UNIVERSITY OF OSLO
Department of Informatics

User and group profiling based on user process usage

Master Thesis

Edson Ochoa
Oslo University College

May 23, 2007

User and group profiling based on user process usage

Master Thesis

Edson Ochoa
Oslo University College

May 23, 2007

Abstract

User profiling based on process usage is on approach for adding an extra security layer to our computer systems. In addition it can be of great value for classification of a company/school network and the their user groups. Groups, or classes of users, in a company might belong to the same division or department that solve similar tasks. In a company, accountants probably use the same set of tools, as would a group of students in a graphic design class. Studying if these similarities in the process that they use, can say something about an individual or a group. It is valuable in the terms of analyzing individual user and group behavior. Recognizing individual users behavior, provides the possibility of an extra layer of security in the form of an authentication scheme. Recognizing group behavior might provide valuable insight when it comes to building a profile for a new user, and see why this user fits the group or not.

This thesis makes use of statistical approaches to discuss the possibility of using process profiling to classify users into groups.

Acknowledgements

I would like to thank my supervisor, Simen Hagen for his patience and good advice. His support and advice has been greatly appreciated.

Edson Ochoa - May 2007

Contents

List of Figures

Chapter 1

Introduction

1.1 User profiling

A persons behavior and characteristics is an important asset in any information system. A pattern in the way someone acts can help identify an individual and setup an environment suitable for this specific person. In computer technology, social networks, intrusion detection systems, recognizing a user based on behavior is of great value. Specially in e-commerce systems, income can be increased if an e-shop keeps track of a customers behavior. Using the history and behavior of a customer, an e-shop can provide you with direct offers to products you most likely will buy. As for instance Google has realized, user behavior is an important source of income when it comes to aiming ads directly based on e-mail messages, search strings etc.

User profiling in security systems is valuable, techniques as program profiling, mouse movement, keystroke dynamics [1, 2, 3, 4, 5, 6] may add new ways of user authentication and verification.

1.2 Security

Security is as strong as its weakest link. Social engineering [7], user negligence as persons willingly give away passwords in order to solve a problem, is always a security hazard for computer systems. The problem is not the security itself, but how the users bypass it. One problem with computer systems is really knowing the correct user has logged in, the system has usually no idea if the user logged in is indeed the user that is supposed to be logged in. A user that gives away his/her password (most often against policy) on the phone to the helpful person in the helpdesk, might be opening his/her account for malicious use. On the other hand, abnormal behavior does not necessarily imply misbehavior or malicious behavior. By itself, this is an interesting field of research, complex in its nature, as everything that has to do with human nature is. Human beings are unique and unpredictable, but somehow in our own fashion we tend to classify each other, and see the similarities and patterns that makes us alike. Recognizing similarities and patterns is something we as human beings do easily, both conscious and unconscious. Even our body, the immune system is able to detect a change of pattern, or recognize a pattern which enables it to take appropriate action.

Combined with other security schemes, user program/process profiling can be a possible solution to this problem could be user program profiling. If a user can be authenticated and verified based on her or his behavior, additional mechanisms as *keystroke dynamics* [6] can be used to re-authenticate or verify a user. Having a user profile, or a behavior blueprint can be valuable in determining if a user is the legitimate one. Nevertheless considering the unpredictability of our nature it is a rather hard task, and there are always errors.

1.3 Classification

Classification of users and similarities as behavior, music taste, shopping patterns etc. can be a hard task when looking at a diverse set of users. Considering user program profiling, various programs/processes can be used for the same tasks and users might not necessarily chose the same tool. *Ontologies* for group classification could be an approach. An *ontology* for the class of programs could be created and match different users and groups based on the tasks they solve rather than specifically on the tool. This layer of abstraction can make it hard to see the differences between users in order to use the data for anomaly detection, but considering the potential of ontologies and the use of it for anomaly detection in other fields it is something worth looking at [8].

Research with user program profiling has been performed with good results [1, 2, 3, 4, 5]. It does not seem to be a very common approach, and there isn't a great deal of research done. Several approaches on program profiling have been based on Unix system command line. Although there has been promising results, considering the way users interact with a computer today through a graphical user interface, these experiments do not cover the trend in todays “modern” use of computers. Nevertheless, these methods are often part of larger systems and the approaches and results are valuable [9].

One of the main problems with the user program profiling approach is the collection of real data and anomalous data to perform the experiments. The diversity of the data and the parameters one can use can be overwhelming and makes it hard to decide on what parameters to conecentrate on. Another aspect of program profiling is what users might see as an invasion of their privacy. It is not necessarily comfortable to everyone that someone could check on how and what you use your computer for.

1.4 Goals

This thesis aims to see if it is possible to determine if a simple statistical approach can provide information about users and the group they belong to. The areas of interests are evaluating if a user profile based on the processes used, is good enough to be matched with the user behavior over time. Considering a company with group profiles, what does it mean if you as an accountant do not fit the rest of the accountants? It is important to keep in mind, as mentioned earlier that anomalous behavior is not equivalent to misbehavior.

The assumption in this thesis is that the processes owned by a user (looking at the process table) will classify a user to a certain group. Considering a company this will mean that the different departments, for instance as *economics, accounting, IT services* etc. will use different set of tools, and from these different department the user behavior should denote them to belong to a certain group (as the various departments). Assuming this, a system administrator could assign the new users to a group and use a group profile to ”monitor” a users behavior. Furthermore users can be looked at individually to see how much the behavior, even though belonging to the same group, varies.

Last, but not least, individual user profiles may particularly valuable in a non-group environment. As for instance considering different users in one host or network where the users do not belong to a certain group.

1.5 Thesis outline

This thesis will be structured as follows:

Chapter 1 introduces the reader for the considerations and goals of this thesis. The reader is introduced to the topics that are going to be discussed throughout the thesis

Chapter 2 contains a literature survey on the subjects relevant to the thesis. This will provide the reader with the appropriate background information, motivations and previous research. The goal is introducing the reader to the aspects and challenges on the research topic.

Chapter 3 explains the methods used to solve the thesis. How data was collected and filtered, the approaches used to build profiles and comparisons of profiles and user process usage. This chapter also introduces the reader to the tests performed on the collected data.

Chapter 4 presents the results achieved in comparing user profiles, matching users to profiles and discusses the patterns and anomalies of the results.

Chapter 5 contains the conclusions for the thesis, based on the discussion of the results achieved. It also explains problematical aspects of the research in this thesis and introduces the reader to possible solutions to these problems. The reader is also introduced to possible future research in the research field.

Chapter 2

Background and Previous Research

In a computer system, knowing what is happening on the system at any given time, is of great value. Auditing and profiling, which in a computer system will contain knowledge of the system is great for solving problems. Profiling is especially important in security aspects of a system like anomaly detection. Although collecting information about a system is valuable, the usual disadvantages as overhead due to monitoring and storage space restricts the scope of auditing. This chapter will introduce the reader to profiling and its areas of use.

2.1 Intrusion Detection

Intrusion detection and prevention systems use a variety of strategies to a protect a system against malicious attacks. An IDS (Intrusion Detection System) system can usually be categorized into *misuse* detection and *anomaly detection*. Misuse detection can be described as a method of detecting a known attack. An IDS will search its database to find a pattern or signature that matches an already known attack, and act according to policy. Anomaly detection systems try to find anomalies by comparing normal traffic, process usage etc. and figure out if it might be malicious [10]. There are great difficulties when it comes to anomaly detection since abnormal does not necessarily have to mean malicious. A key element of an anomaly detection system is to keep down the number of false positives.

There are several approaches to anomaly detection schemes in areas as network traffic, process usage and user behavior.

2.1.1 Anomaly Detection

Part of the abstract of what is considered as the first paper on anomaly detection describes an approach to an intrusion detection model [11] as follows.

> “The model is based on the hypothesis that security violations can be detected by monitoring a system’s audit records for abnormal patterns of system usage.”

Anomaly detection as mentioned in the past section handles system occurrences that are not normal. What is normal is defined in a profile which works as a baseline on how the system is supposed to behave. This means that an anomaly detection system needs time to build a profile and “learn” what is normal. Furthermore, the issue of flagging certain actions as dangerous is a complex task, considering that abnormal not necessarily has to be malicious. There are several approaches to how an anomaly detection system can build profiles and the difficulties are generally the same. Determining how much data to gather and what parameters to use has a great impact on what the end result will be [11]. In addition, one has to consider the overhead and overall resource usage of a large scale anomaly detection system. Another factor is to determine a threshold of when the system behaves different enough from the normal behavior to determine this as an anomaly.

2.2 Profiling

Profiling is widely used in various scenarios. In addition to being an essential part of an anomaly detection model it serves well in various areas. Customer profiles are commonly used in e-commerce and recommender systems [12], where a customer profile is built in order to recommend the customers products that fits to their needs and what they want. The Internet community is currently seeing a growth in various social networking websites where profiling is used in order to recommend music, friends, movies and so forth.

