UCLog: A Unified, Correlated Logging Architecture for Intrusion Detection

Zhenmin Li[†], Jed Taylor[†], Elizabeth Partridge[†‡],
Yuanyuan Zhou[†], William Yurcik[‡],
Cristina Abad[†‡], James J. Barlow[‡] and Jeff Rosendale[‡]

[†]Department of Computer Science,
University of Illinois at Urbana-Champaign
{zli4, jtaylr2, epatrid, yyzhou}@uiuc.edu
[‡]National Center for Supercomputing Applications (NCSA)
{byurcik, cabad, jbarlow, jeffr}@ncsa.uiuc.edu

Abstract

Activity logs can be used for intrusion detection; however, most previous work on intrusion detection examines only activity logs from a single component. Doing so fails to take advantage of the naturally existing correlations among activities in different types of logs, such as network logs and system call logs.

This paper explores correlation for intrusion detection. Specifically, we propose UCLog*, a unified logging architecture that can effectively capture correlations among entries in different types of logs. UCLog enables the intrusion detection system to make some sense of the myriad of different available logs and correlate the information the logs present to enhance the intrusion detection process.*

We have evaluated UCLog by using it to detect the infection of a host with the Yaha virus. Our results show significant improvement when the information available in several logs is correlated.

Key words: *intrusion detection, auditing, logging, file I/O monitoring*

1 Introduction

Computer systems are vulnerable targets for terrorists and criminals to break-in. Intrusions can compromise the integrity, confidentiality,

or availability of the systems and the information they hold [8]. For example, in January 2004 when the SQL Slammer Worm in less than seven hours the Internet traffic was overloaded to the point that many private networks were jammed and web sites stopped responding. The cost of the shut downs and cleaning up the infection was estimated to be around $1 billion [25].

An intrusion detection system (IDS) attempts to detect attacks by monitoring system or network behavior. While many existing IDSs require manual definitions of normal and abnormal behavior (intrusion signatures) [37], recent work has shown that it is possible to identify abnormalities automatically using machine learning or data mining techniques [15, 5, 7, 18, 23] to generate models or rules, which the IDS can use to detect intrusions.

However, most of the previous work on intrusion detection focuses on activities generated by a single source, resulting in many false positives and undetected intrusions. For example, some studies [23] detect intrusions by monitoring the network traffic such as tcpdump logs, whereas other studies [7, 18] monitor only system call logs. These studies usually have high false positive rates. Correlating activities from all involved components and then analyzing them together would give the IDS a better *picture* of the system, increasing the detection accuracy and reducing the false positives.

Unfortunately, few studies have been conducted on fusing audit log data from heterogeneous sources into a single framework for assessing suspicious behavior. Some existing systems, such as Windows, provide APIs to allow both device drivers or applications to log events in a centralized view [27]. Similarly, the ReVirt system [16] can record all actions below the level of the virtual machine, allowing arbitrarily detailed information about all user-level behavior at a given time. These systems provide a centralized mechanism to log activities from heterogeneous sources, but activities are not organized in a structured making it hard to capture their correlations.

Another common problem with current IDSs is that it is hard or impossible for the user to specify his/hers critical applications which should be watched for instrusions. The UCLog architecture makes it easy to add application-specific loggers and monitors, which form a middleware layer that enable the IDS to monitor different applications

Sophos Ex. 1005
Sophos Ltd. and Sophos Inc., v. Webroot Inc.
IPR2023-00732
Page 1 of 15

without modifying the code of the applications or the IDS itself.

This paper proposes a unified logging architecture, called UCLog, that effectively explores activity correlations for intrusion detection. UCLog is motivated from and complements previous work on secure logging such as Self-Securing Storage System (S4) [39] and ReVirt [16]. UCLog also complements previous work. More specifically, the UCLog architecture provides several benefits:

- UCLog organizes all audit logs from heterogeneous sources in a very structured way and is closely integrated into the intrusion detection system.
- UCLog effectively exploits activity correlations between different components to improve the accuracy of intrusion detection and to reduce the number of false positives.
- At an intrusion, UCLog can be used to automatically identify the source and the execution path of the intrusion, whereas most current systems require an administrator to manually examine the logs to find such information.
- UCLog can easily be extended to use information from new applications in the intrusion detection process, by adding the corresponding logger and monitor to the system.

We have implemented the UCLog architecture on a real system running Microsoft Outlook Express on Windows 2000. We evaluate this system using many normal and abnormal traces. Abnormal traces are collected when the system is infected by e-mail viruses. Using these traces, we have studied how correlating activities from the system call sequence, network activity, and file I/O operations can improve the intrusion detection accuracy. Our results show that our proposed architecture can not only provide the means to make an intrusion more obvious to an IDS, but can also help reduce the number of false positives. Compared to a baseline system that only monitors a single component, our system improves the accuracy by up to a factor of 2.5.

This paper is organized as follows. Section 2 briefly describes background material. Section 3 discusses the effectiveness of correlation for intrusion detection. Section 4 presents the UCLog architecture. Section 5 and 6 present the evaluation methodology and results. Section 7

summarizes related work. In Sect. 8 we describe what still needs to be done, and in Sect. 9 we conclude.

2 Background

This section describes some background on activity logging and how log files are used in intrusion detection systems (IDSs).

2.1 Activity Logging

Activity logging is a mechanism to collect the information about past activities that the system or administrator can use to characterize access behavior, audit user accounts, replay operations, or detect anomalies and rollback to a previous state [16, 39]. For example, the system can log every user login/logout to monitor any possible break-ins. Research issues about logging include what and how to log, how to manage and protect logs, and how to use the logged data.

