Keep Your Friends Close: The Necessity for Updating an Anomaly Sensor with Legitimate Environment Changes

Angelos Stavrou
George Mason University
Fairfax, VA, USA
astavrou@gmu.edu

Gabriela F. Cretu-Ciocarlie
Columbia University
New York, NY, USA
gcretu@cs.columbia.edu

Michael E. Locasto
George Mason University
Fairfax, VA, USA
mlocasto@gmu.edu

Salvatore J. Stolfo
Columbia University
New York, NY, USA
sal@cs.columbia.edu

ABSTRACT

Large-scale distributed systems have dense, complex code-bases that are assumed to perform multiple and inter-dependent tasks while user interaction is present. The way users interact with systems can differ and evolve over time, as can the systems themselves. Consequently, anomaly detection (AD) sensors must be able to cope with updates to their operating environment. Otherwise, the sensor may incorrectly classify new patterns as malicious (a false positive) or assert that old or outdated patterns are normal (a false negative). This problem of "model drift" is an almost universally acknowledged hazard for anomaly sensors. However, relatively little work has been done to understand the process of identifying and seamlessly updating an operational network AD sensor with legal modifications like changes to a file system or back-end database.

In this paper, we highlight some of the challenges of keeping an anomaly sensor updated, an important step toward helping anomaly sensors become a *practical* intrusion detection tool for real-world network and host environments. Our goal is to eliminate needless false positives arising from the gradual de-synchronization of the sensor from the environment it is monitoring. To that end, we investigate the feasibility of automatically deriving and applying a "data" or "model patch" that describes the changes necessary to update a "reasonable" AD behavioral model (*i.e.,* a model whose structure follows the core design principles of existing anomaly models). We propose an update procedure that is *holistic* in nature: specifically, we present preliminary results on how to update a sensor that monitors the request and response messages for non-dynamic HTTP requests and software patches. In addition, we propose extensions for dynamic, database-driven requests and responses.

AISec'09, November 9, 2009, Chicago, Illinois, USA.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS—*Security and Protection*

General Terms

Security, Experimentation

Keywords

anomaly detection, model update, concept drift

1. INTRODUCTION

Although the use of anomaly detection as part of an intrusion detection infrastructure seems to provide benefits in terms of detecting previously unknown exploits and attack patterns, anomaly sensors, as currently practiced, are often seen as too much of a liability in terms of their false positive production. Further complicating this problem, real software systems reside in dynamic, fluid environments, and an overly sensitive anomaly detector will have problems refraining from issuing spurious alerts in response to otherwise normal changes in user behavior or underlying operating data.

Indeed, anomaly detection models can produce a great number of false positives as they drift or desynchronize from legitimate updates to their operating environment. Figure 1 presents an instance of this phenomenon, where the number of false alerts surges significantly because of HTTP requests for content that was not encountered previously by the sensor (the anomaly detection sensor used here is Anagram [22]). This figure is part of the motivation for our previous work [3], where we proposed a gradual, online learning component along with a training data sanitization process in order to cope with legitimate changes that were caused by external factors (in this case the users' behavior). A system component that would control the AD system reaction to such a shift in users' behavior would have avoided the spurious alerts.

In this paper, we study model update techniques that are performed in response to controlled, legitimate changes of the operating environment of sensors. Behavioral, data, and software modifications (*i.e. patches*) are *legitimate* but, at the same time, disruptive activities with the potential to render the AD model outdated very quickly. For example, if a host–based AD system monitors an application, the AD must update (*i.e., patch*) its behavioral model in response to changes in the underlying data, usage, or code modifi-