Profiles for behavior of persons, computer systems, networks etc. is an essential part in order to provide a better service and understanding of the subject profile. While in the area of anomaly detection profiles provide a baseline of normal behavior. There are a vast amount of approaches for a system to build and “learn” a profile. In [11], a statistical approach is presented. Other researchers as in [9, 4, 5, 1, 2, 10] use support vector machines, sequence matching and neural networks.

2.2.1 System Calls and Commands

There is a lot of intrusion detection research done with system calls, privileged processes and user commands. One of the most interesting topics is to consider anomaly detection as analogous to the body's immune system. An interesting approach is presented in [13], which looks at a systems privileged processes and the system calls from these. The idea is to create a system that can distinguish between its normal behavior and anomalous behavior, similar to how the immune system does this by looking at a characteristic structure called a *peptide*, which is a a short protein fragment. The authors in [13] describe the *short sequences of system calls* as analogous to a *peptide*.

The notion of looking at privileged processes makes sense considering that the processes or commands that pose a dangerous threat are the privileged ones. Furthermore, as discussed in [14, 13] an individual host for each host is necessary in order to prevent the weaknesses of a general profile or database in a network system. Although there are great advantages as portability and maintainability, the weakness of an error that can be exploited in an entire system makes it too insecure.

2.2.2 User Profiling

A different approach to anomaly detection is *user* program profiling. The idea is to monitor the processes of the distinct users of a system. User profiling involves solving two problems: *authentication* and *insider misuse* [1]. By building a profile based on process usage and process parameters, users can be differentiated from each other and anomalies can be detected. There are combined methods to provide user authentication and anomaly detection by program based profiles. The focus here is taken from *system* behavior to *user* behavior. Now, the problem with user profiling based on processes is human nature. If it is compared to profiling based on a processes' collection of system calls or profiling system performance it turns out to be more complex. This is due to the human factor of the data. Human behavior is generally unpredictable and a vast amount of anomalous data has to be tolerated. An example is a computer user in a company, getting a new assignment might mean a change of behavior when it comes to process usage.

User program profiling is a recent field of research. A handful of experiments have been performed [1, 2, 3, 4, 5]. As mentioned above, the goals of user-based program profiling is to provide *authentication* and detect *insider misuse* or *user impersonation*. For a system to have the possibility to detect the insider threat in addition to provide an extra layer of authentication a user based profile of host/system usage needs to be built. This is similar to user profiles for e-commerce [12, 15] and social networks here.

The approach presented in [1, 2, 3] is based on process information from the distinct users *process table*. [2] looks at a login session, which is defined to be everything that happens from the time a user logs in, until the user logs out. Several assumptions are made for this approach. An important one is mentioned in [3].

> We expect that the "behavior" of any given user, if defined appropriately, would be very hard to impersonate.

The problem is that human behavior as user profiling is, is very hard to define. The problem [1, 2, 3, 4, 5] try to solve is user authentication. Can a specific user session show similarities to another session by the same user? And will a session from one user be different than the one for another? The parameters used to describe a user session profile in [2] are:

Contents of the title bar whenever

- A new window is created
- Focus switches to a previously existing window
- Title bar changes in current window

For process table:

- Birth
- Death
- Continuation (existing process uses up CPU time)
- Background
- Ancestry

Timing

- Date and time of login
- Clock time since login
- CPU time

This data is based in three types of records *Window*, *Process* and *Ancestry*. More features are described for each of these records in addition to the ones mentioned above which provide a rich dataset. One of the main problems with research in this area is collecting data. The need of real "normal" data and for instance anomalous data from real time environments as a company, is hard to retrieve.

2.3 Ontologies

Ontologies can be described as a way of achieving shared understanding. Systems can be designed independent from each other, an ontology is the solution for these separate systems to work together [16]. An example is to think about how we as human beings relate to an object, and automatically make references to this object. In [17] an example is made by thinking about a table, and the attributes we give them. As humans we make associations surrounding an object, as in the table example, it might have the following attributes;

- Piece of funiture
- Table made of wood
- Four legs
- Bought in IKEA

To describe this for computers, an ontology is created in order to make a system understand these associations by classifying a table as being for instance a child of furniture, with attributes as material, where it came from and so on.

There a lot of research in progress in this field which is vast. Since its ultimate goal is to bind everything together so all systems can share their knowledge.

2.4 Summary of Background and Previous Research

Auditing and profiling in general is a powerful tool for a system administrator. Event logs can provide vital information for forensics in a case of intrusion detection. Furthermore it might be a great asset in determining the best possible configuration for a hosts performance. Logged data is vital in all aspects of anomaly detection systems. There has been done research in fine grained and more rough profiling.

An important aspect of profiling and auditing is to find out what data you need to solve a specific task. It is important to find a balance considering the impact monitoring can have on a system.

Chapter 3

Methodology

The analysis was done in three phases.

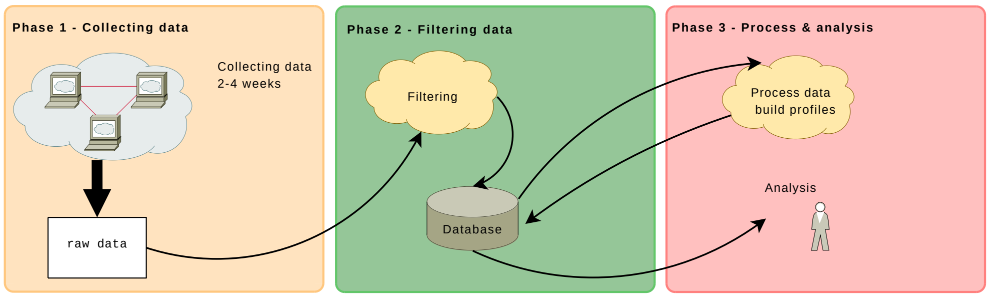

Figure 3.1: Phases of methodology

Phase 1: a 2-4 week period collecting process logs for users, explained in section *3.3*

Phase 2: a period of data filtering and trimming. All the process date was stored in a database with its relevant information, explained in section *3.4*

Phase 3: analyzing the filtered data, presenting the data in a suitable manner, and do calculations on them like building profiles and comparing users. Explained in section *3.5*

3.1 Equipment

The equipment used in the experiments is a range of normal laptops and workstation computers. Hardware will not be discussed, since it has nothing to do with the collection of data, or the tests ran on it.

3.2 Tools

The following tools were used for logging, parsing and analysis:

- Python
- Bash
- PHP
- MySQL
- Windows Management Instrumentation (WMI) / Visual Basic Script (VBScript)

3.3 Phase 1 - Collecting data

The data pool needed for this experiment is hard to get. For this thesis it was gathered from 22 different users in a span of 2 - 4 weeks. These were distinct users, nobody defined to be in any specific group. As mentioned, similar data has been used in other experiments [1, 2, 3, 4, 5], but this data was not available anymore when starting this thesis. The author contacted T. Goldring [1, 2, 2] and C.N. Manikopoulos [5] with no or little response. The available links in [1, 2, 3, 4, 5] are not longer available.

The subjects used different flavors of Linux and Windows. Two scripts were made for the purpose of logging in these operating systems. The users were simply asked to add the logging script to their startup sequence in order to start logging everytime they logged in the their graphical user interface. The scripts can be found in the appendix section *5.4 on page 55* and section *5.4 on page 54*.

3.3.1 Process metrics

The metrics contained for a process is:

- **Creation date:** This is the date / timestamp of when the specific process was started.
- **Caption:** The name of the process
- **Process Id:** The process id of the specific process in a session.
- **Elapsed time:** The total amount of seconds a specific process has been running.
- **User / Owner:** The owner of the process.

A process is considered to be *unique* when the *creation timestamp* and the *process id* of the same program for instance *firefox.exe*, do not match.

3.3.2 Logging/Collecting

The logging scripts were ran at the users personal computers for a period of 2-4 weeks. The scripts for both operating systems (Windows and Linux) started logging every time the users logged in to their graphical user interface. Each time the scripts started, a new file was created, marked with the username and a timestamp. The script would then start retrieving the process status information. The scripts would simply run continuously and retrieve information from the user process table every 5 seconds, and write to a log file.

At the end of the logging period (2-4 weeks) these files were sent to the author for processing and filtering.

3.3.2.1 Windows - *processloggerd.vbs*

The Windows script was based on Visual Basic scripting using the Windows Management Instrumentation tools (WMI). WMI is pre-installed in Microsoft Windows Vista, Windows Server 2003, Windows XP, Windows ME and Windows 2000. This made it possible to create a script where the users did not have to do much to get it working. Basically the users had to download the scripts from a website.