Event logging in Microsoft Windows provides a standard, centralized way for applications to record important software and hardware events [27]. By default, there are three types of logs: security, system, and application. Additionally, Windows event logging provides an event log component and a dynamic-link library (DLL) that includes some application programming interfaces (APIs) for logging management such as storing, retrieving, and backing up data. MS Windows event logging organizes these events in a very simple way but does not provide any support to find correlations between events from different components, nor is it designed to easily connect with any IDS.

To avoid the risk of compromise, it is important to develop protection mechanisms for logging. Dunlap et al. propose ReVirt [16], in which the operating system is moved to a virtual machine and logging takes place below the level of the virtual machine. This protects the integrity of the logging system. Furthermore, the ReVirt system can replay the instruction-by-instruction execution of the system in order to test the effects of a particular command. The ReVirt logging architecture is designed for simple monitoring, recovery, and replay. But similar to the MS Windows event logger, it does not support correlations among activities in different components and is not connected with any IDS.

Additionally, the systems examined in this paper would benefit from using a secure logging system such as Rio/Vista [26] or Self Securing Storage System(S4) [39]. Rio/Vista is a cache designed to survive OS crashes, coupled with a recoverable memory library which allows fast logging [26]. S4 isolates logging from user interactions, to reduce performance costs [39].

Our work complements previous work such as ReVirt, MS Windows Event logger, Rio/Vista, and S4 by providing a unified logging architecture that captures correlations and integrates with the IDS to build a system that is highly secure, reliable and self-recoverable.

2.2 Intrusion Detection Systems

Since log files contain information about past activities, the data collected by a logging system can be used by an IDS to detect signs of suspicious behavior.

A generic IDS contains three major components: sensors, analyzers, and user interface [2]. Sensors are responsible for collecting data, such as network packets and log files. The analyzer determines if an intrusion has occurred by analyzing the data from sensors. The user interface enables an administrator to view output from the system and to control the behavior of the IDS.

Analyzers can be classified in two: policy-based and anomaly-based [3]. In a policy-based detection system, known signatures of intrusive or suspicious behavior are defined by the security policy. In contrast, an anomaly-based detector reacts to deviations from previously observed normal behavior (e.g. by using data mining or machine learning techniques [23, 7, 11, 5]). These definitions of normal behavior can be automatically updated. Our system can easily combine both mechanisms. Each log can be processed by any type of analyzer, and the analyzers collaborate with each other to identify suspicious behavior.

Most previous work on IDSs analyze a single component, ignoring the correlations among different components. We first show how correlation can be used to improve the accuracy of intrusion detection. Then, we study how to build a logging architecture that can conveniently capture these correlations and integrate them with the IDS. More details on existing IDSs can be found in Sect. 7.

3 Exploring Correlations for Intrusion Detection

We have conducted several experiments to examine the correlation between logs of different system components. The motivation of our experiments is to justify that the correlation can potentially increase the accuracy of an intrusion detection system. In other words, when the network shows abnormal behavior, is the system call sequence also abnormal? And vice versa?

In our experiments, we have collected several network and system call traces. Some of these traces are collected when the system behaves normally, and the others are collected when the system is infected by e-mail viruses. We use data mining tools to analyze these traces separately and identify abnormal regions for each trace, then we examine the results of every network trace and the results of its corresponding system call trace (collected simultaneously) to check for correlations between the network activity and system calls.

Our experiments consist of three steps: collecting audit data, creating detection models by using data mining techniques, and then applying the models to detect abnormal behavior.

In order to train and test such a system, the first step is to collect multiple traces of normal and abnormal user behavior simultaneously from different subsystems. In our experiments, we collected traces from a Windows 2000 host, but the methods can be similarly applied to any other operating systems. The logging tools we used are described in Sect. 5. We used traces containing the Win32 Yaha virus [31] and normal traces when training our system.

The next step is to train the detection system by applying a data-mining algorithm, RIPPER [12, 13], to the trace data to extract useful behavior patterns. The rules generated by RIPPER are used to classify normal and abnormal system call sequences. The processing detail is described in Sect. 5.

After obtaining the rules from the training data, the next step is prediction and post-processing of test data. Similar to Lee's post-processing approach in [23], each sequence of system calls in the testing trace is predicted as "normal" or "abnormal" according to the rules. The predictions are then divided into regions of size $3l + 1$. If the number of abnormal prediction in a region is greater than a half, we predict

the region as "abnormal". Next, we take an overlapping sliding window of size 500 and pass it over the sequence of predictions. For each system call sequence, we calculate the number of abnormal predictions in this sliding window. Results are graphed in Fig. 1 and give us an idea of the behavior of a given abnormal trace.

Figure 1(a) shows the results of the system call trace collected when the system is infected by the Yaha virus. As shown on the curve, there are some patches of increased abnormal behavior, but it is difficult to determine if any intrusion has really taken place, especially for the first abnormal spike. It is possible that either the user has just issued a command or tried a new feature that he has never tried before, or that a real intrusion occurs, i.e., the user just opened an e-mail attachment that compromises the local address book and fires off infected e-mail to other people in the address book. By examining the system call behavior alone, these types of questions can be difficult to answer because the system calls alone may not look all that abnormal.

When we examine the network traffic corresponding to the same time period, we can see an obvious increase in the network traffic as shown in Fig. 1(b). This means that the abnormal increase in the network traffic confirms the abnormal behavior in the system call sequence, therefore, the IDS can report an intrusion warning with a higher confidence.

A more detailed demonstration of how log correlation can improve the intrusion detection process was presented in [1].

4 UCLog Architecture

In this paper, we propose a unified logging architecture that captures the correlations among activities from different components and is closely integrated into the intrusion detection system. UCLog consists of several components as shown in Fig. 2: loggers, monitors, a centralized activity database, and a background rule generator. The loggers and monitors form a middleware layer that enables the IDS to use information held by different and heterogeneous logs and correlate it to improve its accuracy. Loggers and monitors should be coupled with corresponding system modules such as network, user shell, file systems, and individual applications. Components in the UCLog architecture could communicate with each other using secure messages with

authentication [6, 36].

