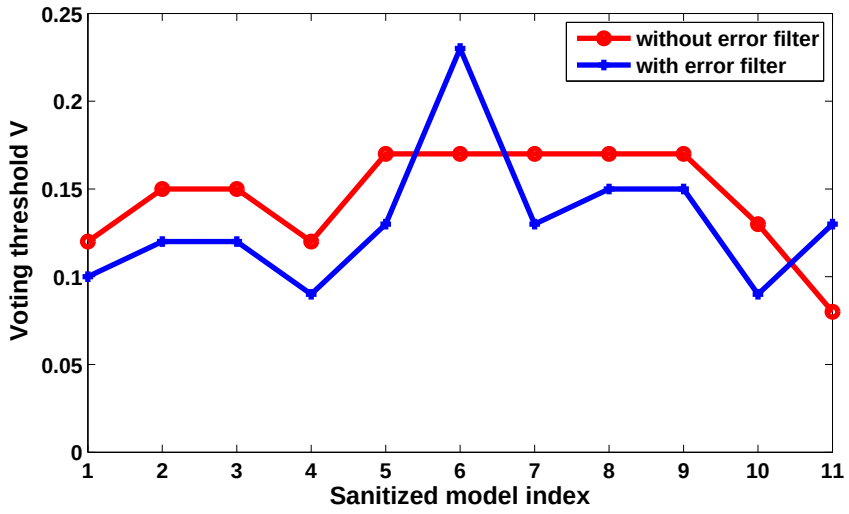


Figure 5.13: Automatically determined voting threshold with and without error filtering for *www1*

5.3 FS/DB-based AD Model Update

In this section, we propose to explore web techniques, in which we can cope with *concept drift* by relying on the internal state of the file system and database. Although these techniques attempt to address the content and behavioral changes in web servers, they can also be applied to other applications as well including Voice over IP (VoIP) servers.

A web server provides either content from the file system or dynamically generated content that interacts with databases through SQL queries(see figure 5.14). Legitimate changes that happen on the file system and in the back-end databases are reflected in the HTTP server responses. An AD model of the behavior will self-update using these changes. Another important aspect is that the replies of the web server have to be consistent with the file system and database content when no changes are made to any of them.

Our solution uses completely different training and self-updating methods for stat-

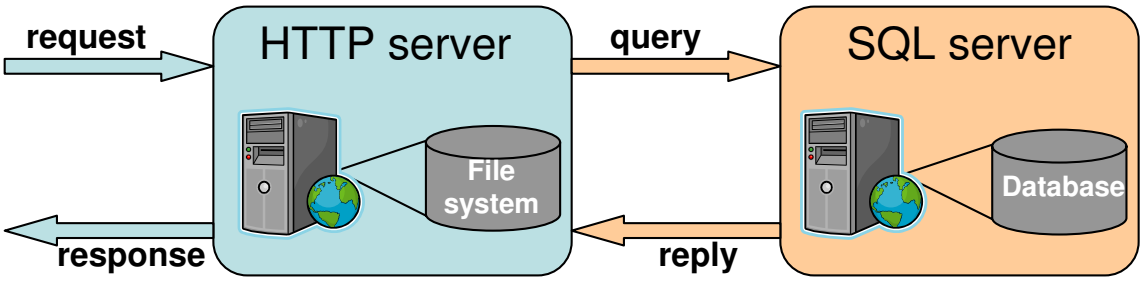


Figure 5.14: Front-end and back-end correlation for web server model update

ically and dynamically generated web pages. For the case of static replies, the overall model of the application maps each possible URL request with the sub-model representation of the correspondent file that resides in the file system and is returned by the web server. When changes are made in the file system only the sub-models of the changed files are altered. If a file is removed than the AD model drops the information about it. If a file is changed, its sub-model is updated. Otherwise, if a file is added its sub-model is built and mapped to the right URL request. In the testing phase, the URL requests that are sent to the server have to match requests from the model and also the replies of the server have to match the replies in the model that are mapped to the requests. This approach guarantees that only legitimate requests can be made to the server.

For the case of dynamic replies, sub-models corresponding to a particular script file (e.g. index.php) can be generated for both the HTTP requests and HTTP replies. The initial models can be trained using a sanitized training dataset (presenting no attacks) generated with our sanitization tool for anomaly detection. When a new request is sent to the web server, it can be first correlated with the SQL query that is generated by the HTTP server, to ensure that the correct information is asked from the SQL server. Another correlation is performed between the SQL reply and the HTTP reply based on the assumption that information returned by the database has to be reflected in the HTTP reply. When new data is introduced to the file system or

to the database, the changes have to be reflected in the two types of sub-models by altering only them accordingly. We speculate that even if the content is dynamically generated there will be common content between pages generated by the same script with different parameters, while part of the different content will be related to the data returned by the database. An attacker must use a great deal more effort to fashion a mimicry attack if we compute different models for each script file.

5.3.1 Feasibility Study

In our proof-of-concept implementation, we first created the notification system. For notifying the AD sensor of any file system alterations, we used inotify-tools [63], a library and set of command-line programs for Linux providing a simple interface to *inotify* (a Linux kernel subsystem that provides file system event notification). Table 5.4 presents an example of the use of inotifywait tool, that monitors the current directory and detects that the file database_changes.txt has been modified. The output of inotifywait can be parsed such that we can distinguish between file modification, deletion and creation. In order to detect alterations on the file systems related to web pages returned by a web server, we propose to recursively monitor the directory where the web site resides.

For the database notification system we used the MySQL triggering mechanism. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table [71]. In order to be able to track all the changes appeared in the database, we propose the use of a mirrored database that is populated with new information when the triggers are activated. The mirrored database is checked by our notification system in order to determine the type of changes that were incurred in the system. The mirrored database and the triggers are generated automatically by parsing the original database. Table 5.5 shows an example of a trigger definition. We first create a mirror of the table *info*, called *metadata_info*. Aside from the columns in *info*, we add one more column, *flag*, that

Table 5.4: *Example of the inotify-tool use to capture the changes in the current director.* The file database_changes.txt is modified while inotifywait is running. inotifywait outputs the changes on the file system.

```
$ inotifywait -m -r -e MOVE -e MODIFY -e DELETE .
Setting up watches. Beware: since -r was given, this may take a while!
Watches established.
./ MOVED_FROM database_changes.txt
./ MOVED_TO database_changes.txt
./ MODIFY database_changes.txt
./ MODIFY database_changes.txt
./ DELETE .#database_changes.txt
```

stores the type of alteration that was made in the database (0, if it is an insert). When an insert is made in *info*, table *metadata_info* is also populated so that the notification system can check the flag. Once the data is processed by the AD sensor, the flag is reset to a neutral value.

Once the notification systems are in place, the AD sensor is informed of any changes that happen on the file system or in the database and processes them accordingly. For our experiments we considered the *www1* host, the case of static files and two types of modeling the file content, keeping either the hash value of a file (md5 in our case) or the set of all the n-grams extracted from the byte content of a file (an Anagram model). We are interested in evaluating three different aspects: how fast the multi-granular updating process is, how much space the models require and how the false positive rate is improved by the multi-granular updating process.

Figure 5.15 presents the training cost in the case of the two network-based anomaly detection systems used before: Anagram (both standalone and coupled with the training dataset sanitization technique) and Payl.

Table 5.5: *Example of a trigger definition.* MySQL trigger definition for an insert event on a table *info*. First a mirrored table is created adding a state flag to it as well. When an insert event occurs the mirrored table is also populated with the inserted elements and the flag is set accordingly.

```
CREATE TABLE metadata_info LIKE info;
ALTER TABLE metadata_info ADD COLUMN flag TINYINT;
```

```
DELIMITER |
CREATE TRIGGER insert_info_trigger AFTER INSERT ON info
    FOR EACH ROW
    BEGIN
    INSERT INTO metadata_info SET metadata_info.email=NEW.email,
    metadata_info.abstract =NEW.abstract, [.....],
    metadata_info.flag=0; // flag ==0 if insert
    END;|
DELIMITER ;
```

For the same web server, *www1*, for which we created the Payl and Anagram models, we considered our multi-granular model approach. Figure 5.16 presents the initial effort in building the multi-granular models. For our experiments we built n-grams models for the html and htm files and for the rest of the static files we used the md5 hash models. Table 5.6 presents the space and time constrains in order to create the initial model for all the files requested in a 24-hour time frame (12GB of data on the file systems; also we only had access to 23,879 static files on the web server to do our evaluation). As expected, the n-gram approach requires more time and space than the md5 hashes. The reduction in space utilization for the md5 hashes is significant, but the number of unique grams is also significantly lower than the total number of grams over all files (see figure 5.17).