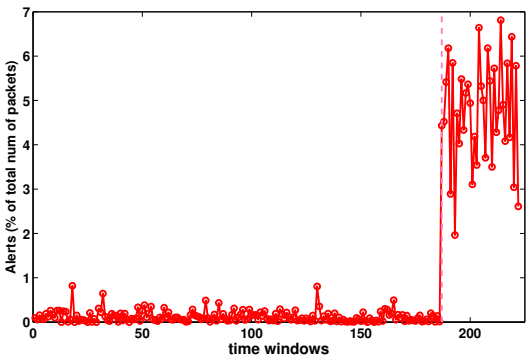

Figure 1: Alert rate using Anagram. Vertical line marks the boundary between old and new traffic. In this case, the surge in false alerts is triggered by a shift in users' requests for new media.

cations [9]. *Legitimate* alterations of the back-end file systems or databases can also alter the normality model of a system leading to the de-synchronization of the AD system and a constant flood of false alerts. We posit that we can implement a feedback mechanism that informs the anomaly detection sensor of such changes and provides the required information for updating the AD model without re-training which is both costly and disruptive.

To that end, we highlight the general issue of updating the AD models in response to changes enacted by a reactive patching mechanism, file system, and back-end database alterations. Our key insight is that certain data patterns are derived from sanctioned changes to the data in the sensor's environment as opposed to malicious requests. Thus, we propose a framework that informs an anomaly sensor which changes result from legitimate environment alterations and adjust its "normal" model by including such changes. This framework can be applied to AD sensors placed either on the network or on the host as opposed to previous research [11] that focuses purely on web traffic. Of course, there is a requirement for specialization of the training phase for each sensor to accept updates to its model. Therefore, being able to adapt in a dynamic environment depends heavily on our ability to:

- Identify and verify legitimate environment changes that pertain to the operation of the AD sensor.
- Update the normality model of the AD sensor in an efficient manner.
- Prevent the degradation of the AD model due to multiple or continuous changes.

We argue for the feasibility of a procedure that amends an AD behavioral model in response to legitimate changes. In addition, we examine different update strategies and their potential impact on the overall system performance.

2. AD MODEL UPDATING

To the best of our knowledge, the two main current approaches to the problem of AD model retraining include: (1) fully retraining the AD sensor, and (2) incorporating a mechanism for gradual, online retraining into the AD algorithm itself. The first choice represents a significant increase in the length of the recovery process, and the second one needs to incorporate techniques that cope with training attacks. Instead, our goal is to harness the fact that patches, especially security–related ones, cause small, localized changes in the underlying AD model and that the file system and database feedback system can take advantage of a multi-granular modeling technique that easily identifies the sub-models that need to be updated. Therefore, if we can provide an automated mechanism that efficiently incorporates these changes into the existing model, we can avoid a lengthy and attack–prone retraining phase.

2.1 Complete Retraining

The first approach seems unsatisfactory because retraining the model may take significant amounts of time, and it represents an additional burden on system administrators. Long training phases (usually necessary to capture the breadth of a complex behavior) impose additional delay on the patch deployment process (sometimes hours or days of down-time). 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.

2.2 Gradual Retraining

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 [8]. 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 our previous work [2], we proposed a novel online learning technique that incorporates training data sanitization and automation methods, maintaining the performance level of the AD sensors over a long time horizon. In this paper, we introduce a specialized approach to the problem of AD model update, which alters 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.

2.3 Spot Retraining

Our approach for model update is 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 system 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.

We analyze the case of patches separated from the file systems and databases, 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.

3. POST-PATCH MODEL UPDATE

Our aim is to analyze the feasibility of building a mechanism that provides enough information to update a known 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, then it seems possible to construct an AD model update procedure that inflicts small changes in the model.

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:

$$\{f_i(args, rval), \ldots\} \rightarrow \{f_k(args, rval)\} \quad (1)$$

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[1]. The simplest case is based on monitoring only sequences of system 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.*[19].

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.

[1]Although we restrict our examination to a host–based model, examining the impact that patches have on n-gram based network content models is an interesting area for future work.

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.