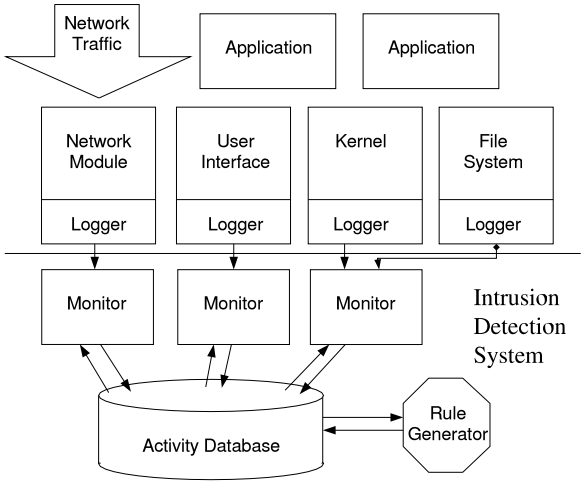
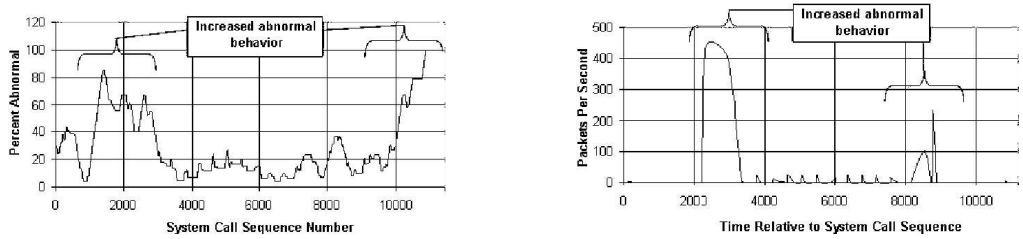


Figure 2. UCLog architecture

Loggers collect information about various aspects of system behavior and send this data to *monitors*. Monitors is watch system behavior and collaborate with each other through the centralized *activity database*. Monitors filter and compress audit data collected by loggers, and compare this data with existing models or signatures. The centralized activity database stores and manages audit data in an organized way. It also supports queries from a monitor to find out correlated activities watched by other monitors, or to trace to the specific processes or users that directly or indirectly cause the occurring intrusion. The *rule generator* produces the models for normal or abnormal behavior and signatures for previous intrusions. The rule generator can run either in the background or off-line, and either on the same or possibly several other machines. In the following subsections, we describe each component in more detail.



(a) System call trace

(b) Network traffic trace

Figure 1. Abnormality in Yaha virus traces

4.1 Loggers

A logger records information about the behavior of a particular part of the environment and transmits that data to the corresponding monitor. For ease of deployment, a middleware layer of loggers with a common programming interface should be implemented. Thus, hiding the details of how to process each log from the other components. Each logging module must be coupled with the subsystem from which the logger is collecting audit data. For example, the network subsystem should have a logger module to collect network activities. Similarly, the user command shell also needs a logger to record user-entered commands. Several particularly useful logging modules for intrusion detection include:

- **Kernel API Logger**: A kernel API logger should record system calls, the parameters of those calls, and the time at which they are made.
- **Network Logger**: A network logger could record all packets passing through the system, all packet headers received by the system, or even record higher, summary-level information (e.g. Cisco NetFlows [10]). The actual implementation would depend on the desired circumstances.
- **File-System Logger**: A file-system logger might detect changes to significant blocks, such as the password file, or detect unusual access patterns. For example, a process which attempts to touch every block once could be harmless, such as a disk defragmentation process, or it could be some user attempting to illegally duplicate the disk.
- **User Behavior Logger**: It would be useful to have some module that records a user's interactions with the system, such as when each user logs in and out.
- **System Events Logger**: A facility that monitors the children, files, and directories created by some process would be useful in many situations.
- **Application-Specific Logger**: Additional application-specific loggers can be added as needed.

Each of the types of loggers listed can collect host-specific or system-wide information. The latter are *distributed loggers*. Distributed loggers are very useful and can be implemented to log system-wide information. For example, it is very common for organizations to have dedicated servers that collect and store logs obtained from different sources (e.g. *syslogs* obtained from each host in the organization). Obtaining a system-wide view is very helpful for intrusion detection, as related

malicious activities may appear in several hosts. Correlating this information can help detect attacks that otherwise may not be obvious.

To reduce the performance overhead, loggers do very little processing but instead stream the data to the corresponding monitor. Some loggers may only be used when fine-granularity logging is required, e.g. an attack is suspected.

4.2 Monitors

Monitors receive audit data from one or more loggers and are responsible for filtering and classifying the stream entries, and detecting any abnormal behavior. Besides loggers, a monitor needs to interact with the centralized activity database to find correlated activities from other sources. Moreover, a monitor also has to consult with the rule generator to classify abnormal behavior. In addition, monitors also communicate with each other to confirm intrusions.

When a monitor receives audit data from loggers, it needs to reduce the data size by pruning unnecessary information such as highly variable parameters, user variable names, and any other unpredictable information. To further pack the audit data, the monitor can compress the packet statistics to connection-level information. A monitor may also combine several log entries from different loggers. For example, if several different loggers generate information from the same event (for example, a file I/O system call is recorded by both the system call logger and the file I/O logger. In this case, the monitor combines those logs to a unified format. The data the monitors store in the activity database must be stored in a unified format, so that the rule learning and processing module can understand any entry. There are several unified log formats that can be used for this purpose, for example, the one used by Snort [37], the one used by Harvester [17], the Common Intrusion Specification Language (CISL) [38], among others. Ideally the unified format should adhere to some standard, and it is believed that an XML format may become a standard [28].