The instructions there where simple:

- Download the file *processloggerd.vbs*.
- Add it to the Start Up folder in the Windows menu.

The script *processloggerd.vbs* would then start every time the user logged in. The first time it starts, it creates a folder named *pslogs* in the home directory. The log files would then be created and saved in this folder. In addition, the script shows a pop-up every time the user logs in, telling it has started. The script will then run continuously during a session and write to the log file every 5 seconds Stopping the logging process is as simple as deleting the script from the startup folder.

3.3.2.2 Example - Windows log format

The format of the log file in Windows was as follows: *StartDate, Elapsed, ParentId, Id, Caption, Description, Path, Owner*

```
20070228130832.625000+060,49016,3380,3748,rundll32.exe,rundll32.exe,D:\WINDOWS\system32\rundll32.exe,,User/Host
20070228130833.328125+060,49016,3380,3924,wscript.exe,wscript.exe,D:\WINDOWS\System32\WScript.exe,,User/Host
20070228130835.968750+060,49014,844,308,wmiprvse.exe,wmiprvse.exe,,,NETWORK SERVICE/Host
20070228131135.593750+060,49013,3248,1176,explorer.exe,explorer.exe,D:\WINDOWS\explorer.exe,,User/Host
20070228150514.312500+060,48833,844,1168,LVCOMSX.EXE,LVCOMSX.EXE,D:\WINDOWS\system32\LVComsX.exe,,User/Host
20070301024516.718750+060,42014,612,2656,logonuiX.exe,logonuiX.exe,D:\WINDOWS\system32\logonuiX.exe,,SYSTEM/Host
20070301024527.531250+060,12,1168,3384,drwtsn32.exe,drwtsn32.exe,,,User/Host
20070301024527.578125+060,1,1168,4424,drwtsn32.exe,drwtsn32.exe,D:\WINDOWS\system32\drwtsn32.exe,,User/Host
```

3.3.2.3 Linux - *processloggerd*

The Linux script *processloggerd* created for this task is a BASH-script. It provides a bit more process information than its Windows equivalent. It is also a simpler script than Windows, with added functionality. In addition to doing the same as the Windows script, it also compresses the log files when you login to your user interface. The Linux script was also downloaded from a website, and the subjects followed similar steps as for Windows.

- Download *processloggerd*
- Add it to your graphical interface startup

Since the subjects using Linux did not necessarily use the same graphical interface. No detailed instructions were given according to how to setup the script for it to start every time the user logs in. Although, considering that the Linux users who collected information are skilled computer users, this was no problem.

As with the Window script, *processlogerd* creates the folder *pslogs* in the subjects home directory and saves its logfiles to that folder.

```
ps -u [user] -o lstart,time,c,etime,ppid,pid,stat,\%cpu,\%mem,comm
```

3.3.2.4 Example - Linux log format

A short example of the log format in Linux follows:

```
                 STARTED     TIME  C     ELAPSED  PPID   PID STAT %CPU %MEM COMMAND

Tue Apr  3 11:45:35 2007 00:00:00  0    03:04:32  3997  4008 Ss+   0.0  0.1 bash
Tue Apr  3 11:45:47 2007 00:00:00  0    03:04:20  3997  4047 Ss    0.0  0.1 bash
Tue Apr  3 11:45:51 2007 00:00:00  0    03:04:16  4047  4052 S+    0.0  0.1 ssh
Tue Apr  3 11:46:14 2007 00:08:37  4    03:03:53     1  4080 TNl   4.6  2.9 beagled-helper
Tue Apr  3 11:50:27 2007 00:00:00  0    02:59:40     1  4662 S     0.0  0.3 gconfd-2
Tue Apr  3 12:08:56 2007 00:08:32  5    02:41:11     1  6490 S     5.3  1.7 gnome-system-mo
Tue Apr  3 14:46:34 2007 00:00:00  0       03:33 12184 12186 S     0.0  0.1 sshd
Tue Apr  3 14:46:34 2007 00:00:00  0       03:33 12186 12187 Ss+   0.0  0.1 bash
Tue Apr  3 14:50:07 2007 00:00:00  0       00:00  3692 12324 R     0.0  0.0 ps
```

3.3.3 Script similarities

As can be seen, the Windows script (*processloggerd.vbs*) and Linux script (*processloggerd*) are similar. The information used form the logs are the same, and in their final form for analysis, the process information format was equal. The main job of these scripts is to:

- Start every time the user logs in to the graphical interface.
- Retrieve user process information every 5 seconds and write this to a file.

3.4 Phase 2 - Parsing and filtering

The parsing and filtering of log information went through several stages. For convenience, the data was stored in a database. This makes the data easily accessible and editable by SQL-queries and any language that support this. As can be seen, the format of the logs in Linux and Windows are quite different although they essentially show much of the same information. After parsing this data and storing it in a database, the data will be identically categorized. The reason for this was to make the analysis part easier considering the amount of data that has to be processed. In addition, the final key process information is not dependent on platform, the parameters are the same, and the analysis for these is identical.

One of the main tasks of parsing and filtering the logfiles is to extract the essential information mentioned in the *metrics* section. A regular log grows everytime the file is written to, in this case every 5 seconds. Much of this information is repeated, especially considering that most programs are run for more than 5 seconds. In addition, some text and parameters were added to the log files in order to help initial analysis and debugging of the logging scripts. These lines need to be ignored, in addition to get rid of the duplicate processes provided by the raw logs. The scripts are available in the appendix in section *5.4 on page 57* and section *5.4 on page 60.*

3.4.1 Parsing - goals

The goals of the parsing scripts are to remove information line in the logs. As for instance description of what the different parts of the logs mean. A quick summary of this can be shown as following:

- Remove lines containing information about format and/or markers for debugging/analyzing.
- Identify duplicate processes
- Retrieve the information of the duplicate processes
- Create an entry for each unique process, with the correct parameters
- Create new list fulfilling the requirements above

In these bulletpoints it can be seen that the requirements of the information we need at the end for analysis is: *Creation date timestamp*, *unique processes*, *elapsed time*, in addition to who owns the process. (In this case per user). As mentioned earlier, a process is considered unique when the process id and creation timestamp do no match for a process with the same name. For instance:

Creation	Process id	Name
1981-12-07 00:05:00	5	firefox.exe
1981-12-07 00:04:00	5	firefox.exe

The above processes will not be considered to be the same, since the creation date differs.

Creation	Process id	Name
1981-12-07 00:05:00	5	firefox.exe
1981-12-07 00:05:00	5	firefox.exe

In this example, the process will be considered as the exact same one, thus only adding it once to the new filtered list of processes.

3.4.2 Parsing - Windows

Regular expressions was a widely used tool in both the process of parsing the Windows and Linux logs. The Windows logs had the advantage of being comma delimited as shown in section *sec:windowslog*. The parsing of the Windows logs can roughly be explained in three steps.

Step 1: In order to complete the "goals of parsing" as described in section *3.4.1 on the facing page*, line by line went to different steps of filtering. The first and most obvious filtering was removing informational lines that the author set in for reference and readability when debugging the log format. Another issue in Windows was that all the version prior to Windows Vista that were monitored, wrote not only *user* specific process information to the logs but added other users as *network services*, *administrator* etc. These lines were identified by *regular expressions* in addition to *information lines* as datestamp for reference, were discarded at the start of the processing.

Step 2: The second part of the parsing splitted each line on commas to retrieve the parameters and prepare them to create a dictionary[1]. All the parameters except the elapsed time was added as a key to the dictionary *trimmed[pid + "," + creation + ","*

[1]a dictionary in *python* is equivalent to a hash array in *perl*

+ *caption* + *","* + *owner] = elapsed.* Then, whenever a new dictionary item is to be added, the key is checked to see if the same process already exists in the dictionary. If it does, and the elapsed time is larger, it is replaced, since we want the total elapsed time of the process to be shown for each distinct process. If the value isn't larger and the process already exists in this dictionary it will ignore it.

Step 3: The third part retrieves all the values of the newly created dictionary which conforms with the metrics listed in *3.3.1 on page 15* and adds them to a *MySQL*-database explained in section *3.4.5*.

3.4.3 Parsing - Linux

Regular expressions were extensively used in the linux log parser. This is due to the format of the Linux log files as shown in section *3.3.2.4 on page 17.* As for the Windows log parsing, the parsing process of the Linux logs can similarly be described as a three step process.

Step 1: Similar information lines were added to the Linux logs as in Windows for readability and debugging the logs. The first step easily discards these lines, since they are of no importance to the metrics needed. Using the `-u username` part of the `ps` command only displays the process table information of the selected user, which relieves us of the problem of having to filter out other uses as the case is with the Windows logs. Although, the problem of the monitoring of a system affecting the system it is monitoring, which is explained in section *3.7 on page 29*, resulted filtering out the `ps` altogether.