3.1 Feasibility Study

Our evaluation contains two directions: first, we examine known anomaly and specification–based sensors to discover their supervised or un-supervised training time. We do so to confirm our hypothesis that such systems have relatively long training periods that frequently 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 [16] 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 un-supervised. An un-supervised 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 [5, 16] 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 1 lists our results for a variety of applications, including stunnel [20], some web servers [1, 6, 15], linux [10], cvs [4] and fetchmail [17], as well as various vulnerabilities in libpng [13], Firefox [7], and Samba [18]. The Δs are computed by counting the number of control and data flow changes as defined in tables 2 and 3.

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

Table 1: *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

Table 2: 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

changes from the text of a patch itself[2]. 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.

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.

[2]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.

Table 3: Data flow changes introduced by patches

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

4. FS/DB AD MODEL UPDATE

In this section, we explore web server applications that exhibit *concept drift* and we attempt to update them harnessing our knowledge about legitimate changes of the state of the file system and database back-ends. Although these techniques attempt to address the content and behavioral changes in web servers, they can also be applied to other applications including Voice over IP (VoIP) services.

A web server presents to the user content that comes either statically from the file system or dynamically generated by interaction with with back-end database using SQL (see figure 2). 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 changes in the back-end data in a manner that is consistent with the data contained in the back-end database. That includes identical or similar requests that refer to the same data or database entries. The differentiating factor between a malicious and a legitimate request is that the former will try to either create a different back-end request or server modified data to the user.

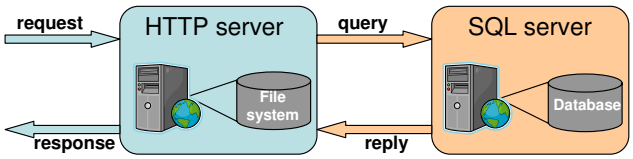

Figure 2: Front-end and back-end correlation for web server model update

Our solution employs different training and self–updating methods for statically 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.

In 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.

4.1 Feasibility Study

In our proof-of-concept prototype, we implemented a simple but effective notification system. To communicate to the AD sensor any legitimate file system alterations, we used inotify-tools [12], a library and set of command-line programs for Linux. This library provides a simple interface to *inotify* (a Linux kernel subsystem for file system event notification). Table 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. From the output of inotifywait 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 recursively monitored the directory where the web site files reside.

Table 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
```

As a database notification system, we used the MySQL event 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 [14]. 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 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 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,

Table 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 ;
```

we considered a real HTTP server (*www1*), 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 3 presents the training cost in the case of two network-based anomaly detection systems: Anagram (both stand alone and coupled with the training dataset sanitization method) and Payl [21].

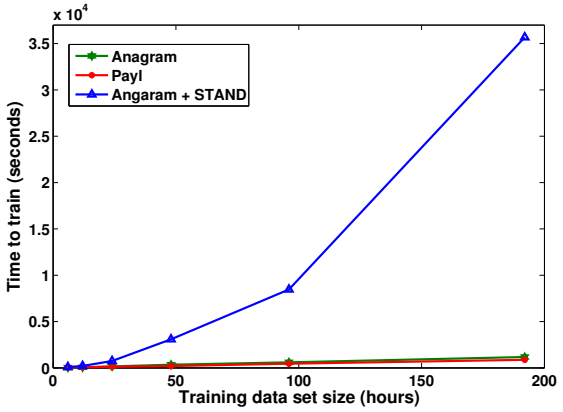

Figure 3: *Time to train an AD sensor for different training data set size.* For the case of Anagram+sanitization, we present the initial effort of building the first batch of micro-models and the sanitized model

For the same web server, *www1*, for which we created the Payl and Anagram models, we considered our multi-granular model approach. Figure 4 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 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