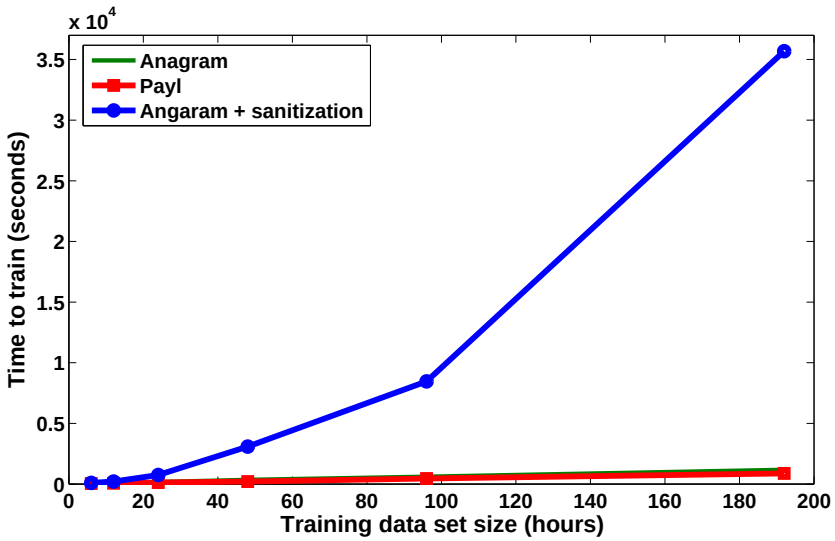


Figure 5.15: *Time to train an AD sensor for different training data set size.* For the case of Anagram+sanitization, we present the initial effort of building the first batch of micro-models and the sanitized model

Table 5.6: Time and space constraints for building multi-granular models

Type	Number of files	Space	Time
n-gram	6,345	22.64 bytes / 5-gram	0.136μs / 5-gram
md5	17,534	32 bytes / file	0.033μs / byte
total	23,879	590 MB	8 min 31.22 s

In order to analyze how our system performs in terms of the updating process, we tested the multi-granular model against real incoming traffic (24 hours of traffic, 380,052 network packets). The processing was not done at the packet level, as it was for the case of Payl and Anagram, but at the flow level. To this end, we implemented a TCP flow reconstruction tool to extract the request/response pair. As our model maps all the possible requests to the sub-models representing each file in the file

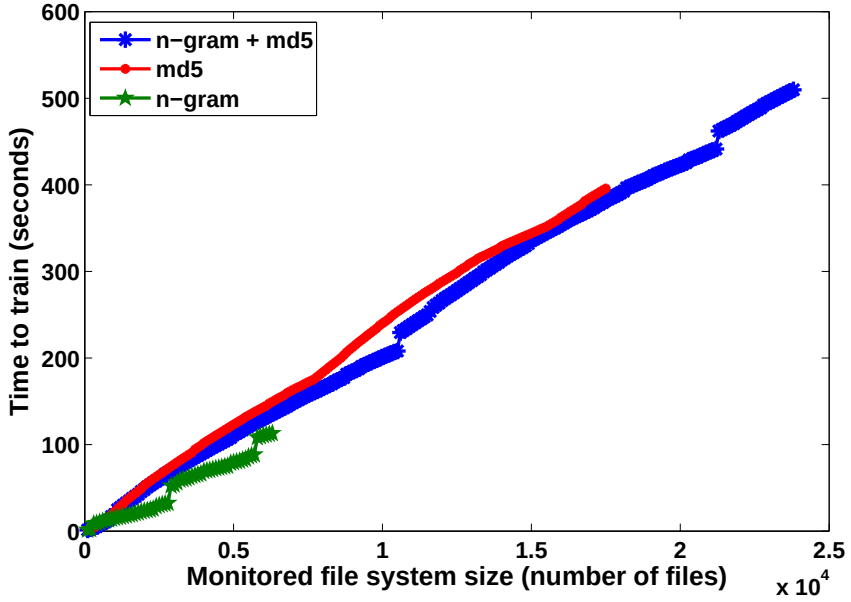


Figure 5.16: *Time to train our multi-granular model.* We have n-gram models for the html and htm files and md5 models for the rest of the static files.

system, it correctly identified all legitimate HTTP responses to legitimate HTTP requests. Alerts were raised for requests that didn't have a correspondent file on the file system or were malformed. For the n-gram approach the testing phase takes 16.08 μs / 5-gram on average, including the time to load the n-gram sub-models, while for the hash-based approach, testing a byte of data is done in 2 μs.

5.4 Post-Patch Model Update

Our goal in this section is to analyze the feasibility of building a mechanism that provides enough information to update a given host-based AD model after a patch is installed. The main assumption is that, if the patch employs only minor tweaks, translation proxies, or shim code, than then it seems possible to construct an AD model update procedure that inflicts small changes in the model.

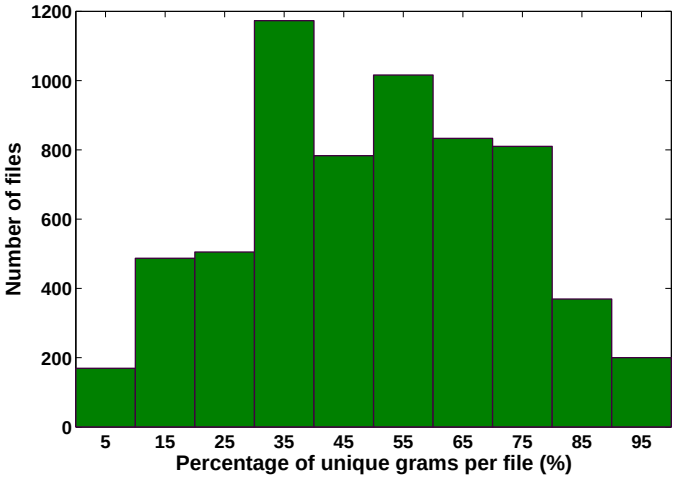


Figure 5.17: Histogram of percentage of unique grams out of the total number of grams in the files

The key problem for post-patch model update is translating from a static description of the expected change in behavior (as expressed by the patch as a change in the software) to the dynamic description of events contained in the model (the model is built while an application is executed). In order to make this discussion concrete, we define a basic, straightforward context window AD model that contains aspects of both data and control flow.

The model employs a context of m function instances to predict the occurrence of other function instances. That is, the model can be logically represented as a table of entries of the form:

$$\{f_i(args, rval), \ldots\} \rightarrow \{f_k(args, rval)\} \tag{5.2}$$

The conditional probability of f_k occurring with a particular set of arguments $args_k$ and return values $rval_k$ is based on the proceeding context of m (which can vary) functions[2]. The simplest case is based on monitoring only sequences of system

[2]Although we restrict our examination to a host–based model, examining the impact that patches

call names or numbers for each process. Modeling both the system calls and the arguments to those calls allows the model to improve its granularity. In addition, we can model library and application function calls and their parameters. One way to model the relationship between calls and arguments is to calculate the aggregated conditional probabilities between specific calls and arguments as presented by Stolfo *et al.*[98].

While it may be fairly straightforward to adjust control flow based directly on the information contained in a patch (*e.g.,* an insertion or removal of a function call), characterizing changes to the data sets representing the arguments or return values represents a more challenging task, and some pathological cases exist. For example, the dynamic behavior of a patch might be such that the application processes a completely different distribution of input data or produces radically different output data. Entries in the model for functions that process such data may now have outdated character distribution models or constraints for their arguments. Arbitrary and widespread behavioral changes will likely perturb the model beyond our ability to micro–patch it. In these cases, simply retraining by replaying a "sanitized" input archive may represent the best option.

We make the simplifying assumption that security–critical patches do not widely perturb the model or constraints on data arguments. Our examination of patches in the next section bears this hypothesis out. However, in cases where dataflow does drastically change between "known" data distributions, we may be able to automatically or manually annotate the model patch with these change types. A model patch can be bootstrapped from the patch text, then improved manually or via symbolic execution — in this case, manually improving the model patch and applying it will still likely result in a faster update of the model rather than complete retraining.

---

have on n-gram based network content models is an interesting area for future work.

5.4.1 Feasibility Study

Our evaluation contains two assessments. First, we review some anomaly and specification–based sensors to discover their supervised or unsupervised training time. We do so to confirm our hypothesis that such systems have relatively long training periods that occasionally require significant user input or supervision (in the previous sections we discussed mostly network based AD sensors, here the focus is on host-based sensors). In some cases, including Systrace [81] and modern PC firewalls that employ a user–driven training mode, supervision can span hours or weeks. Second, we summarize the changes in data and control flow enacted by a series of security–critical patches.

Cost of Training

We can classify training cost based on two broad categories: supervised and unsupervised. An unsupervised training phase usually requires several thousand of requests. Moreover, some host–based detection systems impose an additional one–time overhead due to static analysis. Furthermore, when a host–based sensor employs dynamic analysis, there is also a per–request latency that can lead to several hours of offline training. Supervised AD systems require user input to drive the training process. In such systems, it is very difficult to quantify the effort required to train the system since it depends on user activity. It is clear, however, that such systems require input from multiple users over a long period of time [81; 33] before they can generate a normality model capable of differentiating between normal and abnormal behavior.

Security Patch Survey

A patch can affect a behavioral model by changing either or both the control and data flow. Examples of changes in control flow include updating, removing, or introducing new decision control structures; introducing a new child function; or inserting a new parent function (*e.g.,* a sanity check on input parameters). Changes in data flow

include adding new variables or symbolic values; adding or removing arguments or function parameters; and modifications to the set of possible return values. We note that our examination is strictly static: it does not execute the patches. In addition, we do not distinguish between macros and function calls.

Table 5.7 lists our results for a variety of applications, including stunnel [100], some web servers [35; 74; 2], linux [55], cvs [18] and fetchmail [82], as well as various vulnerabilities in libpng [69], Firefox [46], and Samba [88]. The Δs are computed by counting the number of control and data flow changes as defined in tables 5.8 and 5.9.

Table 5.7: *Survey of patches.* We list the vulnerable version of an application, the size of a patch in lines (including comments), and the changes in data and control flow introduced by the patch, as listed above. The magnitude of the difference between the changes and the application's total size supports the notion that patches introduce relatively confined model updates.

Application	**Patch Size (lines)**	**control flow Δ**	**data flow Δ**
Linux-2.4.19	20	3	1
ghttpd-1.4	16	4	5
nullhttpd-0.5.0	12	2	1
stunnel-3.21	29	0	3
libpng-1.2.5	98	10	12
cvs-1.11.15	81	1	2
Apache-1.3.24	11	0	1
fetchmail-6.2.0	183	1	5
Samba (CVE-2004-0882)	65	0	7
Samba (CVE-2004-0930)	386	99	39
Firefox-2.0.0.3	22	8	0

As our results would infer, we conjecture that most security–critical patches enact

Table 5.8: Control flow changes introduced by patches

Control Flow Changes
- new decision control structures
- new decision conditions
- new child functions
- replacement of a function
- insertion of parent (like a sanity check)
- jump to error handler case

Table 5.9: Data flow changes introduced by patches

Data Flow Changes
- new variables
- new arguments
- deleted arguments
- new return values
- deleted return values

small changes to the system that only affect or invalidate correspondingly small parts of an application's behavioral model. If this hypothesis is correct, then it seems possible to construct an AD model update procedure that *derives the necessary changes from the text of a patch itself*[3]. The key challenge is to notify the AD about a patch in terms that it understands: changes in control and data flow. This challenge is the essence of automatic post-patch AD model update.