Step 2: The second part is very similar to the Windows paring, the difference is in the extensive use of *regular expressions*. A dictionary was created as in section *3.4.2 on the preceding page*, with a minor technical difference due to the way `ps -u username` **only** retrieves information from the selected user. The dictionary ended up as *trimmed[pid + "," + creation + "," + caption] = elapsed.* The same tests regarding the processes' prior existence and elapsed time as in section *3.4.2* are performed.

Step 3: As in section *3.4.2* the values retrieved from the dictionary are added to a *MySQL*-database explained in section *3.4.5 on page 22*.

3.4.4 Parsing and filtering summary

As explained in section 3.4.2 on page 19 and section 3.4.3 on the preceding page the output that the scripts generate is identical. The parsing scripts were similar in the way they were ran too, a minor difference is the parameters given to the command. The windows parser would be run with the command: `winparser-0.1.py logfile`, similarly in Linux `linparser-0.1.py logfile username`. The reason for the extra parameter *username* for the *linparser-0.1.py* is that the logfile itself, does not contain any username. For that reason, the *username* is given as a parameter.

For instance, to parse a windows log, the following command would be entered in the command line to read through all the log files of a user:

```
bash$> for logs in *.log;do echo $logs; winparser-0.1.py $logs;done;
```

An extract of the output a parsing operation provided is illustrated here:

```
 /media/disk/logs/user/01-04-2007_104115-User-process.log
Old list: 109923
New list: 48
Adding to database...
Inserted records: 48
Updated records: 0
Done!
-------------------------------------------------------------
/media/disk/logs/user/01-04-2007_155624-User-process.log
Old list: 116698
New list: 85
Adding to database...
Inserted records: 85
Updated records: 0
Done!
--------------------------------------------------------------
/media/disk/logs/user/01-04-2007_221031-User-process.log
Old list: 172468
New list: 56
Adding to database...
Inserted records: 56
Updated records: 0
```

```
Done!
```

The **old list** is the number of lines from a single log file, while **new list** is the list generated of unique processes in the log file. **Inserted records** is the number of records from a file that has been added to the database. **Updated records** is the number of records that are updated, in the case of an existing entry in the database with a lower elapsed time. This is explained in section 3.4.2 on page 19 and section 3.4.3 on page 20.

The command entered to do the same task for linux users would be:

```
bash$> for logs in *.log;do echo $logs; linparserr-0.1.py $logs username;done;
```

The output for a Linux user identical to the output for a Windows user.

3.4.5 Database

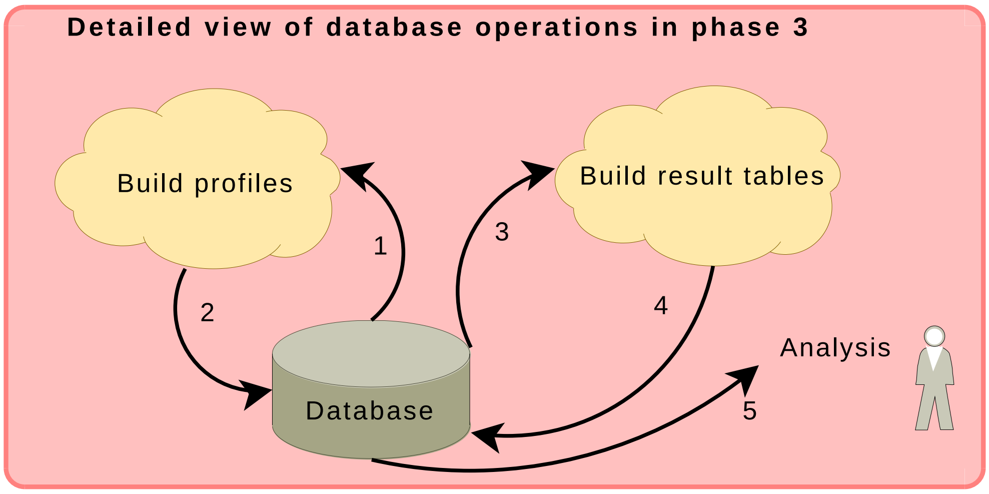

Figure 3.2: Phase 3 details - database

The database is essential for the construction of profiles and manipulation of data. The raw log data is filtered to the database, each unique process, is added to a table *processentries* which contain the metrics described in section *3.3.1 on page 15*. An overview of the database is available in the appendix in section. 5.4 on page 53.

From *processentries*, the tables *weekly* and *daily* are generated by PHP scripts. These two tables are structured as follows:

Table: daily/weekly

- intDailyId/intWeeklyId - *Primary key.*
- intDaynr/intWeeknr - *Day/Week number, counting from 1 to the last day (total days/weeks of logging.)*
- strCaption - *Name of the process.*
- intElapsed - *Daily/Weekly use of the process.*
- intInstances - *Daily/Weekly instances of the process.*
- intUserId - *Foreign key to the user table.*

These tables do not contain any information about the date and time of the process being started. What the script processed and added to these tables, is the daily process usage of each user. Counting from *day 1*, until the last *day*. The weekly table is exactly the same, but here it will add the processes used during a whole week. These tables are essential for creating the weekly and daily profiles. The data from the tables *daily* and *weekly* is again processed, and from this data, the profiles are created. The table *daily*-data is used to create the daily profile (table: *profile_daily*) and the same for the table *weekly* (table: *profile_weekly*). The reason for doing this in several steps is to filter and structure the data, making it easier to present it for analysis. The *profile_weekly* and *profile_daily* tables are as follows:

Table: profile_daily/profile_weekly

- intProfileDailyId/intProfileWeeklyId - *Primary key.*
- strCaption - *Name of the process.*
- avgInstances - *Average daily/weekly instances.*
- sdInstances - *Standard deviation of daily/weekly instances.*
- totInstances - *Total instances from the whole period.*
- avgElapsed - *Average daily/weekly elapsed time in seconds.*
- sdElapsed - *Standard deviation of daily/weekly elapsed time in seconds.*
- totElapsed - *Total elapsed time in seconds for the whole period.*
- intUserId - *Foreign key to the user table.*

3.5 Phase 3 - Analysis and methods

3.5.1 Profile creation

There are two types of profiles created for performing tests:

- Daily
- Weekly

As explained in the database section, these profiles are generated from the information available in the tables *weekly* and *daily*. The profiles are simply generated by using the average values each day/week. Gaussian distribution is assumed and the standard deviation for elapsed time and instances for each day or week are calculated. This is easily done by using SQL to retrieve data from *daily* and *weekly* tables. An example SQL string is:

```
SELECT DISTINCT intUserId, strCaption, AVG(intInstances) as avgInstances,
        STD(intInstances) as sdInstances, AVG(intElapsed) as avgElapsed,
        STD(intElapsed) as sdElapsed, SUM(intElapsed) as totElapsed,
        SUM(intInstances) as totInstances
FROM daily
WHERE intUserId=5
GROUP BY strCaption
```

The above SQL string retrieves the average instances, elapsed time and the standard deviations to this per process, the `GROUP BY strCaption` part groups everything by process name, this way the values retrieved are per process. The information this SQL string retrieves is then inserted to the *profile_daily* table. This is done for each day/week for all users and for each unique daily/weekly process.

In a user profile, each process has the following information available:

- Processname
- Average instances
- Standard deviation for instances

- Total instances
- Average elapsed times in seconds
- Standard deviation for elapsed time in seconds
- Total elapsed time

This is the same in weekly and daily profiles, the difference is only the timespan in which the average and standard deviations are calculated. A user profile is the collection of distinct processes and the information each process holds.

3.5.2 Profile comparison

The comparisons are done on the users static data. Each day or week is compared to the user profile, day 1 compares to daily profile, day 2 compared to daily profile and so on. The same goes for the weekly usage and weekly profiles. The users are not only matched to their own profiles, but also to the profiles to the other users. This is platform dependent, which means Linux users will compare themselves only to Linux users and Windows user only to Windows users. In addition to comparing a users day/week to the respective profiles, comparisons of the user profiles will be done in order to see how they fit together. This might say something about similar users.

The way it is possible to compare the users to all the profiles is by finding the overlapping process for the chosen user at the chosen day, only the values of the overlapping processes between the user and the profile are used for hit rate and distance/angle calculations.

3.5.2.1 Matching - hit rate

A simple form of testing how well a users daily usage matches its profile is by defining a hit rate. This can be measured in percents. The daily/weekly usage (for each day/week) will be compared to the respective daily/weekly profile, and also for testing all users to all user profiles. The total number of distinct process in the profile depicts 100% of the processes. Meaning that if a daily/weekly usage matches this fully inside the standard deviations for *instances* and *elapsed time* the hit rate will be 100%. It is important, at least when comparing one users process usage to different users profile to find the overlapping processes.