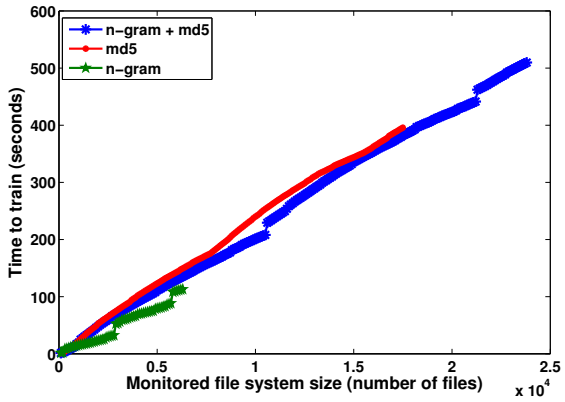

Figure 4: ***Time to train our multi-granular model.*** **We have n-gram models for the html and htm files and md5 models for the rest of the static files.**

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).

Table 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

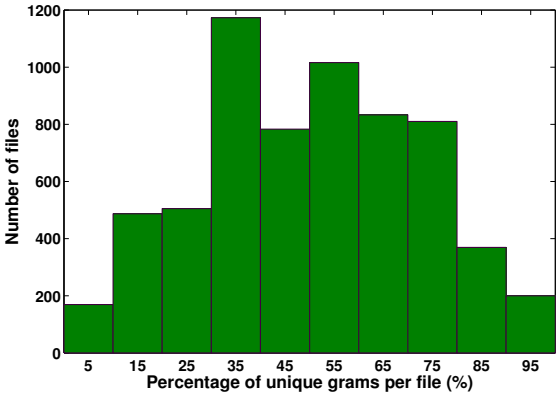

Figure 5: Histogram of percentage of unique grams out of the total number of grams in the files

In order to analyze the performance 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 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. CONCLUSION AND FUTURE WORK

As the system under protection evolves over time, its behavior becomes increasingly inconsistent with the original normality model, eventually rendering the AD sensor unusable. To amend this, we propose an approach that monitors internal components of the protected system for changes that can be verified as legitimate. We analyze the case of web services with back-end file system and database and, the case of software patching. To that end, we study the feasibility of automatically deriving and applying a "model patch" based on information related to alterations in the file system and database. We present baseline results on how to update a sensor that monitors the request and response messages for non-dynamic HTTP requests. Our preliminary results indicate 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. Furthermore, we show that we can extend the static model for dynamic, database–driven requests and responses. We also examine model changes that stem from limited scope patching of software systems that are monitored by AD systems. 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. As future work, we intend to implement this model update procedure for different host–based AD sensors.

6. REFERENCES

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

[2] G. F. Cretu, A. Stavrou, M. E. Locasto, A. D. Keromytis, and S. J. Stolfo. Casting Out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[3] G. F. Cretu-Ciocarlie, A. Stavrou, , M. E. Locasto, and S. J. Stolfo. Adaptive Anomaly Detection via Self-Calibration and Dynamic Updating. In *Proceeding of the 12th International Symposium On Recent Advances In Intrusion Detection, RAID*, 2009.

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

[5] D. Gao, M. K. Reiter, and D. 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.

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

[7] Known Vulnerabilities in Mozilla Products.

http://www.mozilla.org/projects/security/known-vulnerabilities.

[8] T. Lane and C. 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.

[9] P. Li, D. Gao, and M. Reiter. Automatically Adapting a Trained Anomaly Detector to Software Patches. In *Proceeding of the 12th International Symposium On Recent Advances In Intrusion Detection, RAID*, 2009.

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

[11] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceeding of the 12th International Symposium On Recent Advances In Intrusion Detection, RAID*, 2009.

[12] R. McGovern. Inotify-tools. http://inotify-tools.sourceforge.net/.

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

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

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

[16] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 207–225, August 2003.

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

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

[19] S. J. Stolfo, F. Apap, E. Eskin, K. Heller, S. Hershkop, A. Honig, and K. Svore. A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection. *Journal of Computer Security*, 13(4), 2005.

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

[21] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 227–246, September 2005.

[22] K. Wang, J. J. Parekh, and S. 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.