[3]We can utilize the actual patching procedure to recover the context of the changes and a limited form of parsing or symbolic execution to gather information about data flow changes.

A Model Update Procedure

Most of the control flow changes we observed resulted from invocations of new functions as well as the insertion of new `if` statements or updates of `if` conditions. Most data flow changes involved new arguments to function calls, or new ways of wrapping those arguments, as well as new `return` statements that introduced new values. A majority of the patches we examined made very minor changes; for example, the patch to ghttpd substitutes the use of a "safe" library function and derives the value of a new argument for that call. The patch for nullhttpd introduces a new `if` statement and condition with a call to an application function to log an error (presumably, the dynamic behavior also involves the invocation of the library `printf()` family of functions and the `write()` system call). We can use a parsing and symbolic execution phase to learn and summarize these implicit changes. We envision generating model patches in a format similar to source code patches like those produced by `diff`. Model patches contain update summaries to the conditional probability entries in the model, along with changes to the format of the arguments and insertion and removal of functions from a call chain.

5.5 Summary

As the system under protection evolves over time its behavior becomes more and more inconsistent with the AD model, eventually rendering the AD sensor unusable if the model is not updated. To amend this, we proposed two different approaches: *progressive model updates* and *induced model updates*. The first one consists of an online learning technique that *progressively* updates the normal model while integrating our sanitization and automation methods. This technique is general and can adapt to legitimate changes that the users infer on the system, aside from changes in the system itself. For the second approach, changes in the AD models are *induced* by changes observed internally in the system through, but not limited to, three entities:

file systems, databases and software patches. These changes are assumed to affect only certain regions in the model, which can be isolated and treated separately in the updating process. This method is specialized for certain scenarios (*e.g.* web server, software changes *etc.*) and assumes that the AD mechanism has direct access to the monitored entities through a notification mechanism.

In the case of progressive updates, the adaptation process consists of re-generating new time granularity and voting threshold values and also creating new normality models (using new and older micro-models in the voting process), which we refer to as "dynamic" models. We have shown that, without this adaptation process, legitimate changes in the systems are flagged as anomalous by the AD sensor leading to an inflation of alerts. Over a very large time window, our dynamic model generation maintains a low alert rate (1%) as opposed to a 7% for a system without updates.

For the induced updates, we analyzed the case of file systems and databases independently from the case of patches. We first investigated the feasibility of automatically deriving and applying a "model patch", based on information related to alterations in the file system and database. We presented baseline results on how to update a sensor that monitors the request and response messages for non-dynamic HTTP requests. The results showed that, by using a multi-granular modeling technique, we can achieve fast initial training along with fast automatic updating phase triggered by the notification system that monitors both the file system and the database. In addition, we proposed extensions for dynamic, database–driven requests and responses. Second, we investigated the feasibility of automatically deriving and applying a different type of model patch that describes the changes introduced by a code patch installation. We examined 11 security–critical patches to obtain an idea of how to summarize the data and control flow changes necessary to update a behavioral model and proposed a model update procedure. For future work, we intend to implement this model update procedure for different host–based AD sensors.

Chapter 6

AD Model Exchange

As presented in chapters 3, 4 and 5, AD models' performance can be improved by sanitizing their training datasets, AD sensors can be automated by determining their operational parameters using the intrinsic characteristics of the system under protection and, last but not least, AD models can be updated such that they adapt to changes in the system behavior. In this chapter, we expand the notion of automated AD optimization to include collaboration between multiple systems.

The key benefit of *Collaborative Security* [95; 75; 89; 48; 73; 103] to intrusion detection is "a better view of global network attack activity" [57], as opposed to the single site view. By leveraging the location diversity of collaborating sites, a more precise model of an attacker's behavior and intent can be provided. Commercial companies [37] are now offering distributed security collaborative systems of the sort proposed several years ago by Locasto*et al.* [57]. Other examples of organizations employing a collaborative approach to defense include the Department of the Treasury and the FSISAC, leveraging the talent and resources of the financial community to protect the community as a whole. The DISA/DoD manages and leverages information from its many customer .mil hosts, while universities with hundreds of divisions and units across schools and departments also follow this model.

However, existing work mostly focuses on sharing alerts triggered by different

intrusion detection systems. While the *exchange of alerts* can successfully improve the capabilities of detecting distributed or multi-stage attacks and, in some cases, reduces the false positive rate, there are scenarios for which this solution might not be suitable or optimal. Consider a dynamic environment in which nodes continuously enter and leave the network; an alert exchange mechanism is not suitable for such a short span environment. Another method for communicating the state of each live node is necessary. For a centralized alert exchange mechanism, a large volume of alerts can be overwhelming, thus the need of reducing or synthesizing the volume of alerts. An alternative approach is to *exchange AD models*, which have the ability to characterize any normal or malicious/abnormal behavior as IDS alerts can be synthesized in the models. We are interested in the protection gain achieved when AD models are shared in order to provide a global view on potential threats or environment specific normal behavior.

6.1 Contributions

For our model exchange applications, we analyze environments with different constraints. In the first scenario, we assume that the collaborative sites fully trust each other and that the identities of individual collaborators are not known to the outside world. In this case, the collaborating sites exchange models of bad behavior that are used locally to further improve the data sanitization process. Even if the identities of the collaborating sites become known, attacking all the sites with targeted or blending attacks is a challenging task. The attacker will have to generate mimicry attacks against all collaborators and blend the attack traffic using the individual sites' normal data models.

The second model exchange application proposes a modeling approach to characterize application behavior, where models are shared between different applications under the assumption of application diversity. This infrastructure measures the ca-

pability of a modeling technique to capture the specific behavior of each application based on the return values exhibited with each run.

The common trait between these approaches is the exchange and corroboration of AD models. However, before a collection of models can be used in a sharing paradigm between sites, devices, hosts or applications, a set of operations needs to be defined that allows extracting the significant information from the exchanged models. In the next section, we will establish a set of possible operations that can be applied on any two given models, and for our applications, we will use them accordingly. The goal is to create a general framework for model operations that can also be suitable for future model exchange approaches.

The main contributions presented in this chapter are the following:

- first, we formalize a set of tools for model operations that can be utilized in AD model exchange scenarios. Different AD model operations have been used by the research community, but no unified set of possible operation has been provided.

- second, we introduce and analyze different model exchange applications for which the AD model exchange process provided a significant improvement over the single site use in the collaborative environment.

6.2 AD Model Operations

AD model operations have been used in the literature [5; 30; 29; 68; 41], but no formal classification was ever made over all possible operations. In this section we present a general set of operations, which have either a new model or a score as the final result. Defining such a set is not a trivial task, given that every operation, when applied on a model type, needs to be adapted to the model's internal structure, thus the need of considering two versions of each operation:

- a *direct* version, that can be applied to models that allow a direct operation, *i.e.* given two models, there is a direct translation from them to a new model/score that represents the operation result;

- an *indirect* version, that applies for the case of more complex models, *e.g.* probabilistic or statistical models, for which the model operations cannot be applied analytically, but *indirectly*. In this scenario, the two input models are used in a testing phase over a dataset, T (*e.g.* either one of the datasets used for training the given models or a new one). We treat the AD sensors that correspond to the models as black boxes, and base our results on the agreements/disagreements on the tested data points. This version could potentially require a higher computational overhead than the direct approach. However, the additional effort is offset by the general applicability of the operations, placing no restrictions on the inner modeling technique used by each sensor.

6.2.1 Model Aggregation

The first operation defines the notion of *model aggregation*. Given two models M_1 and M_2, trained using the same AD algorithm, we want to construct a new model that represents the aggregate between the two of them. Thus, we combine the characteristics of the two models in order to construct a model that represents both behaviors. The result is a new model, M_{aggr}. For example, in the case of Anagram models, the aggregated model contains all the n-grams present in both input models. In case the models are complex and they do not permit the direct operation, their aggregation can be defined indirectly, by generating a new training dataset for the result model, T_{aggr} (see table 6.1).

$$M_{aggr} = M_1 \bigcup M_2 \tag{6.1}$$

where $M_1 = AD(T_1)$, $M_2 = AD(T_2)$ and T_1 and T_2 are training datasets used for training the models.

Table 6.1: *Indirect Model Aggregation.* T_{aggr} contains all the data points that were deemed normal by either of the models.

```
routine indirectAggregation(T, M1, M2)
Taggr ← {}
for all di in T
    if 1=test(di, M1) or 1=test(di, M2)
        Taggr ← di ∪ Taggr
Maggr ← AD(Taggr)
return Maggr
```

6.2.2 Model Intersection

The second operation defines the notion of *model intersection*. Given two models M_1 and M_2 trained using the same AD algorithm we want to construct a new model that represents the common behavior between the two of them. Thus we intersect the characteristics of the two models in order to construct a model that represents the common behavior. The result is a new model, M_{int}. For example, in the case of Anagram models, the intersection model contains the common n-grams between the two input models. In case the models do not permit the direct operation, the intersection can be defined indirectly as presented in table 6.2.

$$M_{int} = M_1 \bigcap M_2 \tag{6.2}$$

where $M_1 = AD(T_1)$, $M_2 = AD(T_2)$ and T_1 and T_2 are training datasets used for training the models.

Table 6.2: *Indirect Model Intersection.* T_{int} contains all the data points that were deemed normal by both models.

```
routine indirectIntersection(T, M1, M2)
T_int ← {}
for all d_i in T
    if 1=test(d_i, M1) and 1=test(d_i, M2)
        T_int ← d_i ∪ T_int
M_int ← AD(T_int)
return M_int
```

6.2.3 Model Differencing

Another operation defines the notion of *model differencing*. Given two models M_1 and M_2 trained using the same AD algorithm we want to construct a new model that represents the behavior of one, but not the other one. Thus we difference the characteristics of the two models in order to construct a model that represents the difference behavior. The result is a new model, M_{diff}. For example, in the case of Anagram models, the difference model would contain only the n-grams that are present in M_1, but not in M_2. In case the models do not permit the direct operation, the intersection can be defined indirectly as shown in table 6.3.

$$M_{diff} = M_1 - M_2 \tag{6.3}$$

where $M_1 = AD(T_1)$, $M_2 = AD(T_2)$ and T_1 and T_2 are training datasets used for training the models.