The formulae for this is:

$$Hitrate = \frac{\text{daily/weekly matching process overlap}}{\text{total processes in profile}} \tag{3.1}$$

3.5.2.2 Euclidean distance and L^p - norm

Euclidean distance [18] is a way of using the Pythagorean theorem repeatedly in order to calculate the distance between two points in Euclidean n-space. The distance between two points $P(p_x, p_y)$ and $Q(q_x, q_y)$ is defined as:

$$d(P,Q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{3.2}$$

The Euclidean distance for two points in n-space is then:

$$d(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2} \tag{3.3}$$

$$d(P,Q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{3.4}$$

L^p-norm [19] can be described as Euclidean distance in the *n-th* root, Euclidean distance is also known as L^2-norm. This provides the following formula:

$$d(P,Q) = \sum_{i=1}^{n} (|p_i - q_i|)^{\frac{1}{\times}} \tag{3.5}$$

3.5.2.3 Angle between vectors

Another method to be used to compare profiles and user daily/weekly usage is calculating the angle between two profiles, or day number x vs. daily profile. This is illustrated in figure 3.3 on the facing page as a 2 dimentional graph.

The idea here is that the more similar the profiles are, the smaller the angle will be compared to two non similar profiles. Another property of the angle method which is different from the L^p-norms is that the angle method is more "sensitive" to the pattern of the profiles rather than to the distance itself. For instance if profile 1 has the value

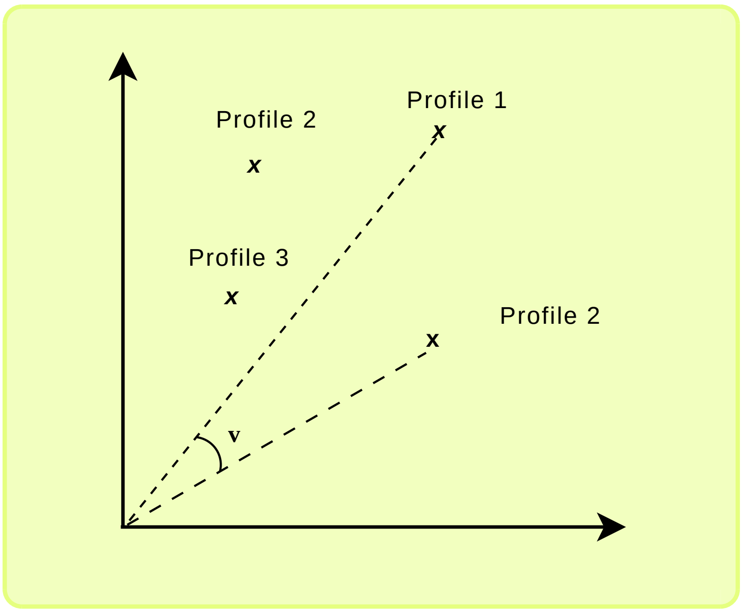

Figure 3.3: Example of the angle between two points

$P1_x = 5$, $P1_y = 10$ and profile 2 $P1_x = 500$, $P1_y = 10000$, the angle between these two profiles will be $\theta = 0$.

The angle between two points through origo is found by:

$$X \cdot Y = |X| \cdot |Y| \cdot \cos\theta \tag{3.6}$$

$$\cos\theta = \frac{X \cdot Y}{|X| \cdot |Y|} \tag{3.7}$$

$$\theta = \arccos\left(\frac{X \cdot Y}{|X| \cdot |Y|}\right) \tag{3.8}$$

The scalar product of the vectors is defined:

$$|v| = \sqrt{v_i^2 + v_i^2 + \ldots + vn^2} \tag{3.9}$$

This provides us with the equation:

$$\theta = \arccos\left(\frac{x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n}{\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \cdot \sqrt{y_1^2 + y_2^2 + \ldots + x_n^2}}\right) \tag{3.10}$$

The same principle is applied for *n-dimensional* space.

3.6 Tests

The tests performed are grouped by operating system. Since a Linux user and a Windows user practically do not have any overlapping processes, a comparison between them are not done.

3.6.1 Profile test comparison

The profiles built from the user data are compared to each other using:

1. Angle distance
2. L^p-norm, from L^1 to L^5 distance

These tests will provide a picture of which users have profiles that are close to each other. It might point out something about similarities in user behavior, and say something about grouping user based on similar behavior.

3.6.2 User process usage hit rate compared to profiles

1. Percentage match between a user and the daily/weekly process usage to his/her own daily/weekly profile.
2. Percentage match between a a user and the daily/weekly process usage to all others daily/weekly profile.

These tests will help determine the proximity between a user and the respective user profile (daily and weekly). In addition, by comparing a user daily/weekly process usage to all the other user profiles, it is possible to analyze how well a user matches his/her own profile compared to any other user in the system.

3.7 Discussion

Collecting data was an issue in this thesis. Ideally the data should be gathered from a company and its various departments, or for instance a university and its faculties. In addition, data gathered for an extended period of time, and from more users would have provided a better basis for comparison and analysis. The papers [9, 4, 5, 1, 2, 10] also describe problems with gathering data, often due to privacy issues. The data collected in these papers, are of a more intrusive nature than the data collected in this thesis.

Although, considering the amount of data collected for this thesis, and the resources available, it was enough. The time given to finish this thesis is limited, and that had an impact on the quality of the logging and parsing scripts. Time available also limits the timespan of the logs gathered from various users. The idea, is that in a real time monitoring system, the logging scripts would do the work of the parsing scripts as. In addition to run some comparisons to a profile using live data from the process table.

As a result of the time limit, the logging and parsing scripts are somewhat ineffective, it simply takes time to filter the information within the database to create profiles.This is due to the amount of raw data material collected. Doing this dynamically with a real time system where the logging script or program would be more effective, would not impact the system performance, nor flood the computer with huge logs as in the case with the scripts in this thesis. It is important to keep in mind though, that the sole purpose with these scripts were to gather the information needed and analyze it.

From the available data pool in this thesis, there were only 5 users in what we can say is a group. This makes it hard to say something about the group comparison and classifying based on “neighboring user profiles”. Therefore, as mentioned in this section, more data from an environment with departments/faculties etc. would have been ideal to analyze this problem more thoroughly.

3.7.1 Observing Data

In the first stages of analysis, when looking at the process data of a Linux user, it could easily be seen that a lot of noise was produced by the `ps` process. Like in Heisenbergs Uncertainty Principle, the observer disturbed what was to be observed. For that reason, it was decided to filter out the processes generated by the observation scripts. This was done both for Linux and Windows. Although there was a difference with the Windows process logs, they did not have the same impact as in Linux. Nevertheless, the processes were there and they would usually not be part of the ”normal” profile. This considering the monitoring process to not be part of the normal system usage.

The drawback with the decision is that in Linux `ps`, or Windows' wscript.exe are "legal" programs to use, and there is a chance that some of observations of these processes are part of "normal use". By removing them, a potential part of the profile is left out. On the other hand, considering that this was very similar for all the users, it was decided that a parameter less which is mostly noise wouldn't affect the final results. In addition, parsing the Linux logs **with** `ps` included took several hours more than when filtering it out.

Chapter 4

Results

The results from the tests are divided in two sections, Windows users (section 4.1 on the next page) and Linux users (section 4.2 on page 38). The tests performed are identical for both type of users, and are further divided in two sub-sections for individual user analysis and group profile analysis.

The individual user analysis consists of the hit rate match test explained in section 3.5.2.1 on page 25. To summarize, the test consists of testing the users proces usage against all the profiles, including their own. The overlapping processes from each period (day or week) are matched to those of the profiles. If the values are inside of the profiles average +/- the standard deviation, it is counted as a hit. The measure is presented in the percentage of the total hits from a user to a profile. This is done both for the daily and weekly process usage of each individual user.

In this chapter, a **Profile** describes the profile built for a specific user. **Profile 1** is for instance the profile for user 1, based on the daily or weekly process usage from this user. **User 1** then describes the process usage for this user, per day or per week.

4.1 Windows users

4.1.1 Individual user analysis

Table 4.1 on the facing page shows the total hit rate of a user against a profile in percent. The first row of the table illustrates the profiles (example Profile 1, **P1**) and the first column are the users (example: User 1, **U1**). The percent hit rate of **User 2** against **Profile 4** is found by matching the row (**U1**) to (**P4**. Figure 4.1 illustrates the values from table 4.1 on the facing page in a histogram, where the x-axis illustrates the user profiles while y-axis represents the hit rate in percents. The boxes themselves illustrate the users daily process usage.