Once audit data is properly filtered, a monitor compares the stream to previously generated models or signatures by using data-mining techniques such as outliers analysis [19] or machine learning techniques. More specifically, a monitor would examine the audit data sequence

and check whether a subsequence matches an intrusion signature or violates a certain model that captures normal behavior. For example, a simple model could be "file activities generated by ordinary users are fewer than 5 I/Os per second during 12am to 6am". Then if the file I/O monitor detects many file accesses from an ordinary user, it can generate a warning that an intrusion is likely taking place.

Classifying information in an isolated way does not take full advantage of the semantic structure supported by the UCLog architecture. It is often useful to correlate the outputs of the monitors to strengthen the classification judgment. For example, a system call sequence may have a fraction of abnormal behavior. Before generating an intrusion alarm, it is better to consult with other components to see if correlated activities are also showing abnormal activities. In addition, when a stream is flagged as unusual, a monitor may need to query the activity database and the rule generator to determine how previous instances of this pattern were classified (if they occurred and were not part of the rule set).

4.3 Activity Database

The activity database stores and manages audit data in a very organized way. It is implemented as an efficient relational database management system in order to present monitors with an interface to query the database to find correlated activities or to trace to the root and the path of an intrusion. To prevent intrusions from breaking into the activity database, the activity database should run in a protected environment and should support only a secured query interface to monitors and the rule generator.

The activity database consists of a set of tables, each corresponding to the filtered log entries from a monitor. Each table describes some measurable and distinct aspect of system behavior. For example, in our prototype implementation, there is a table of system calls which records <*processID, processName, systemCall, date, timeStamp, etc*>, a table for network connections recording <*startTime, duration, sourceHost, destHost, bytesFromSource, bytesFromDest, processId, etc*>, and another for system behavior recording <*date, time, eventCode, pid, user, parentID*>.

The database schema is crucial to correlate audit data from different sources. Correlations between logs exist when two log entries have the same or related values for one or more fields. For example, a network connection might be associated with a given system call if the time is within a certain distance and both belong to the same process. Two system calls are correlated if they are made by two different processes that are executed by the same user. Two file I/Os can be correlated if they are operating on the same file or the same directory. Any such correlations can be easily queried using a SQL statement.

In addition to the output of monitors, the activity database also stores "rules" (models and signatures) to describe appropriate or illegal system behavior. Since data can be retrieved in so many different ways from the activity database, it is possible to store several different granularities of a rule. In general, the most commonly used rule set would apply to any system-level behaviors; however, the activity database can also store rules describing how a particular instance (e.g., a user, an application or a file) should behave. For example, the activity database can store rules describing the network traffic generated by FTP or the SMTP daemon. Although instance-specific rules would be too expensive to use when filtering all the data streams, they would be of invaluable use in identifying (or ruling out) potential causes of anomalies.

Another key advantage of the UCLog architecture is the ability to trace back and identify the source and the execution path of the occurring intrusion. At an intrusion, a system administrator who is assigned to analyze the intrusion might ask questions like "What process was running when a large spike of network activity occurred yesterday at midnight?". In most current systems, the administrator needs to examine the logs manually in order to find out the answers to his/her questions. But in UCLog, the activity database supports various queries to provide answers to these questions. To further simplify the administrator's job, some simple tools can be built on top of the activity database with a user-friendly interface and relieve the administrator from writing SQL statements.

One of the most challenging issues of the activity database is scalability. If the system is running for a long time or on a large-scale network, the size of database may quickly increase to Giga or Tera bytes. One

solution toward reducing the footprint of the activity database is to investigate algorithms to compress old events and remove obsolete events. Another solution is running the activity database on a dedicated server.

4.4 Rule Generator

The rule generator is responsible for constructing the models for normal behavior and signatures for previous intrusions. The behavior models are used to detect any anomalies, while the signatures are used to detect known intrusions [4].

To model normal behavior, the rule generator retrieves the training data from the activity database. It is convenient to query particular data from the activity database to generate instance-specific rules via SQL statements. For example, to generate rules for normal system call sequences of a specific application in a specific time period, the generator only needs to query the data of normal system call logs with the WHERE conditions of *processName* and *timeStamp* of an SQL statement. Then the training data is formatted and fed to modeling algorithms. In our current implementation, we use a data-mining algorithm, RIPPER, to generate rules for system call sequences, and statistics methods for other logs.

In contrast, since a signature-based IDS depends on the nature of various known intrusions, most intrusion signatures may come from outside of the system, possibly from an anti-virus library. The rule generator can import such signatures from other systems as well as from intrusions captured as part of training data. Data-mining techniques suitable for the latter.

The rules can be stored in the database for future use, or directly returned to the monitor that initiates the generating rule query. When applying the models to anomaly detection, the IDS can decide what percentage of the activity to flag as abnormal by verifying if the activity follows the behavior models.

4.5 Application Programming Interface

An application programming interface (API), to allow an application or a logger to communicate with a specified monitor, will be implemented in future versions. It would also be useful to incorporate some

mechanism for specifying non-trivial correlations between two audit tables. In our current implementation, logging imposes only semantic information to the logged records; correlation occurs only at query time. Future versions may include some mechanism for the user to specify a relationship between tuples or fields.

5 Evaluation Methodology

The goal of our evaluation is to answer the following questions: "Can UCLog improve the accuracy of intrusion detection?", "Is it efficient?", "Does it impose high space overheads?".