Table 6.3: *Indirect model differencing.*T_{diff} contains all the data points deemed normal by M_1 and abnormal by M_2

.

```
routine indirectDifferencing(T, M1, M2)
Tdiff ← {}
for all di in T
    if 1=test(di, M1) and 0=test(di, M2)
        Tdiff ← di ∪ Tdiff
Mdiff ← AD(Tdiff)
return Mdiff
```

6.2.4 Model Similarity

We now define an operation that doesn't have as a result a model: *model similarity*. Given two models M_1 and M_2, trained using the same AD algorithm, we want to determine how similar the models are to each other. This is a means of deciding if two different behaviors that are represented by the models have the same characteristics. The result is a similarity score, $sim_{1,2}$, with bounds depending on the used distance metric (normalization might be applied). For example, in the case of Anagram models, the similarity metric can be the number of common n-grams among the two input models. There is also a indirect version of this operations presented in table 6.4.

$$sim_{1,2} = dist(M_1, M_2) \tag{6.4}$$

where $M_1 = AD(T_1)$, $M_2 = AD(T_2)$ and T_1 and T_2 are training datasets used for training the models.

Based on the previous operation, we can expand the list of model operations to *model clustering*. If data points can be assigned to different clusters so that data points in the same cluster are similar, the same procedure can be applied to models.

Table 6.4: *Indirect model similarity.* $sim_{1,2}$ counts how many times the two models agree. This metric can be normalized by defining the percentage of agreements out of the total number of tested data points.

```
routine indirectSimilarity(T, M1, M2)
sim1,2 = 0
for all di in T
    if (1=test(di, M1) and 1=test(di, M2)) or
      (0=test(di, M1) and 0=test(di, M2))
        sim1,2 = sim1,2 + 1
return sim1,2
```

Each cluster contains models that characterize similar behaviors.

6.3 Model Exchange for Cross-sanitization

Chapter 3 noted that our local sanitization architecture has a weakness in the presence of long-lasting training attacks in the training data. Because attacks of this type may span multiple micro-models, a large portion of the micro-models might be poisoned. Since we predicate our cleaning capability on micro-model voting, extensive poisoning of the training data would seriously deteriorate our ability to detect long-lived or frequently occurring attack payloads. We hypothesize, however, that the distribution of such long-lived attacks among Internet hosts would require an adversary with significant resources (*e.g.,* a potentially large number of source IP addresses) — a requirement that effectively limits the scope of such attack to few target hosts or networks.

Given this hypothesis, we can counter the effects of such attacks by extending our sanitization mechanism to support sharing models of abnormal traffic (or an

alert model) among collaborating sites. Sharing these models enables a site to re-evaluate its local training data[1]. Our goal is to enhance the local view of abnormal behavior characteristics (rather than normal behavior characteristics, which cannot be meaningfully shared because they are unique to an individual site). As we will show, *cross-sanitization* between sites boosts our ability to remove long-lived or frequent attacks from the training data (regardless of whether or not the attack data is "targeted", *i.e.,* injected specifically to blind the sensor).

In some sense, attack vectors that saturate training data define normal traffic patterns. Local knowledge alone may not provide enough evidence to weed out consistent attack vectors in training data. To isolate and remove these vectors, we need to incorporate knowledge from some other remote source. This information sharing is the essence of cross-sanitization: comparing models of abnormality with those generated by other sites (see full architecture in figure 6.1).

Cross-sanitization compares models of abnormality because normal models are tightly coupled with an individual site's traffic. In contrast, the consistency of characteristics of abnormal packets across sites can help filter out attacks that saturate the training data. Individual sites can utilize this external knowledge to cross-sanitize their training set and generate a better local normal model.

For an attacker to successfully blind each sensor in this type of environment, she would need to identify each collaborator and launch the same training attack on all participating sites for the same time period. Accomplishing this goal requires a significant amount of knowledge and resources. Even in the case of botnets exhibiting the same behavior with a footprint of up to 350,000 infected machines, as reported by Dagon *et al.* [19], a botnet training attack would still impose a significant cost in terms of resources on the attacker side, given that a botnet training attack is akin to a probing attack. If the same resource is used against multiple collaborators, they

[1]To alleviate the privacy concerns of sharing content, these models may incorporate privacy-preserving representations [75].

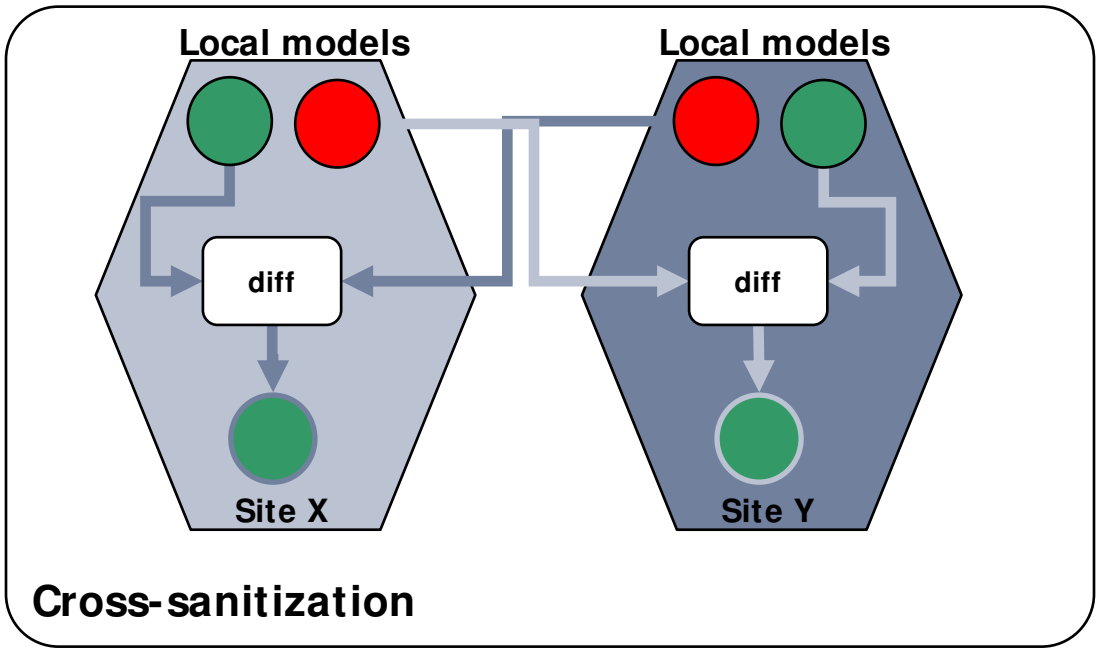


Figure 6.1: *Cross-sanitization architecture.* The local sanitized and abnormal models are produced as shown in figure 3.1. In the cross-sanitization the difference operation is applied between the local sanitized model and the remote abnormal model.

may notice common source IPs of potential botnet members and become aware of the attack. Therefore, we postulate that when a particular site experiences a targeted training attack, the attack data will not appear at all collaborating sites at the same time. As a result, with a large enough group of collaborators, some fraction of sites will have seen the attack, but will *not* have had their model corrupted by it. In this case, sharing abnormal models helps cleanse the local models of sites in the group that have been corrupted. When a site with sanitized model M_{san} receives the abnormal models $M_{abn_1} \ldots M_{abn_M}$ from its collaborators, it needs to apply the model differencing operation to compute a new model, M_{cross}.

$$M_{cross} = M_{san} - \bigcup \{M_{abn_i}\} \tag{6.5}$$

where M_{san} is the locally sanitized model and M_{abn_i} are the remote abnormal models exchanged between collaborators.

The differencing of models may be applied either directly or indirectly across sites, depending on the particular AD algorithm in use and the specific representation of the model.

Polymorphic attacks present a special challenge because each propagation attempt will display a distinct attack vector that may be captured in different abnormal models. We conjecture, however, that a polymorphic attack targeting a single site can still be captured by the local sanitization scheme as presented in section 3.4.1.

Although direct/indirect cross-sanitization can help identify abnormal samples that have poisoned a site, we must take care during this process. We conjecture that sites may exhibit content diversity (*i.e.,* they do not experience identical traffic flows); thus an abnormal model from site B may include some common but ultimately legitimate data from site A. In other words, data items that are indeed normal for a particular site can be considered abnormal by others. If site A attempts to identify abnormal content in its local model using cross-sanitization with site B, then A may incorrectly remove legitimate data patterns from its model along with truly abnormal or malicious data patterns. Doing so increases the false positive rate — an increase that may not be matched by an increase in detection rate.

An alternative approach to reconciling different models or disagreements between models involves the use of a shadow server. When the sanitized model and an abnormal model disagree on the label of a packet (for example, the sanitized model labels it normal and the abnormal one as abnormal), we redirect the traffic to the shadow server to determine if the packet causes a real attack. Based on this information the packet is used in the construction of either the local sanitized model or the local abnormal model.

6.3.1 Evaluation

This section shows that even if local sanitization fails to detect an attack, we can compensate by using the external information received from other collaborating sites. Furthermore, we show that the performance of the local architecture remains unaffected when faced with polymorphic attacks. The experiments in this section use the Anagram sensor and the traffic from the previously presented hosts, *www*, *www1*, and *lists* (see chapter 3), and were conducted on a PC with a 2GHz AMD Opteron processor 248 and 8G of RAM, running Linux.

We will assume that at least some of the collaborative sites are poisoned by a long lasting training attack, while others were able to filter it and use it for building the abnormal model. If the targeted site receives an abnormal model that contains an attack vector, the local sanitized model can be "cross-sanitized" by removing the common grams between the two models (direct cross-sanitization). Given the diversity in content exhibited by different sites, the same gram can be characterized differently by different sites. Therefore, it is possible that after cross-sanitation the sanitized model becomes smaller. As an immediate consequence, the false positive rate will increase.