User to day profile hit rate

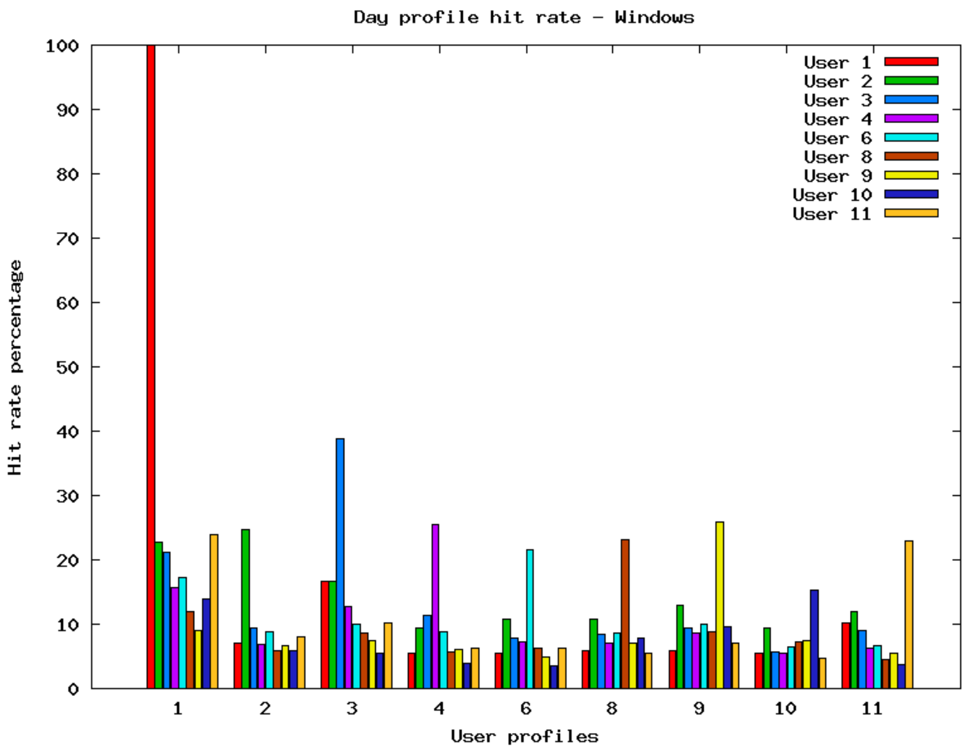

Figure 4.1: Hit rate for user daily usage to daily profile

Figure 4.1 on the preceding page shows a clear trend when it comes to a user matching her/his own profile. The histograms shows a clear peak for the user matching her/his own profile. The following table shows the average hit rate of a user against the profiles. *User 1* works as a verifier of the analysis since it is a short log that spans only for one day. This means that the hitrate is 100% due to that it will always hit its own profile on all values. This is because there is only one instance and elapsed time value for each distinct process.

	P 1	**P 2**	**P 3**	**P 4**	**P 6**	**P 8**	**P 9**	**P 10**	**P 11**
U 1	100,00	7,10	16,67	5,52	5,56	5,79	5,88	5,51	10,29
U 2	22,83	24,77	16,65	9,37	10,87	10,73	12,84	9,43	11,98
U 3	21,10	9,46	38,86	11,39	7,93	8,48	9,50	5,66	8,97
U 4	15,62	6,86	12,80	25,57	7,29	7,10	8,67	5,54	6,31
U 6	17,22	8,74	10,01	8,79	21,67	8,54	9,94	6,52	6,69
U 8	11,96	5,97	8,65	5,75	6,21	23,21	8,67	7,28	4,52
U 9	8,92	6,72	7,53	6,03	4,97	7,13	9,94	7,50	5,47
U 10	1,88	5,84	5,41	3,95	3,45	7,79	8,76	15,29	3,75
U 11	23,99	7,95	10,26	6,18	6,27	5,46	7,14	4,75	22,94

Table 4.1: Windows - Daily hit rate, Users vs. Profiles

User to week profile hit rate

Identical to the *user to day profile hit rate*. Table 4.2 on the facing page illustrates the user to profile hit rate for matching overlaping processes. Figure 4.2 illustrates the values of table 4.2 on the facing page in a histogram.

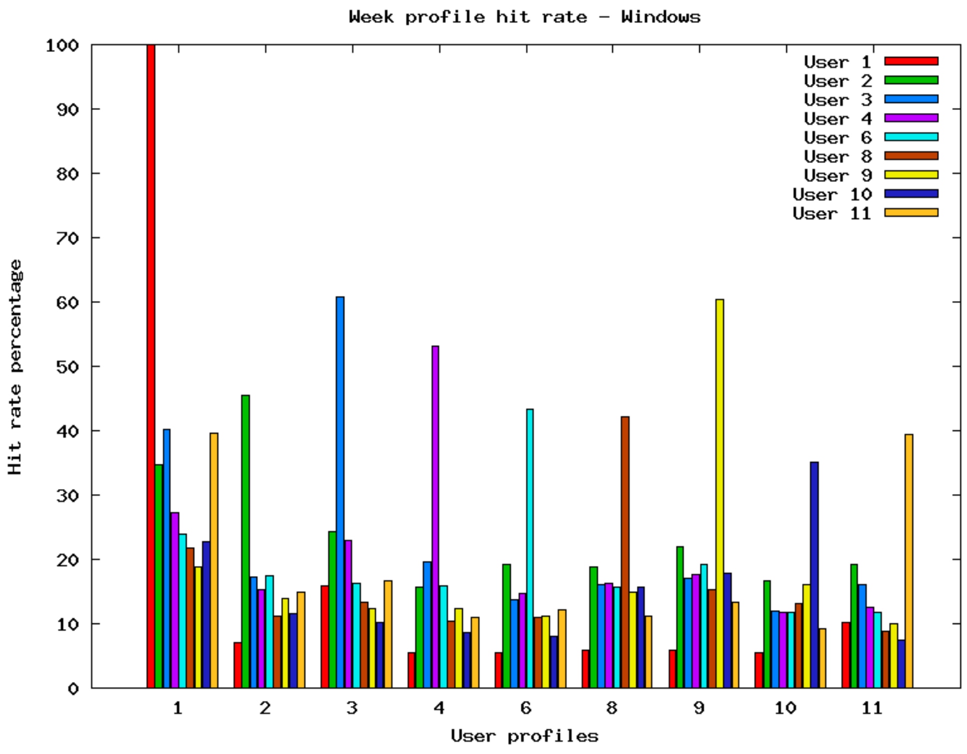

Figure 4.2: Hit rate for user weekly usage to weekly profile

The weekly profile hit rate is very similar to the daily profile hit rate. A difference is that every user generally gets a higher hit rate on every profile, including their own. In figure 4.2 and table 4.2 on the facing page one can see that the hit rate for a user to his/her own profile still is clear.

	P 1	**P 2**	**P 3**	**P 4**	**P 6**	**P 8**	**P 9**	**P 10**	**P 11**
U 1	100,00	7,10	15,97	5,52	5,56	5,79	5,88	5,51	10,29
U 2	34,78	45,44	24,37	15,67	19,22	18,77	22,02	16,76	19,18
U 3	40,22	17,26	60,71	19,66	13,73	16,12	17,06	12,01	16,00
U 4	27,17	15,32	22,90	53,10	14,66	16,32	17,65	11,81	12,57
U 6	23,91	17,42	16,18	15,86	43,36	15,60	19,26	11,81	11,86
U 8	21,74	11,18	13,31	10,46	11,01	42,15	15,29	13,12	8,76
U 9	18,84	13,98	12,32	12,41	11,11	14,88	60,39	16,01	9,90
U 10	22,83	11,61	10,19	8,62	8,02	15,70	17,94	35,04	7,43
U 11	39,67	14,84	16,60	11,03	12,19	11,16	13,24	9,15	39,50

Table 4.2: Windows - Weekly hit rate, Users vs. Profiles

The Windows user to profile comparisons seems to match themselves well. The trend is quite clear, although the standard deviation has to be taken into consideration. These values can be retrieved from the database using *SQL*. Table 4.3 shows the hit rate per week for all users comparing to profile 10.

User	**Profile**	**Average hit rate**	**Standard deviation**
1	10	5.51181102362	0
2	10	16.760404949414	1.1689881715829
3	10	12.00787401575	1.5120365645806
4	10	11.81102362206	2.0074879975067
6	10	11.811023622057	3.7762452939627
8	10	13.123359580085	5.5615800788585
9	10	16.0104986877	2.5982926342709
10	10	35.039370078752	13.586955847832
11	10	9.1535433070762	2.7541473581891

Table 4.3: Users vs. Profile 10

As we can see in table 4.3 on the preceding page, the average hit rate value of for instance **user 10** to **profile 10**, is roughly the double of the other users to the same profile. This is equivalent to the daily comparisons aswell.

Another attribute which is common when comparing the users to the profiles is the standard deviation. A user compared to her/his own profile, will have a slightly higher standard deviation than the rest of the users compared to the same profile. The example in table 4.3 on the previous page shows a standard deviation noticable higher than for user 10 vs. profile 10 than for the rest. The reason can be seen in table 4.4.