5.1 Experimental Architecture

We have implemented the proposed architecture on two 2.56 GHz, Pentium 4 PCs with 512 MB RAM. Each machine has a 32 GB hard drive with an Intel Ethernet Adapter. Machine 1 runs Red Hat Linux 7.3 and acts as the e-mail server. Machine 2 runs Windows 2000 Service Pack 3[1], serving as a client machine. The UCLog architecture is implemented on Machine 2.

We use several different loggers to capture data from multiple vantage points. For network behavior we use WinDump [14], a Windows port of the popular tcpdump. We use TDIMonitor version 1.01 [34], a logging utility that monitors the Transport Driver Interface. The information provided by WinDump and TDIMonitor are combined into the network traffic monitor and the activity database in order to associate a particular process ID with each network connection. To capture system call behavior we use APISpy32 v3.0 [21] with some modifications that allow us to capture the system time of each system call. Due to some limitations of APISpy32 we also use FileMon v5.01 [35] to capture certain file I/O operations. We use the built-in Windows 2000 security logging features in order to capture process information (start time, end time, process ID, and parent process ID) that we later use in correlating the results from different logs.

[1]Earlier versions of Windows contain a bug that prevents us from retrieving the correct process ID for each activity (see the Microsoft Knowledge Base Article 277743).

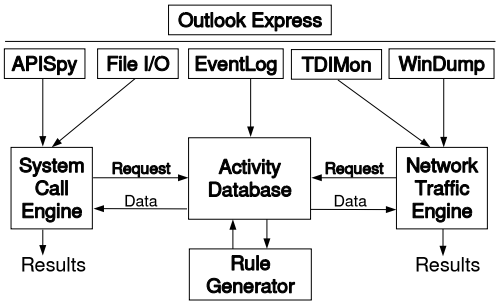


Figure 3. Experimental architecture

In Fig. 3 we give an illustration of our testing architecture. The architecture consists of five loggers, two monitors, a centralized activity database and a rule generator. The data from the five loggers is stored in the central database. While system calls can be logged without any processing, network traffic requires some pre-processing. The network traffic is stored on a connection basis. All of the traffic for a particular connection is aggregated together and stored as one entry in the table. When the data is retrieved from the database it is also necessary to perform some calculations, such as number of connections within the past 10 seconds, in order to help detect an intrusion.

5.2 Intrusion Detection Through Data Mining

While others have used data mining techniques to detect intrusions [23], to our knowledge none of them have correlated information from multiple logs in the data mining process.

When using these data mining techniques to distinguish between an intrusion and normal behavior, it is necessary to capture logs that describe both behaviors. We selected Outlook Express as the entry point for our intrusion. First, we use the loggers to capture several hours of *normal* end-user behavior to use in the training phase of our IDS. The traces record around 800,000 activities. Then, we captured several hours of *abnormal* behavior. Abnormal behavior consists of opening an infected e-mail message and executing the virus. We take roughly 80% of the data points from each set to train the data mining-based rule

generator and use the remaining 20% of the data for testing.

The training process is performed similarly to Lee's study [23]. The system call sequence is broken up into smaller subsequences of 13[2] system calls for training, and each new subsequence is labeled as either normal or abnormal, depending upon which trace it comes from. It is necessary to understand that a normal trace is likely to contain some abnormal subsequences, but the percentage is quite small. Also, an abnormal trace will contain some normal system call subsequences since opening an infected e-mail message does contain some normal operations. These subsequences are fed into the data mining tool RIPPER [12, 13]. RIPPER takes the system call sequences and generates a set of rules that is used to classify new sequences as either normal or abnormal. Once this training has taken place we are ready to start detecting intrusions.

5.3 Log Files

For testing, we used five loggers to collect audit data. Each logger produces a lot of entries; therefore, some filtering is necessary before adding the data into the activity database.

- **System Call Log** – Generated by APISpy32, contains a list of system calls executed by all running processes on the system. We extract the system call, process ID, and time of execution.
- **WinDump Log** – Generated by WinDump. It contains the contents of the UDP and TCP packet headers of all traffic to and from the client machine.
- **TDIMonitor Log** – Generated by TDIMonitor. Captures network traffic at a different layer than WinDump. This logger is able to associate a process ID with each connection. This information is combined with the WinDump log to give us a process ID with each packet header.
- **File I/O** – Created by FileMon. It records all of the file I/O on the system. This information is combined with the system call log and stored in the database.

[2] Our tests showed that size 13 provided optimal results.

- **System Events** – Contains the system events generated by Windows 2000. We configured this log to record information on the creation and termination of all processes. This information includes process ID of each process and its parent process, along with the user-name that created the process.

5.4 Experiment Procedures

We compare the results of UCLog with a baseline IDS. UCLog and the baseline system are very similar. The primary difference between the two is that UCLog explores correlations by using the activity database, whereas the baseline system does not use correlations. The baseline system is very similar to the one used by Lee [23]. In our comparison, we mainly examine the abnormal percentage for both normal and abnormal runs.

In our first set of experiments we measure the baseline results just using the IDS without taking advantage of correlations. We use the 20% of traces that are not used in training as our testing data. The rules that RIPPER generated are applied to the traces. Each sequence of 13 system calls is classified as either normal or abnormal according to the rules. Next, a window of size 7[3] is passed over the list of sequences. If more than half of the sequences in that window are determined to be abnormal, then that region is considered abnormal. Once that window passes over all of the data we can determine what percentage of the trace is "abnormal". For a trace that does not contain an intrusion, we expect the percentage of abnormality to be quite small. If an intrusion takes place, we expect the percentage of abnormality to be noticeably larger. This procedure is similar to many other data mining-based intrusion detection systems [3, 23].

In our second set of experiments we measure the results of using the UCLog architecture to exploit correlations. Once again, we use the 20% of the traces that were not used in training the IDS. The network monitor observes the network traffic and in the meantime also queries the activity database to check if correlated system call data has shown similar behavior. The network monitor combines the system call behavior and the network behavior and makes a judgment on whether

[3] On our tests, a post processing window of size 7 provided optimal results.

the entire system is suspicious. If only one of the network or system call monitors shows abnormal behavior, the threshold for abnormality is decremented, and vice versa.