We consider all the possible cases in which each of our three hosts model is poisoned by each of the four attacks present in our data. When one site is poisoned, we consider that the other two are not. Every poisoned host receives the abnormal models M_{abn} of its peers in order to cross-sanitize its own model, M_{pois}. Table 6.5 presents the average performance of the system before and after cross-sanitization when using direct and indirect cross-sanitization.

In the case of direct operations, once the cross-sanitization is done, the detection rate is improved, but the false positive rate degrades. To further investigate how the cross-sanitization influences the performance of the local systems, we analyze the size of the models (presented in Table 6.6).

As Table 6.6 shows, the size of the models has decreased. This decrease leads to

Table 6.5: Performance when the sanitized model is poisoned and after it is cross-sanitized when using direct/indirect model differencing

Model	www1		www		lists	
	FP(%)	DR(%)	FP(%)	DR(%)	FP(%)	DR(%)
M_{pois}	0.10	44.94	0.26	51.78	0.10	47.53
$\boldsymbol{M_{cross}}$ **(direct)**	**0.23**	**100**	**0.70**	**100**	**0.16**	**100**
$\boldsymbol{M_{cross}}$ **(indirect)**	**0.10**	**100**	**0.26**	**100**	**0.10**	**100**

an increase in the FP rate. As we mentioned before, this behavior is a disadvantage of our distributed sanitization method, as it depends on site diversity. Furthermore, this phenomenon provides a potential avenue of attack for an adversarial collaborator. We consider defending against this type of attack to be out of the scope of our current efforts, but Byzantine robustness or reputation systems [42] can be applied in the future. Of course, even appropriately authenticated or otherwise trustworthy peers could be exploited after they are included in the collaborative network. We note that dealing with trusted insiders is an inherently hard problem faced by many large-scale collaborative systems, from anonymity networks to mandatory access control frameworks.

In order to improve our method for cross-sanitization, we can use the indirect model operations approach. This approach tests the poisoned local model and the collaborative abnormal models against the second training dataset used in our local methodology. The goal of this method is to eliminate the packets responsible for poisoning the local model from the training data set. As we can observe in table 6.5, the FP rate has improved and the detection rate remains at 100%. The improvement of the FP rate is reflected in the size of the cross-sanitized models (see table 6.6).

In terms of computational performance, as expected, the indirect model cross-

Table 6.6: Size of the sanitized model when poisoned and after cross-sanitization when using direct/indirect model differencing

Model	www1		www		lists	
	#grams	file size	#grams	file size	#grams	file size
M_{abn}	2,289,888	47M	199,011	3.9M	6,025	114K
M_{pois}	1,160,235	23M	1,270,009	24M	43,768	830K
$\boldsymbol{M_{cross}}$ **(direct)**	**1,095,458**	**21M**	**1,225,829**	**24M**	**37,113**	**701K**
$\boldsymbol{M_{cross}}$ **(indirect)**	**1,160,006**	**23M**	**1,269,812**	**24M**	**43,589**	**828K**

sanitization is more expensive than the direct one (see Table 6.7). There is a tradeoff between how fast the cross-sanitization needs to be done and how high the FP rate is. If a higher FP rate is allowed, a quicker cross-sanitization can be applied by using the direct version; otherwise the best solution is the indirect operations.

Table 6.7: Time to cross-sanitize for direct and indirect model operations

Method	www1	www	lists
direct	13.98s	26.35s	16.84s
indirect	1966.68s	1732.32s	685.81s

6.4 Model Exchange between Applications

The growing sophistication of software attacks has created the need for increasingly finer-grained intrusion detection systems to improve the overall software protection mechanisms. This implies moving anomaly sensors beyond the network or even host level, to monitoring the applications themselves. However, such fine-grained mecha-

nisms are currently perceived as too expensive in terms of their performance impact, and questions relating to the feasibility and value of such analysis remain unexplored.

In this section, we attempt to demonstrate the feasibility of building application profiles at a fine-grained level of detail by analyzing the resulting models' capability in discriminating between applications. We apply the model operations framework presented in section 6.2 to the case of application profiles based on function return values. We focus on this feature of application behavior because return values help drive control flow decisions, which in turn define the dynamic characteristics of the application. Profiles of this type are built at the binary level — that is, without making changes to the application's source code, the operating system, or the compiler. Such modeling techniques are suitable for an overall system protection mechanism that monitors each application individually and determines when the system is under attack.

6.4.1 Implementation and Evaluation

The frequency distributions of return values were captured using a Pin [59] tool occurring in a selection of real software. These distributions provide insight into both the micro and macro structure of a return value behavior model. Our analysis aims to show that return values can reliably classify similar runs of a program as the same program as well as distinguish between execution models of different programs.

The Pin tool intercepts the execution of the monitored process to record each function's return value. The tool builds a table of return value frequencies. After the run of the program completes, we create a final return value frequency model. As a proof of concept, a model for a particular monitored process is simple, consisting of the average frequency distribution over multiple runs of the same program.

We examine two hypothesis dealing with the efficacy of execution behavior profiles based on return value frequency:

1. traces of the same program under the same input conditions will be correlated

Table 6.8: *Manhattan distance within and between models.* The diagonal (shown in italics) displays the average distance between each trace and the behavior profile derived from each trace of that program. All other entries display the distance between the execution models for each program. We omit the lower entries because the table is symmetric. Note the difference between gzip and gunzip as well as the similarity of gzip to itself.

	date	**echo**	**gzip**	**gunzip**	**md5sum**	**sha1sum**	**sort**
date	*3.03e+03*	3.72e+03	1.61e+07	1.87e+06	6.46e+04	6.47e+04	5.45e+03
echo	-	*548*	1.61e+07	1.87e+06	6.41e+04	6.42e+04	5.43e+03
gzip	-	-	*212.4*	1.79e+07	1.61e+07	1.61e+07	1.61e+07
gunzip	-	-	-	*1.91e+04*	1.92e+06	1.92e+06	1.87e+06
md5sum	-	-	-	-	*3.03e+04*	3.38e+04	6.56e+04
sha1sum	-	-	-	-	-	*1.67e+04*	6.57e+04
sort	-	-	-	-	-	-	*4.24e+03*

with their model;

2. the model of all traces of one program can be distinguished from the model of all traces of another program.

For the first hypothesis, we use Manhattan distance as a similarity metric in order to compare each trace of the same process with the return value model of that process. In effect, we compare each return value frequency to the corresponding average frequency among all traces of that program. To evaluate the second hypothesis, we use the Manhattan distance between each process model. The base set for the return values consists of all return values exhibited by all processes that are analyzed over all their runs. Table 6.8 shows how each model for a variety of program types (we include a variety of programs, like sorting, hashing, simple I/O, and compression) stays

consistent with itself under the same input conditions (smaller Manhattan distance) and different from models for each other program (larger Manhattan distance).

Each process has a particular variance with each trace quantified in the similarity value between its model and the trace itself, but when compared against the rest of the processes it can be easily distinguished. We ran each program ten times under the same input profile to collect the traces and generate the model for each program. We used as input profiles generic files/strings that can be easily replicated (in some cases no input was needed): date - N/A, echo - "Hello World!", gzip - *httpd-2.2.8.tar*, gunzip - *httpd-2.2.8.tar.gz*, md5sum *httpd-2.2.8.tar*, sha1sum - *httpd-2.2.8.tar* and sort - *httpd.conf* (the unmodified config file for httpd-2.2.8).

6.5 Summary

Collaborative security is a powerful concept as it provides an ensemble view of attackers' intentions by leveraging site/device/application diversity. The process of exchanging AD models in a collaborative setting, as described in this chapter, enables systems to communicate either normal or abnormal/malicious behavior profiles, in order to improve their protection mechanisms.

The first step towards using AD models in a distributed setting is to formalize a set of operations that can be applied on the models; these operations can later be assembled to produce the desired collaborative algorithm:

- *model aggregation* combines two input models into a model that represents both behaviors;
- *model intersection* constructs a model that represents the common behavior encapsulated in two input models;
- *model differencing* constructs a new model that represents the behavior of one of two input models, but not the other one;

- *model similarity* provides a measure of the similarity between the behavior profiles encapsulated in each of the two input models.

The implementation of these operations depends on the type of AD model. Along these lines, we distinguish between two categories of sensors. The first one uses models that allow analytical operations for adding or removing content; we refer to this implementations as *direct operations*. Examples include sensors such as Anagram, where a model can be directly manipulated by adding or removing n-grams. However, our framework is not restricted to this type of models. In the case of sensors where analytical operations are not possible, such as the ones producing statistical or probabilistic models, we have implemented the same set in the form of *indirect operations*, treating the sensor as a black box and using an additional data set for operations. This makes for a general framework, rendering the following results applicable in a wide variety of environments.

For the first application of model exchange, referred to as *cross-sanitization*, we showed that this method can help mitigate the risks of local anomaly detectors being evaded by targeted training attacks. Our approach is to share models of abnormal traffic among collaborating sites that can be used to further improve the sanitization process described in chapter 3. A site can cross-sanitize its local training data based on the remote models, using the aggregation and differencing operations. Our results show that if one of the collaborating sites was targeted by the same attack and was able to capture it in its abnormal model, the detection rate can be improved up to 100% for all the collaborators.

The second application that we present focuses on application-level behavioral modeling. In this case, AD models are used to confirm that applications conform to their typical behavior characteristics. This goal requires a method for building behavioral profiles that is substantially different from the network-level cases described before: application profiles are based on the frequency distribution of function return values. However, the same general principles of AD model corroboration apply. We

have demonstrated that these models can be shared among a set of real software applications as they are consistent within a set of multiple application runs, and that they are discriminative between different applications. Based on the successful application of the AD model exchange principles in differnt environments, we conjecture that the same approach can in the future be generalized for other types of profiling techniques as well.

Chapter 7

Conclusion

We have started this study by discussing anomaly detection as a first-class defensive technique, and a valid alternative to the commonly used signature-based detection mechanisms. In this thesis, we developed a framework that improves the performance of content–based AD sensors, making them more easily deployable while maintaining the vital ability of detecting zero-day or polymorphic attacks. At this point, we review the contributions of the thesis, summarize the main results and identify what we believe to be some of the most promising avenues for continuing this work.