User	**Profile**	**Hit rate**	**Week**
10	10	41.7322834646	1
10	10	39.3700787402	2
10	10	52.7559055118	3
10	10	42.5196850394	4
10	10	41.7322834646	5
10	10	29.1338582677	6
10	10	28.3464566929	7
10	10	4.72440944882	8

Table 4.4: User 10 vs. Profile nr. 10

Table 4.4 shows that in Week 8, the hit rate of user 10 to her/his own profile isf 4,7%. This is quite devastating for the calculated average of user 10 to profile 10. In table 4.5 we can look at the closest rival to **user 10** on **profile 10**, which is **user 2**.

User	**Profile**	**Hit rate**	**Week**
2	10	15.7480314961	1
2	10	18.1102362205	2
2	10	15.7480314961	3
2	10	15.7480314961	4
2	10	41.7322834646	5
2	10	18.8976377953	6
2	10	16.5354330709	7

Table 4.5: User 2 vs. Profile nr. 10

As we can see in table 4.5 on the preceding page, the hit rate is significantly lower than **user 10** to **profile 10**. The reason for the “anomaly” in week 8 for **user 10** to **profile 10**, is the lack of process usage in week 8, this can be seen in the daily hit rate of **user 10** to **profile 10** in table 4.6. This has an impact in the standard deviation of the hit rate for **user 10** to **profile 10**, but as we can see by comparing table 4.4 on the preceding page and table 4.5 on the facing page the weekly hit rate for a user compared to his/her own profile is still substantially higher than for the rest of the users compared to this profile.

User	**Profile**	**Hit rate**	**Day**
10	10	4.72440944882	50
10	10	0.787401574803	51
10	10	0.787401574803	52

Table 4.6: User 10 vs. Profile nr. 10 - 3 last days

4.1.2 Group comparison analysis

Compared to the group analysis for the Linux profiles in section 4.2.2 on page 43,the distances between the Windows the profile, both daily and weekly are significantly larger. When listing the profile distances using L^p-norm and the angle ranked by least distance

4.2 Linux users

This section is simlar to section 4.1 on page 32, the hit rate for each user against all the profiles is calculated. Figure 4.3 illustrates the data in table 4.7 on the facing page and figure 4.4 on page 40 illustrates table 4.8 on page 41. The data in the tables show the percent hit rate for the users against all profiles in their respective period (daily/weekly).

4.2.1 Individual user analysis

User to day profile hit rate

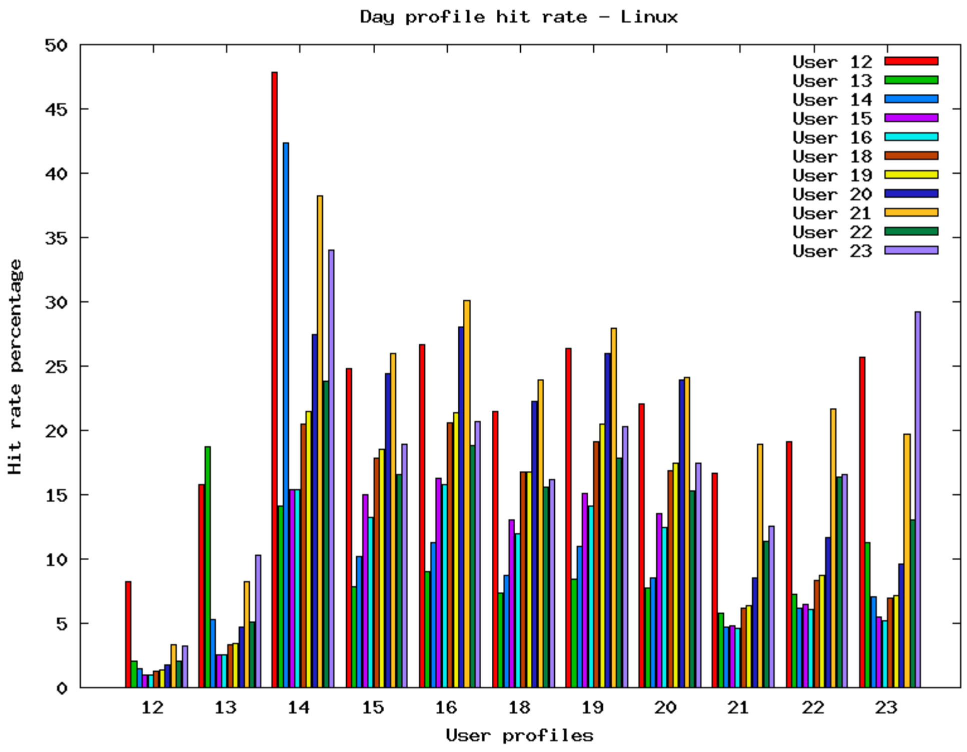

Figure 4.3: Hit rate for user daily usage to daily profile

Figure 4.3 on the preceding page does not show a very clear trend as figure 4.1 on page 32 and figure 4.2 on page 34. The users that seem to hit their profile best are users; 12, 13, 19, 21 and 23. Another trend here is that user 12 is getting a good hit rate on all profiles, even “beating” user 14 to his/her own profile.

	P 12	**P 13**	**P 14**	**P 15**	**P 16**	**P 18**	**P 19**	**P 20**	**P 21**	**P 22**	**P 23**
U 12	8,21	15,81	47,84	24,84	26,67	21,46	26,42	22,05	16,71	19,09	25,67
U 13	2,05	18,75	14,10	7,85	9,07	7,35	8,45	7,75	5,76	7,24	11,30
U 14	1,51	5,31	42,33	10,21	11,28	8,76	10,99	8,54	4,73	6,18	7,01
U 15	1,00	2,56	15,38	14,99	16,28	13,09	15,13	13,54	7,76	6,44	5,51
U 16	0,94	2,55	15,44	13,26	15,81	11,99	14,14	12,43	4,57	6,09	5,23
U 18	1,28	3,34	20,51	17,82	20,58	16,79	19,10	16,88	6,19	8,29	6,99
U 19	1,34	3,46	21,44	18,57	21,36	16,79	20,47	17,49	6,42	8,68	7,21
U 20	1,79	4,68	27,46	24,44	28,01	22,22	25,94	23,90	8,57	11,71	9,64
U 21	3,31	8,25	38,20	25,99	30,05	23,94	27,98	24,07	18,91	21,63	19,66
U 22	2,08	5,11	23,84	16,62	18,84	15,56	17,80	15,29	11,42	16,33	13,06
U 23	3,22	10,31	34,00	18,93	20,67	16,13	20,28	17,49	12,53	16,53	29,18

Table 4.7: Linux - Percent hit rate per day, users vs. profiles

User to week profile hit rate

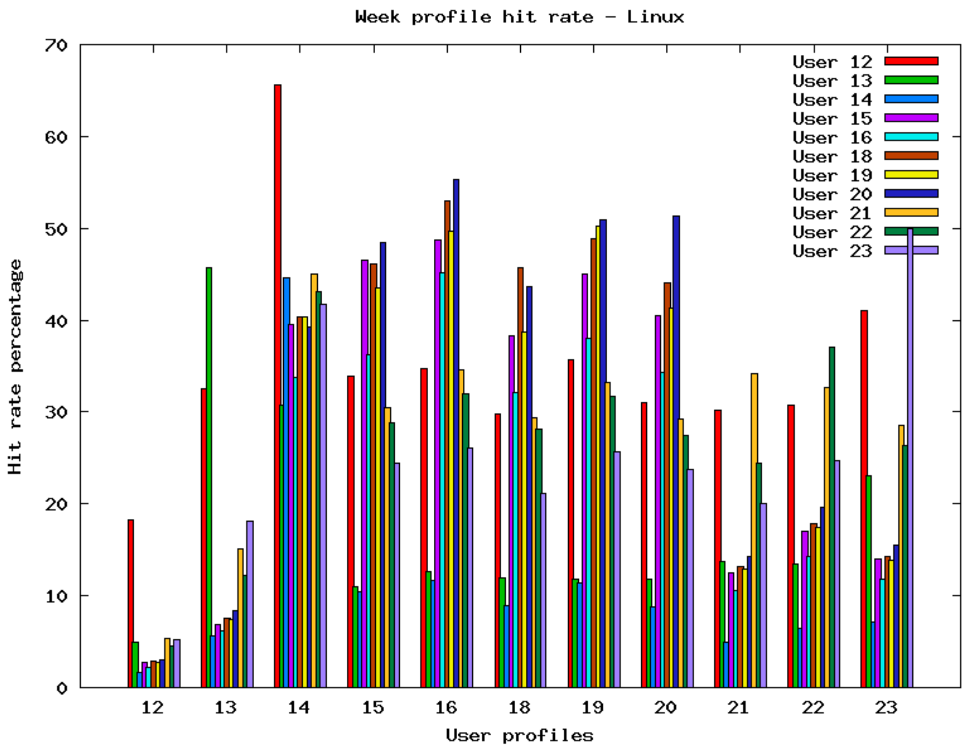

Figure 4.4: Hit rate for user weekly usage to weekly profile

Figure 4.4, not surprising shows the same trend as figure 4.3 on page 38.