6 Evaluation Results

Our results show that UCLog can significantly improve the accuracy of intrusion detection. In abnormal runs, UCLog can improve the confidence level of abnormal behavior. In normal runs, UCLog can decrease the number of false positives. Moreover, UCLog is relatively efficient and has reasonable space requirements.

Figure 4 compares the results without correlations (baseline) and with correlations (UCLog). The numbers are measured in terms of percentage of abnormal regions. The higher the percentage, the more likely an intrusion has occurred. If we compare two IDSs, the better IDS should give a higher abnormal percentage for an abnormal trace, and give a lower abnormal percentage for a normal trace. The results show that UCLog performs better than the baseline system and improves the intrusion detection accuracy.

For abnormal traces, Fig. 4(a) shows that the percentage of abnormality is higher when correlation is used. For example, in a30, UCLog shows 31% abnormality, versus 19% for the baseline system. This indicates that UCLog has higher confidence in reporting intrusion warnings.

For normal traces, UCLog reports lower abnormality than the baseline system (see Fig. 4(b)). For some traces, the percentage abnormality with UCLog is significantly lower. For example, with n10, the baseline system reports 1.4% abnormality, whereas UCLog reports only 0.8% abnormality. Therefore, if 1% is the cutoff, the baseline would report a false intrusion but UCLog would not.

6.1 Monitoring File I/O

We have also examined whether monitoring file I/O can be useful in improving intrusion detection accuracy. Pennington et al. [29] presented a storage-based IDS, which monitors access to specific files or directories. The administrator is able to describe which files or directories the IDS should monitor and the specific restrictions for that file

or directory. We implement a similar monitor for our architecture to demonstrate how monitoring and correlating an additional source can help to improve the IDS accuracy.

We added a file I/O monitor that watches the *c:\recycler* directory. The Yaha virus used in our experiments creates several files inside this directory. This is actually a common technique that current worms and viruses often use to conceal their whereabouts. Our new file I/O monitor watches any I/O into this directory and generate warnings upon file creations.

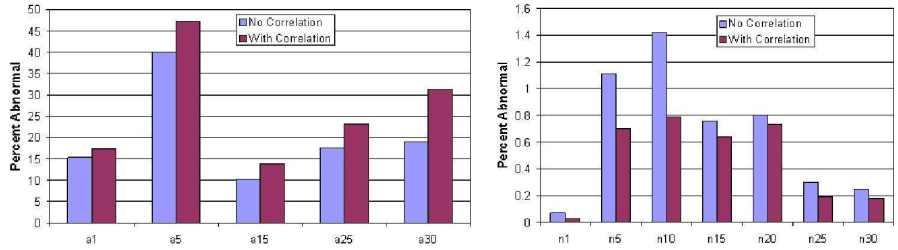
The Windows address book is often compromised by a virus and used to find other potential victims. The Yaha virus reads the address book and sends itself to the contacts. We decided that for testing purposes, no process should be able to secretly access the address book other than Outlook Express or Explorer. In fact, in recent versions of Outlook if you try and access the Contacts information from another program you are prompted for permission. So, our new monitor also looks for any access to the address book by an unauthorized process.

The information from our file I/O monitor is correlated with the other logs in the same way as we correlated the network traffic logs. If we detect any files created in the *c:\recycler* directory, or if we detect any unauthorized access to the Windows address book, we query the database for the network traffic information and the system call information. Then we reprocess the system call data taking into account any anomalies in the network traffic along with the new information from the file I/O monitor. If the file I/O monitor detects the expected abnormal behavior, it increases the abnormality number.

Figure 5 shows the results with the new file I/O monitor. Each abnormal trace contained the suspicious behavior that we were monitoring for. The normal traces did not contain any of the suspicious behavior. Notice that as we correlate more logs, our ability to detect intrusions increases. For example, in a30, the abnormality is 48% with correlations, whereas the abnormality without correlations only gives 19%, a factor of improvement of 2.5.

6.2 Query Performance

To evaluate the usefulness of our trace-back capabilities, we measure the average time to perform several common queries to the activity



(a) Abnormal traces (b) Normal traces

Figure 4. Comparison of UCLog with baseline system. Figures are not in the same scale

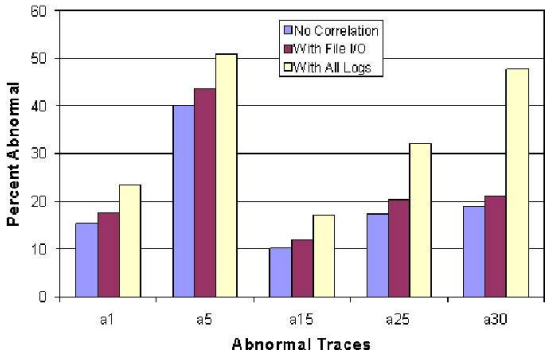


Figure 5. Percentage of abnormality when using no correlation, when correlating file I/O with system calls and when correlating all three logs

database. This was done by selecting several different parameters for each relevant query, and performing multiple runs for each set of parameters. The results are summarized in Table 1. All four simple queries (1–4) return results within 0.01 to 0.43 seconds, which indicates the UCLog architecture can be used to detect intrusions in a timely manner. Query 5 combines two of the other queries, as might be done while investigating an attack. Even for more advanced queries such as 5, the average time to run them is still small.