7.1 Thesis Summary

Our first step was to introduce a novel data sanitization technique for content–based AD sensors, using the notion of micro-models as "normal" models trained on small slices of a training dataset. Using these models in weighted voting schemes, we significantly improve the quality of unlabeled training data, making it as "attack-free" and "regular" as possible. As a result, the sensor can use as training data an unlabeled, and thus potentially dirty, sample of incoming traffic. Our technique is agnostic to the AD algorithm used, and can therefore be extended to other AD sensors that model the content of a high volume of streaming data. The next step was

to further reduce the dependence on human operators, thus making content–based AD system more easily deployable in real-life scenarios. We enhanced the training phase of AD sensors to include not only a sanitization phase, but also with a self-calibration phase that automatically determines the sanitization parameters. Our results show that the sanitization scheme with automatically determined parameters achieves comparable results to the case of empirically determined optimal parameters.

To complete the deployment process, we also considered the case in which the system under protection evolves over time, rendering an AD model unusable unless modified to adapt to these changes. We proposed two different approaches: *progressive model updates* and *induced model updates*. The first one consists of an online learning technique that *progressively* updates the normal model while integrating our sanitization and calibration methods with an aging mechanism for micro-models. This technique is general and can adapt to legitimate changes in the behavior of the uses, as well as changes in the system itself. For the second approach, updates of the AD models are *induced* by changes observed internally in the system through three entities: file systems, databases and software patches. These changes are assumed to affect only certain regions in the AD model, which can be isolated and treated separately in the updating process. This method is specialized for certain scenarios (*e.g.* web server, software changes *etc.*) and assumes that the AD mechanism has direct access to the monitored entities through a notification mechanism.

Our view is that AD models can be considered first-class objects that can be manipulated and communicated. As such, they can also be utilized in a collaborative security approach which aims to provide an ensemble view of attackers' intentions by leveraging site/device/application diversity. As part of this effort, we formalized the set of operations that can be used to process AD models; we then applied these operations for algorithms that aim to improve security mechanisms by exchanging models of either normal or abnormal behavior. We proposed four different operations: *model aggregation*, *model intersection*, *model differencing* and *model similarity*.

We also distinguish between two categories of sensors. The first one uses models that allow analytical operations for adding or removing content; we refer to these implementations as *direct operations*. However, our framework is general and can also accommodate sensors where analytical operations are not possible (*e.g.* those producing statistical or probabilistic models). For these cases we have implemented the same set in the form of *indirect operations*, treating the sensor as a black box and using an additional data set for aggregation based on testing results.

The first application of model exchange that we present, referred to as *cross-sanitization*, showed that this approach can help mitigate the risks of local anomaly detectors being evaded by targeted training attacks. Our approach is to share models of abnormal traffic among collaborating sites that can use these models to further improve their local sanitization processes. A site can cross-sanitize its local training data based on remote models, using the aggregation and differencing operations.

The second application that we present focuses on application-level behavioral modeling. In this case, AD models are used to confirm that applications conform to their typical behavior characteristics. This goal requires a method for building behavioral profiles that is substantially different from the network-level cases described before: application profiles are based on the frequency distribution of function return values. However, the same general principles of AD model corroboration apply. We have demonstrated that these models can be shared among a set of real software applications as they are consistent within a set of multiple application runs, and that they are discriminative between different applications. Based on the successful application of the AD model exchange principles in different environments, we conjecture that the same approach can in the future be generalized for other types of profiling techniques as well.

7.2 Result Summary

In order to support the validity of the techniques presented above, we conducted a number of experiments, detailed in chapter 3, chapter 4, chapter 5 and chapter 6. We have learned the following lessons:

- When our training dataset sanitization method was employed, the studied AD sensors detected approximately 5 times more attack packets than without the sanitization method.

- In order to study the feasibility of a fully automated deployment mechanism, we compared the results obtained with empirically determines optimal parameters against those computed using the self-calibration phase. Modeling traffic from two different sources, the fully automated calibration shows a 7.08% reduction in detection rate and a 0.06% increase in false positives, in the worst case, when compared to the optimal empirically determined parameters.

- Without a model adaptation process, legitimate changes in the system are flagged as anomalous by the AD sensor leading to an inflation of alerts. Over a very large time window, our model update generation maintains a low alert rate (1%) as opposed to a rate of 7% for a system without updates.

- Using induced model updates results both in a fast initial training phase and a fast automatic updating phase. Updates can be triggered by a notification system that monitors both the file system and the database. For the case of updates induced by patches, we examined 11 security–critical patches which verified our conjecture that effected changes are localized in relatively small areas of the underlying AD model. We were able to summarize the data and control flow changes necessary to update a behavioral model and propose a patch-based model update procedure.

- Collaboration between multiple sites can alleviate the risk of training attacks,

where one site includes the attack in its normality model. Our cross-sanitization technique improved the detection rate up to 100% in the case where one site missed the training attack, but other collaborators detected it and labeled it as abnormal. We also showed that AD models can be shared among a set of real software applications as they are consistent within a set of multiple application runs, and that they are discriminative between different applications.

7.3 Future Work

In this thesis we have focused on a generalized framework for automated deployment of accurate anomaly detection sensors. We believe that achieving this goal will enable new applications in both the short and long terms, and also reveal a number of new and highly promising research directions.

In the concluding remarks of each chapter, we have presented a number of short term extensions that can be added to the methods presented in that chapter; we briefly review some of the most important ones here. For example, the sanitization process could potentially use more complex weighting strategies during its core voting process. The model update approaches can benefit from a micro-model clustering approach that minimizes the delay introduced by the sanitization technique in the update process. A parallel implementation of the online sanitization process can also help alleviate the computational effort for high traffic bandwidth. For induced updates based on patches, different implementations of the model update process for different host–based AD sensors are the logical next steps.

Looking beyond immediate improvements and refinements, we believe that one of the key characteristics of the presented work is its agnostic approach to the AD algorithm that is used. This makes it general and applicable to a wide range of sensors. More importantly, it can also enable it to combine multiple sensors in a unified framework, following the guidelines traced out in this thesis. The first step in

this direction will be to characterize more sensors, from the point of view of compliance with the interface defined in our work. We will then extend the voting process used for data sanitization to accept input from multiple content-based sensors during the voting process. In this thesis, we have presented self-adaptive AD methods that leverage diversity in both time (by using individual micro-models for self-sanitization) and space (by corroborating alerts from multiple sites). Adding a new dimension by integrating multiple core AD techniques seem a natural direction to follow.

So far, our main focus has been on content-based anomaly detection. In the future, we believe that a particularly promising direction is to use the experience we have gained to develop hybrid technologies that can leverage a larger spectrum of malware detection. We envision the use of a unified framework for multiple sensors operating at different levels(*e.g.* network level, host level, *etc.*). The sensor selection process should be done such that redundant sensors cannot be evaded by the same type of attack input. While we have not yet achieved this level of integration, it is our directional goal. Finally, an important aspect is that by correlating events triggered by different components of a hybrid approach we can determine the *intentions or the capabilities of the attackers*. Such profiles will be instrumental in developing appropriate response strategies.

Appendix A

Automated AD Sensors

A.1 Self-Calibration

Given a new packet in a live network stream, as well as a history of packets contained in the current micro-model, these function compute the likelihood of seeing new content in the future. If this likelihood dips below a given threshold, the current model is deemed to be stable and a new micro-model is started.

A.1.1 Processing a Packet

```
1  /**
2  * This function processes a packet from the stream to extract
3  * the information needed in the calibration of the time granularity
4  * for each micro-dataset
5  *
6  * @param packet - a StandPacket object that contains all the
7  * header and payload information for TCP or UDP packets
8  * @throws IOException
9  * @throws FileNotFoundException
10 */
11 protected void processPacket(StandPacket packet) {
12   // if this is the first packet processed in the calibration phase
```

```
13      // prevDate=null, so it needs to be set to the current packet
14      // timestamp value
15      if (prevDate == null) {
16        prevDate = packet.getTimestamp();
17      }
18
19      // increment the number of packets
20      numPackets++;
21
22      // show progress in processing packets
23      if (numPackets \ 1000 == 0
24      Utils.logger.log(Level.INFO, numPackets
25                       + " have been processed so far.");
26      // extract the current date as the date of this packet
27      currDate = packet.getTimestamp();
28
29      // check if this is the first packet in a micro-model and set the
30      // model start timestamp
31      if (firstPacket) {
32        firstPacket = false;
33        modelStartDate = currDate;
34      }
35
36      // check if we reached the delta difference (600s) between
37      // timestamps and compute a new likelihood value; we have a
38      // buffer of Utils.lLimit (10) likelihood values on which we
39      // compute the stability
40      if (currDate.getTime() - prevDate.getTime() >= Utils.delta) {
41
42        // compute the number of common grams
43        int commonGrams = 0;
44        CUBloomFilter inter = deltaGrams.and(allGrams, "inter", null);
45        if (inter != null)
46            commonGrams = inter.getUsedSize();
```

```
47
48        // compute the likelihood as the number of new grams over the
49        // total number of grams
50        double lhood = (double) (deltaGrams.getUsedSize() - commonGrams)
51                / (allGrams.getUsedSize() + deltaGrams.getUsedSize());
52
53        // construct the array of likelihoods
54        addLikelihood(lhood);
55
56        // compute the stability
57        stable = computeStability();
58
59        // add the current delta grams to all the grams and clear
60        // deltaGrams
61        allGrams = allGrams.or(deltaGrams, "all_new", null);
62        deltaGrams.reset();
63        prevDate = null;
64
65      } else {
66        // collect all the information for this delta interval
67        byte[] packetData = packet.getData();
68
69        // add the new grams in the last delta interval
70        deltaGrams.insertNGrams(packetData, new int[] { Utils.gramSize });
71      }
72    }
```