Table 1. Micro-benchmark results

ID	Query	Avg. Time
1	Select all system calls occurring in some time range	0.196 sec
2	Select all system calls belonging to some process in a given trace	0.425 sec
3	Select all program names belonging to some parent	0.011 sec
4	Select all process IDs receiving packets in some time range	0.082 sec
5	Query 4 and then Query 1 (with processID filtering)	0.6 sec

6.3 Space Requirement

In addition to facilitating queries and detection, the structured database format is more space efficient than storing raw logs. The sum of the three tables stored requires 90,561,664 bytes. In contrast, the total size of the unprocessed log files is 720,896,371 bytes. This 87%

reduction in the size of data that must be stored is a result of eliminating duplicate data especially between the APISpy and FileMon logs, and as a result of removing the data packets from the WinDump log.

7 Related Work

Our work builds upon a large body of previous work. ReVirt [16] proposed by Dunlap et al. is one such example. It records all actions below the virtual machine. It also uses fault-tolerance techniques to protect the integrity of the logging system, and allow ReVirt to replay execution instruction-by-instruction even if the kernel of the operating system is compromised. However, their study does not address how an intrusion would be detected. The UCLog architecture would fill this missing part in the ReVirt system to detect intrusions.

Another related work is secure storage. The self-securing storage system (S4) [39] maintains multiple file versions for recovery and logs all operations for auditing. Based on S4, Pennington et al. [29] propose storage-based intrusion detection that allows storage systems to transparently monitor abnormal activity. In this system, the IDS embedded in a storage device continues to work even if the client systems are compromised. Their study only monitors file I/O operations, whereas UCLog monitors activities from multiple sources.

Schneier and Kelsey [36] describe a computationally cheap method to prevent log files from being read or undetectably compromised by the attackers, by using encryption, authentication, and one-way hash functions. This technique would be useful to integrate into UCLog to ensure the integrity of audit data.

Much work on OS-based intrusion detection has focused on approaches that exploit system call interposition and tracing [18, 20, 23, 22, 7, 40]. The first two studies detect anomalies by extracting patterns of system call sequences from traces and then monitoring for unknown patterns. Lee and Cabrera et al. [23, 7] employ data mining techniques to extract patterns from system call traces for intrusion detection. Warrender et al. [41] compare several methods of representing normal behavior of system calls, such as enumeration of observed sequences, methods based on relative frequencies of sequences, data mining techniques, and hidden Markov models. Ko et al. [22] integrate intrusion detection

techniques with software wrapping technology to improve effectiveness. Wagner et al. [40] propose to detect intrusions via static analysis. Their idea is to constrain the system call trace of a program's execution to be consistent with its source code. When an intrusion occurs, it causes the program to behave in a manner inconsistent with the intent of its author. The biggest difference between Wagner's and our IDSs is that our intrusion detection model is built from the normal system call trace itself using logs, while Wagner's specification is constructed from the program code by static analysis. In many cases, the source code is not provided by the vendor. Even if the source code is available, the limitation of compilers such as the common aliasing problem, makes this solution not applicable to many large commercial server programs.

In [24], Lee et al. propose an architecture for a real-time based system. Such a system consists of sensors that filter and format data, detectors that analyze sensor data for attacks, a data warehouse which provides models of attacks for use by the detectors, and a module to update the data warehouse. Although these sensors have a standardized interface, correlations are not exploited.

Among others, the IDES project and the CIDF [38] standard would allow intrusion detection systems to use multiple kinds of log files. The IDES project, an early design for an intrusion-detection system, formats logged data so that it can be organized by user behavior [15]. The description given in [15], however, is more a high-level discussion of what log information would be useful for an IDS, rather than a consideration of how multiple kinds of logged data can be practically correlated. CIDF [38], the Common Intrusion Detection Framework standard, has been proposed to coordinate distinct intrusion detection systems. CIDF follows a model similar to [24]. Components share tagged data, using a Lisp-like unified format. There is evidence that an XML protocol may become standard [28]. While this standard would allow log files to be used interchangeably within a given IDS; our goal is to allow multiple logs to be used together.

Logging has also been used to improve system reliability [9, 33, 32, 30]. DejaVe [9] is a record/replay tool for Java. By logging the logical thread schedule information, DejaVe can force the exact same schedule during re-executing. Ronsse et al. [32] propose RecPlay that combines record and replay techniques so that it allows debuggers to

debug parallel programs in a deterministic mode.

Recently, Wu et al. [42] proposed CIDS, a Collaborative Intrusion Detection System that combines alarms from multiple IDSs (Snort, Libsafe and Sysmon) and uses a Bayesian network to aggregate and combine the alarms to improve accuracy. The input data, learning method and architecture differs from UCLog, but the basic idea is the same: to improve IDS accuracy by correlating information from different and preferably disjoint sources. We plan to compare UCLog with CIDS in the future.

8 Future Work

Our work has several limitations. First, we have only evaluated our system in Windows 2000 with only the Outlook Express application. It would be interesting to evaluate other systems and other applications. Second, we have not yet integrated our system with a rollback-capable system such as S4. This remains as our immediate future work. Third, it would be interesting to see whether multi-stream data mining would be helpful to exploit correlations. Lastly, we have not compared UCLog with other intrusion detection methods such as the ones using compiler analysis. We believe both static analysis and dynamic analysis of log files would be useful for intrusion detections and can complement each other.

9 Conclusions

This paper proposed UCLog, a unified logging architecture that effectively explores activity correlations for intrusion detection. UCLog organizes audit data from heterogeneous sources into a structured way and is closely integrated into the intrusion detection system. It also exploits activity correlations between different components to improve the intrusion detection accuracy. Moreover, it can trace back to the source and the execution path of an intrusion, relieving administrators from manually examining the logs to find such information. By extending the UCLog architecture by adding additional specific loggers and monitors, the user can tailor the IDS and make it closely monitor critical applications.