A.1.2 Model Stability

```
1   /**
2   * This function implements a linear least square approximation to
3   * check when the content stabilizes to decide the value of time
4   * granularity. It keeps Utils.lLimit (10 by default) points to
```

```
5   * check the stability.
6   *
7   * @return true if it is stable and false if not
8   */
9   private boolean computeStability() {
10    if (lList.size() < Utils.lLimit)
11      return false;
12
13    double sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0,
14    avg = 0, std = 0, a = 0, b = 0;
15
16    for (int i = 0; i < lList.size(); i++) {
17      sumX += i + 1;
18      sumX2 += Math.pow(i + 1, 2);
19      sumY += lList.get(i);
20      sumXY += lList.get(i) * (i + 1);
21    }
22    if (lList.size() == Utils.lLimit) {
23      avg = sumY / Utils.lLimit;
24      for (int i = 0; i < lList.size(); i++)
25        std += (avg - lList.get(i)) * (avg - lList.get(i));
26
27      std /= Utils.lLimit;
28      std = Math.sqrt(std);
29    }
30    // we fit a line y=a+b*x
31    a = (sumY * sumX2 - sumX * sumXY)
32        / (Utils.lLimit * sumX2 - sumX * sumX);
33    b = (Utils.lLimit * sumXY - sumX * sumY)
34        / (Utils.lLimit * sumX2 - sumX * sumX);
35
36    // when the regression coefficient b approaches 0 we consider that
37    // data is stable as long as the std value is also small
38    if (b < 0.01 && b > -0.01 && std < 0.1) {
```

```
39        Utils.logger.log(Level.INFO, "a = " + a + " b = " + b);
40
41        // compute the granularity when the model is stable
42        granularity = (currDate.getTime() - modelStartDate.getTime()) /
              1000;
43        firstPacket = true;
44        return true;
45      }
46      return false;
47    }
```

A.2 Self-Sanitization

Here we present the main functions used in the sanitization process of chapter 3. We note that our tool was also integrated with the Worminator [112] infrastructure in order to exchange the abnormal models in the cross-sanitization process.

A.2.1 Process a Packet

This function processes a network packet and test it using the micro-models. Based on the decisions of the micro-models, a voting score is computed. The first 20 percent of data is used for calibrating the voting threshold, while the rest is used for building the sanitized model.

```
1   /**
2   * This function processes a packet from the stream to extract
3   * the information needed in the sanitization process; the voting
4   * threshold is determined automatically.
5   * We implement the weighted voting strategy
6   *
7   * @param packet - a StandPacket object that contains all the
8   * header and payload information for TCP or UDP packets
```

```
9   * @param numMicroDatasets - number of micro-models that are used
10  * in the voting process
11  */
12  public void processPacket(StandPacket packet, int numMicroDatasets) {
13    // computing the abnormal votes
14    double abnVote = 0;
15    // total number of packets
16    double totalNum = 0;
17
18    // a model that is shared in the Worminator infrastructure
19    Model worminatorAbnModel = null;
20    // if the model is available the cross-sanitization is applied
21    boolean modelAvailable = false;
22
23    // if the cross-sanitization is set in the configuration
24    if(Utils.worminatorAbnModel != null){
25      worminatorAbnModel = Utils.worminatorAbnModel;
26      modelAvailable = true;
27    }
28
29    // test the packet against all the micro-models
30    for (int i = 0; i < microModels.length; i++) {
31      boolean result;
32
33      // there are two strategies for storing the testing results
34      // if the results of the testing against micro-models are stored
35      // in the model
36      if (Utils.storeResultsOnModels){
37        if (microModels[i].getResult(packet.packetID) == 0)
38          result = false;
39
40        else if (microModels[i].getResult(packet.packetID) == 1)
41          result = true;
42
```

```
43          else {
44            result = microModels[i].testData(packet.getData());
45            microModels[i].setResult(packet.packetID, result);
46            totalTests++;
47          }
48        } else {
49          // if the results are stored in the saved traffic
50          if (Utils.storeResultsOnPackets) {
51            if (packet.result[microModels[i].modelID] == 0) {
52              result = microModels[i].testData(packet.getData());
53              totalTests++;
54              if (result){
55                packet.result[microModels[i].modelID] = 1;
56              }
57              else {
58                packet.result[microModels[i].modelID] = 2;
59              }
60            }
61            else {
62              result = (packet.result[microModels[i].modelID] == 1);
63            }
64          }
65          else {
66            result = microModels[i].testData(packet.getData());
67            totalTests++;
68          }
69        }
70
71        if (!result) {
72          abnVote += microModels[i].getPacketNum();
73        }
74
75        totalNum += microModels[i].getPacketNum();
76      }
```

```
77
78
79      // make the vote weighted
80      abnVote /= totalNum;
81
82      // the first 20 perc. of packets are used for calibration
83      if (numMicroDatasets <= microModels.length * 0.2) {
84        // setting the voting threshold automatically
85        for (int i = 0; i < thresholds.length; i++) {
86          if (abnVote <= thresholds[i]) {
87            numPacketsTh[i]++;
88          }
89        }
90      }
91      // for the next part we actually build the sanitized
92      // and the abnormal models
93      else {
94        if (!setTh) {
95          setTh = true;
96          th = getVotingThreshold();
97        }
98
99        if ((abnVote <= th) || Utils.bypassTimeGranularity) {
100         if(modelAvailable){
101           // applying the cross-sanitization
102           System.out.println("Testing with worminator abnormal model");
103           if(!worminatorAbnModel.testData(packet.getData())){
104             sanModel.addData(packet.getData());
105           }
106         } else{
107           // adding the data to the sanitized model
108           sanModel.addData(packet.getData());
109         }
110       } else {
```

```
111         // adding the data to the abnormal model
112         abnModel.addData(packet.getData());
113       }
114     }
115   }
```

A.2.2 Voting Threshold

```
1   /**
2   * This method generated the voting threshold automatically.
3   * It normalizes the number of normal packets generated for
4   * each threshold. It returns max(y-x) as the interest point.
5   *
6   * @return the voting threshold
7   */
8   public double getVotingThreshold() {
9     double norm = 0;
10    double max = 0;
11    int maxPos = 0;
12    int size = numPacketsTh.length - 1;
13
14    // normalize the numPacketsTh
15    norm = numPacketsTh[size - 1] - numPacketsTh[0];
16
17    // compute the maximum (y[k]-y[0])/norm-th[k]
18    for (int k = 0; k < thresholds.length; k++) {
19      System.out.println("Num packets: "+numPacketsTh[k]);
20      double aux = (double) ((numPacketsTh[k] - numPacketsTh[0]) / norm)
21                  - thresholds[k];
22      if (aux > max) {
23        max = aux;
24        maxPos = k;
25      }
```

```
26      }
27      Utils.logger.log(Level.INFO, "THE VOTING THRESHOLD IS: "
28                      + thresholds[maxPos]);
29      return thresholds[maxPos];
30    }
```

Bibliography

[1] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005.

[2] Apache mod_rewrite Buffer Overflow Vulnerability. `http://www.securityfocus.com/archive/1/archive/1/441487/100/0/threaded`.

[3] Daniel Barbara, Julia Couto, Sushil Jajodia, Leonard Popyack, and Ningning Wu. Adam: Detecting intrusions by data mining. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, pages 11–16, 2001.

[4] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J.D. Tygar. Can Machine Learning Be Secure? In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (CCS)*, 2006.

[5] Zafer Barutcuoglu and Ethem Alpaydin. A comparison of model aggregation methods for regression. In *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*. Springer, 2003.

[6] Burton H. Bloom. Space/time trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Damiano Bolzoni, Sandro Etalle, and Pieter Hartel. Panacea: Automating Attack Classification for Anomaly-based Network Intrusion Detection Systems.

In *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2009.

[8] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[9] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Static Analysis on x86 Executables for Preventing Automatic Mimicry Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 213–230, 2007.

[10] Philip K. Chan and Salvatore J. Stolfo. Experiments in Multistrategy Learning by Meta-Learning. In *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 314–323, Washington, DC, 1993.

[11] Suresh N. Chari and Pau-Chen Cheng. BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In *Proceedings of the* 9^{th} *Symposium on Network and Distributed Systems Security (NDSS)*, 2002.

[12] Jedidah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2005.

[13] Gabriela Cretu, Janak J. Parekh, Ke Wang, and Salvatore J. Stolfo. Intrusion and Anomaly Detection Model Exchange for Mobile Ad-Hoc Networks. In *Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC)*, Las Vegas, NV, 2006.

[14] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, and Salvatore J. Stolfo. Extended Abstract: Online Training and Sanitization of AD Systems. In *Proceedings of the NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security*, 2007.

[15] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Salvatore J. Stolfo, and Angelos D. Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[16] Gabriela F. Cretu, Angelos Stavrou, Salvatore J. Stolfo, and Angelos D. Keromytis. Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, 2007.

[17] Gabriela F. Cretu-Ciocarlie, Angelos Stavrou, , Michael E. Locasto, and Salvatore J. Stolfo. Adaptive Anomaly Detection via Self-Calibration and Dynamic Updating. In *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2009.

[18] CVS Heap Overflow Vulnerability. `http://www.us-cert.gov/cas/techalerts/TA04-147A.html`.

[19] David Dagon, Cliff Zou, and Wenke Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 13thNetwork and Distributed System Security Symposium (NDSS)*, 2006.

[20] Sanjoy Dasgupta, Daniel Hsu, and Claire Monteleoni. A general agnostic active learning algorithm. In *Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems (NIPS)*, 2007.

[21] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, 11(61-9), 2003.

[22] Thomas G. Dietterich. Ensemble Methods in Machine Learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[23] Pedro Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 155–164, 1999.

[24] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[25] Prahlad Fogla and Wenke Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.

[26] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[27] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the European Conference on Computational Learning Theory*, pages 23–37, 1995.

[28] Vanessa Frias-Martinez. *Behavior-Based Admission and Access Control for Network Security*. PhD thesis, Columbia University, 2008.

[29] Vanessa Frias-Martinez, Salvatore J. Stolfo, and Angelos D. Keromytis. Behavior-based network access control: A proof-of-concept. In *Proceedings of the Information Security Conference (ISC)*, pages 175–190, 2008.