We have implemented UCLog on a real system running Outlook Express on Windows 2000. Our prototype architecture uses MySQL to manage the activity database and data mining tools to generate rules and detect anomalies. We evaluated UCLog using normal and abnormal traces. Using these traces, we studied how correlating activities from the system call sequence, network activity, and file I/O can help improve the IDS accuracy. Compared to a baseline system that only monitors a single component, our system that correlates system call sequences, network traffic and file I/Os improves the accuracy by up to a factor of 2.5. UCLog also reduces false alarms when the system is perfectly normal.

References

[1] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou, and K. Rowe. Log correlation for intrusion detection: A proof of concept. In *Proc. of the 19th Computer Security Applications Conf. (ACSAC 2003)*, Dec. 2003.

[2] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99TR-028, School of Computer Science, Carnegie Mellon University, 2000.

[3] S. Axelsson. Research in intrusion-detection systems: A survey. Technical Report 98–17, Dep. of Computer Engineering, Chalmers University of Technology, Dec. 1998.

[4] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99–15, Dep. of Computer Engineering, Chalmers University of Technology, Mar. 1999.

[5] D. Barbará, J. Couto, S. Jajodia, and N. Wu. ADAM: A testbed for exploring the use of data mining in intrusion detection. *ACM SIGMOD Record*, 30(4):15–24, 2001.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proc. of the 16th Intl. Cryptology Conf. (CRYPTO 1996)*, Aug. 1996.

[7] J. Cabrera, L. Lewis, and R. K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4):25–34, 2001.

[8] CERT Coordination Center, Carnegie Mellon University. CERT/CC overview incident and vulnerability trends, May 2003. <http://www.cert.org/present/cert-overview-trends/>. (Feb. 2004).

[9] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proc. of the 2nd SIGMETRICS Symp. on Parallel and Distributed Tools*, Aug. 1998.

[10] Cisco Systems, Inc. Netflow services and applications, a whitepaper, July 2002. <http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_w%p.htm>. (Feb. 2004).

[11] C. Clifton and G. Gengo. Developing custom intrusion detection filters using data mining. In *Proc. of the Military Communications Intl. Symp. (MILCOM 2000)*, Oct. 2000.

[12] W. W. Cohen. Fast effective rule induction. In *Proc. of the 12th Intl. Conf. on Machine Learning*, July 1995.

[13] W. W. Cohen. Learning trees and rules with set-valued features. *AAAI/IAAI*, 1:709–716, 1996.

[14] L. Degioanni, F. Risso, and P. Viano. Windump: tcpdump for Windows, Jan. 2002. <http://windump.polito.it/>. (Feb. 2004).

[15] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 5th Symp. on Operating Sys. Design and Implementation (OSDI 2002)*, Dec. 2002.

[17] farm9.com, Inc. Harvester architecture overview, Aug. 2002. <http://farm9.org/Harvester/ArchitectureOverview.php>. (Feb. 2003).

[18] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proc. of the IEEE Symp. on Research in Security and Privacy*, May 1996.

[19] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., 2000.

[20] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[21] Y. Kaplan. Api spying techniques for windows 9x, nt, and 2000, Apr. 2000. <http://www.internals.com/ar-ticles/apispy/apispy.htm>. (Feb. 2003).

[22] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proc. of the 9th USENIX Security Symp.*, Aug. 2000.

[23] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proc. of the 7th USENIX Security Symp.*, Aug. 1998.

[24] W. Lee, S. Stolfo, P. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang. Real time data mining-based intrusion detection. In *Proc. of the DARPA Information Survivability Conf. and Exposition (DISCEX II)*, June 2001.

[25] R. Lemos. Counting the cost of slammer. In *CNET News.com*, Jan. 2003.

[26] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proc. of the 16th ACM Symp. on Operating Sys. Principles (SOSP 1997)*, Oct. 1997.

[27] Microsoft Corporation, MSDN Library. Logging application, server, and security events, 2004.

[28] S. Northcutt and J. Novak. *Network Intrusion Detection: An Analysts' Handbook*. New Riders, 2000.

[29] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. Technical Report CMU-CS-02-179, School of Computer Science, Carnegie Mellon University, Oct. 2002.

[30] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. of the 29th Intl. Symp. on Computer Architecture (ISCA 2002)*, May 2002.

[31] L. Rohde. Yaha virus lingers into the new year, Jan. 2003. IDG News Service.

[32] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Sys. (TOCS)*, 17(2):133–152, 1999.

[33] M. Ronsse and W. Zwaenepoel. Execution replay for treadmarks. In *Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing*, Jan. 1997.

[34] M. Russinovich. TDIMon, July 2000. <http://www.sysinternals.com/ntw2k/utilities.shtml>. (Feb. 2004).

[35] M. Russinovich and B. Cogswell. Filemon for Windows, July 2003. <http://www.sysinternals.com/ntw2k/utilities.shtml>. (Feb. 2004).

[36] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and Sys. Security (TISSEC)*, 2(2):159–176, 1999.

[37] Snort Project, The. Snort: The open source network intrusion detection system, June 2004. <http://www.snort.org>. (Jun. 2004).

[38] S. Staniford-Chen, B. Tung, and D. Schnackenberg. The common intrusion detection framework (CIDF). In *Information Survivability Workshop*, Oct. 1998.

[39] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. of the 3rd Symp. on Operating Sys. Design and Implementation (OSDI 2000)*, Dec. 2000.

[40] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symp. on Security and Privacy*, May 2001.

[41] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symp. on Security and Privacy*, May 1999.

[42] Y.-S. Wu, B. Foo, Y. Mei, and S. Bagchi. Collaborative intrusion detection system (CIDS): A framework for accurate and efficient ids. In *Proc. of the 19th Computer Security Applications Conf. (ACSAC 2003)*, Dec. 2003.