[30] Vanessa Frias-Martinez, Salvatore J. Stolfo, and AngelosD. Keromytis. Behavior-profile clustering for false alert reduction in anomaly detection sensors. In *Proceedings of the Computer Security Applications Conference (ACSAC)*, pages 367–376, Dec. 2008.

[31] J. Gama, P. Medas, G. Castillo, and P. P. Rodrigues. Learning with drift detection. In *Proceedings of the XVII Brazilian Symposium on Artificial Intelligence*, 2004.

[32] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[33] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral Distance for Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–81, September 2005.

[34] Carrie Gates and Carol Taylor. Challenging the anomaly detection paradigm: a provocative discussion. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, 2006.

[35] ghttpd Log() Function Buffer Overflow Vulnerability. `http://www.securityfocus.com/bid/5960`.

[36] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[37] Kelly Jackson Higgins. AV Vendor Adopts 'Herd' Intelligence, 2007. `http://www.darkreading.com/document.asp?doc_id=140292`.

[38] James Hoagland. SPADE, Silicon Defense, 2000. `http://www.silicondefense.com/software/spice`.

[39] S. A. Hofmeyr, Anil Somayaji, and S. Forrest. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[40] Javits and Valdes. The NIDES statistical component: Description and justification. 1993.

[41] Udo Kelter, Jrgen Wehren, and Jrg Niere. A generic difference algorithm for uml models. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[42] M. Kinateder and K. Rothermel. Architecture and Algorithms for a Distributed Reputation System. In *Proceedings of the First International Conference on Trust Management*, 2003.

[43] R. Klinkenberg and T Joachims. Detecting concept drift with support vector machines. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, 2000.

[44] R. Klinkenberg and S Ruping. Concept drift and the importance of examples. In *Franke, J., Nakhaeizadeh, G., Renz, I., eds.: Text Mining Theoretical Aspects and Applications*, 2003.

[45] Ralf Klinkenberg. Meta-learning, model selection, and example selection in machine learning domains with concept drift, 2005.

[46] Known Vulnerabilities in Mozilla Products. `http://www.mozilla.org/projects/security/known-vulnerabilities`.

[47] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[48] Christopher Kruegel, Thomas Toth, and Clemens Kerer. Decentralized event correlation for intrusion detection. In *Proceedings of the International Conference on Information Security and Cryptology (ICISC)*, pages 114–131, 2001.

[49] Christopher Kruegel, Thomas Toth, and Engin Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the Symposium on Applied Computing (SAC)*, Madrid, Spain, 2002.

[50] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, Washington, D.C., 2003.

[51] Christopher Kruegel, Giovanni Vigna, and William Robertson. A multi-model approach to the detection of web-based attacks. *International Journal of Computer and Telecommunications Networking*, 48(5):717–738, 2005.

[52] Terran Lane and Carla E. Broadley. Approaches to online learning and concept drift for user identification in computer security. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, 1998.

[53] Peng Li, Debin Gao, and Michael K. Reiter. Automatically Adapting a Trained Anomaly Detector to Software Patches. In *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2009.

[54] Richard P. Lippmann and Joshua Haines. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, pages 162–182, 2000.

[55] Local DoS Attack in Linux Kernel. `http://www.sfu.ca/~siegert/linux-security/msg00047.html`.

[56] Michael E. Locasto, Gabriela F. Cretu, Shlomo Hershkop, and Angelos Stavrou. Post-Patch Retraining for Host-Based Anomaly Detection. In *Columbia University Computer Science Department Technical Report, CUCS 035-07*, 2007.

[57] Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis, and Salvatore J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *Proceedings of the IEEE Information Assurance Workshop*, West Point, NY, 2005.

[58] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis, and Salvatore J. Stolfo. Return value predictability profiles for self—healing. In *Proceedings of the 3rd International Workshop on Security (IWSec)*, pages 152–166, Berlin, Heidelberg, 2008. Springer-Verlag.

[59] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[60] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection (RAID)*, 2009.

[61] Matthew Mahoney, , Matthew V. Mahoney, and Philip K. Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. Technical report, 2001.

[62] Matthew V. Mahoney. Network traffic anomaly detection based on packet bytes. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 346–350, New York, NY, USA, 2003. ACM.

[63] Rohan McGovern. Inotify-tools. `http://inotify-tools.sourceforge.net/`.

[64] John McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by

Lincoln Laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–291, 2000.

[65] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[66] C. Monteleoni and M. Kaariainen. Practical online active learning for classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, June 2007.

[67] David Moore and Colleen Shannon. The Spread of the Code Red Worm (CRv2). http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml.

[68] Henry Muccini. Using model differencing for architecture-level regression testing. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 59–66, Washington, DC, USA, 2007. IEEE Computer Society.

[69] Multiple Vulnerabilities in libpng. `http://www.us-cert.gov/cas/techalerts/TA04-217A.html`.

[70] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[71] MySQL 5.0 Reference Manual: Using Triggers. `http://dev.mysql.com/doc/refman/5.0/en/triggers.html`.

[72] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Security and Privacy*, Oakland, CA, 2005.

[73] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 245–254, New York, NY, USA, 2002. ACM.

[74] Null httpd Remote Heap Overflow Vulnerability. `http://www.securityfocus.com/bid/5774`.

[75] Janak J. Parekh. *Privacy-Preserving Distributed Event Corroboration*. PhD thesis, Columbia University, 2007.

[76] Bambang Parmanto, Munro Paul W., and Howard R. Doyle. Improving Committee Diagnosis with Resampling Techniques. *Advances in Neural Information Processing Systems*, 8:882–888, 1996.

[77] Emanuel Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[78] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1998.

[79] Tadeusz Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[80] Phillip A. Porras and Peter G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Systems Security Conference*, oct 1997.

[81] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 207–225, August 2003.

[82] Remote Code Injection Vulnerability in fetchmail. `http://fetchmail.berlios.de/fetchmail-SA-2005-01.txt`.

[83] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 335.2, Washington, DC, USA, 2003. IEEE Computer Society.

[84] Jamie Riden, Ryan McGeehan, Brian Engert, and Michael Mueter. Know your Enemy: Web Application Threats, 2008. `http://www.honeynet.org/book/export/html/1`.

[85] Konrad Rieck and Pavel Laskov. Language models for detection of unknown attacks in network traffic. *Journal in Computer Virology*, 2(4):243–256, 2007.

[86] Haakon Ringberg, Augustin Soule, Jennifer Rexford, and Christophe Diot. Sensitivity of pca for traffic anomaly detection. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 109–120, New York, NY, USA, 2007. ACM.

[87] Dan Roth and Kevin Small. Margin-based active learning for structured output spaces. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, 2006.

[88] Samba Security Releases. `http://samba.org/samba/samba/history/security.html`.

[89] Vitaly Shmatikov and Ming-Hsiu Wang. Security against probe-response attacks in collaborative intrusion detection. In *Proceedings of the 2007 Workshop on Large Scale Attack Defense (LSAD)*, pages 129–136, New York, NY, USA, 2007. ACM.

[90] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Proceeding of the 14th international conference on Architectural*

support for programming languages and operating systems (ASPLOS), pages 37–48, New York, NY, USA, 2009. ACM.

[91] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Technical Conference*, June 2005.

[92] Anil Somayaji and Stephanie Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[93] Yigbo Song, Michael E. Locasto, Angelos Stavrou, A. D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of Modeling Polymorphic Shellcode for Signature Detection. In *Columbia University Computer Science Department Technical Report, CUCS 007-07*, 2007.

[94] Yingbo Song, Angelos D. Keromytis, and Salvatore J. Stolfo. Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[95] Stuart Staniford-Chen, Steven Cheung, R. Crawford, and M. Dilger. GrIDS - A Graph Based Intrusion Detection System for Large Networks. In *Proceedings of the National Information Computer Security Conference*, Baltimore, MD, 1996.

[96] Angelos Stavrou, Gabriela Cretu-Ciocarlie, Michael Locasto, and Salvatore Stolfo. Keep Your Friends Close: The Necessity for Updating an Anomaly Sensor with Legitimate Environment Changes. *To appear in Proceedings of the 2nd Workshop on Security and Artificial Intelligence (AISec)*, 2009.

[97] Salvatore Stolfo, Wei Fan, Wenke Lee, Andreas Prodromidis, and Phil Chan. Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.

[98] Salvatore J. Stolfo, Frank Apap, Eleazar Eskin, Katherine Heller, Shlomo Hershkop, Andrew Honig, and Krysta Svore. A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection. *Journal of Computer Security*, 13(4), 2005.

[99] Salvatore J. Stolfo, Shlomo Hershkop, Chia-Wei Hu, Wei-Jen Li, Olivier Nimeskern, and Ke Wang. Behavior-based modeling and its application to email analysis. *ACM Transactions on Internet Technology (TOIT)*, 6(2):187–221, 2006.

[100] STunnel Client Negotiation Protocol Format String Vulnerability. `http://www.securityfocus.com/bid/3748`.

[101] Kymie M.C. Tan and Roy A. Maxion. Why 6? Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 188–201, May 2002.

[102] Kai Ming Ting and Ian H. Witten. Stacked generalizations: When does it work? In *Proceedings International Joint Conference on Artificial Intelligence*, pages 866–873, 1997.

[103] Johannes Ullrich. DShield home page, 2005. `http://www.dshield.org`.

[104] Giovanni Vigna and Richard A. Kemmerer. Netstat: A network-based intrusion detection approach. *Journal of Computer Security*, 7:37–71, 1998.

[105] W3C. HTTP Status Codes, 1992. `http://http://www.w3.org/Protocols/HTTP/HTRESP.html`.

[106] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Security and Privacy*, Oakland, CA, 2001.

[107] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2002.

[108] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, Seattle, WA, 2005.

[109] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[110] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[111] Daniel Wolpert. Stacked Generalization. *Neural Networks*, 5:241–259, 1992.

[112] Worminator. `http://worminator.cs.columbia.edu/public/index.jsp`.

[113] Dit-Yan Yeung and Calvin Chow. Parzen-window network intrusion detectors. In *Proceedings of the Sixteenth International Conference on Pattern Recognition*, pages 385–388, 2002.