An explanation of why user 12 matches all profiles well is the timespan of the logging period for this user. User 12, is by far the user with the most unique processes, and also the longest period of logging. The high hit rate **user 2** gets against **profile 14** is a good example of this. **User 12** has a total of 433 distinct processes, while **user 14** has a total of 39 unique processes. Now if we look at figure 4.5 on page 42, where user and profile is excluded we can see a similar trend.

	P 12	**P 13**	**P 14**	**P 15**	**P 16**	**P 18**	**P 19**	**P 20**	**P 21**	**P 22**	**P 23**
U 12	18,30	35,52	65,64	33,95	34,66	29,79	35,63	31,00	30,20	30,78	41,07
U 13	4,88	45,76	30,77	10,93	12,60	11,96	11,75	11,78	13,77	13,46	23,00
U 14	1,59	5,61	44,62	10,39	11,60	8,87	11,42	8,81	4,96	6,45	7,10
U 15	2,68	6,89	39,56	46,51	48,73	38,29	45,00	40,48	12,49	16,99	14,05
U 16	2,23	6,17	33,70	36,21	45,21	32,11	38,04	34,29	10,55	14,29	11,83
U 18	2,84	7,50	40,29	46,18	53,03	45,66	48,93	44,13	13,21	17,83	14,29
U 19	2,78	7,40	40,29	43,52	49,71	38,73	50,18	41,27	12,92	17,37	13,82
U 20	3,06	8,32	39,19	48,50	55,38	43,59	50,89	51,27	14,29	19,61	15,57
U 21	5,34	15,05	45,09	30,43	34,59	29,38	33,23	29,17	34,21	32,63	28,48
U 22	4,48	12,23	43,46	28,88	31,96	28,18	31,67	27,41	24,41	37,09	26,30
U 23	5,21	18,17	41,67	24,42	26,03	21,13	25,63	23,75	19,97	24,75	50,00

Table 4.8: Linux - Percent hit rate per week, users vs. profiles

In figure 4.5 on the next page where the user **user 12** is excluded we can see that **user 21** takes the "role" of **user 12**. Looking at unique process count[1] again, **user 21** has the second most total unique processes, which is 199. Table 4.9 shows the process count for each user, ordered by the highest number of unique processes. The balance of the amount of unique processes per user seem affects the hit rate results of the users vs. profiles.

User	**Unique Processes**	**Total Processes**
12	433	36943
21	199	13470
22	153	11655
13	139	5633
23	122	9657
18	97	2356
20	90	3242
15	46	1769
19	80	2683
16	73	1495
14	39	265

Table 4.9: Linux users, process count

[1]The unique process count is the total amount of unique process appearences. This is based on the process name.

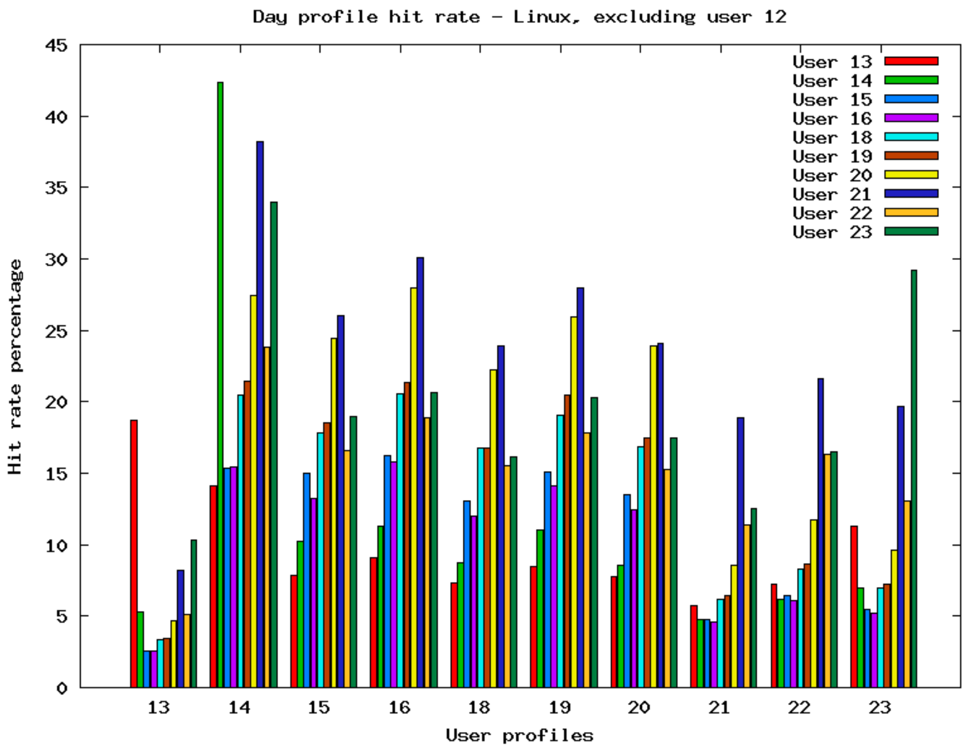

Figure 4.5: Hit rate for user daily usage to daily profile excluding user and profile 12

4.2.2 Group comparison analysis

As opposed to the analysis of groups with the Windows users profiles, the Linux users have a group which we can say is predefined. These are the user profiles of user 15, 16, 18, 19 and 20. Now if we list the 10 users with the smallest distance sorted by L^2-norm, based on the day profiles.

Profile 1	**Profile 2**	**L^2-norm distance**
16	19	36943
15	23	13470
19	20	11655
18	20	5633
15	19	9657
15	16	2356
19	23	3242
16	20	1769
16	23	2683
16	20	1495

Table 4.10: Linux day profiles, 10 shortest L^2-norm distances

Looking at this top 6 hits, we can see that the closest matches are:

1. 16-19
2. 15-23
3. 19-20
4. 18-20
5. 15-19
6. 15-16

All these belong to our “predefined group” except from 15-23. This can be an effect of the differences in total unique processes per user. In table 4.9 on page 41 we can see that for **profile 15**, the unique process count is 86 and for **profile 23** it is 122.

Now, sorting by angle we also see records from the Windows users. These can be discarded considering that it is Profile 1, the “dummy” profile.

Profile 1	**Profile 2**	**Angle**
1	11	0.198730357549
15	23	0.209806715597
16	19	0.282252740249
15	19	0.284005648765
1	3	0.289382672116
19	21	0.291277000583
15	20	0.292745103946
15	21	0.297134277465
1	2	0.303806015512
19	23	0.317341361954

Table 4.11: Linux day profiles, 10 shortest L^2-norm distances

Similarly in table 4.11, looking at the top 6 hits, we get the the following closest matches:

1. 15-23
2. 16-19
3. 15-19
4. 19-21
5. 15-20
6. 15-21

These are similar to the results in table 4.10 on the previous page, altough point *4*, *5* and *6* do not show in table's top 6 profile pairs with shortest distance. Point *5* is present as the 10th row in the same table. For comparison, we can do the same query for the weekly profiles, table 4.12 on the facing page shows the top 6 profile matches with the shortest distance.

Profile 1	**Profile 2**	**L^2-norm distance**
15	18	442.21512271
15	20	598.902983058
19	20	659.99450911
16	18	681.581681837
15	19	712.010405484
18	20	723.30472053

Table 4.12: Linux week profiles, top 6 hits, L^2-norm distances

The 6 pair of profiles with shortest distances are shown in table 4.13.

Profile 1	**Profile 2**	**Angle**
15	18	0.190953421091
15	20	0.236727465788
19	21	0.244862668744
16	18	0.245696845341
16	23	0.254645001453
18	23	0.283548437749

Table 4.13: Linux week profiles, top 6 hits, angle

Table 4.12, using L^2-norm seems to match the "predefined group" better. In table 4.13, the angle comparison shows profile 23 in the top 6 profile pairs with shortest distance. Considering the "prior definition" of the group, the angle comparison does not match as well as lL^2-norm.

The same pattern can be seen when looking at the profiles 15-20 seperately. In most cases, they will match each other, with the angle distance being less accurate if we look at profile 15-20 as a predefined group.

Chapter 5

Conclusions and Critique

Trends in the data can be seen and it would be interesting to see if they still would exist with a larger dataset. The individual and group analysis results are encouraging to pursue further research in the field.

Throughout the analysis, L^2-norm, euclidean distance, was used to see the distance between the profiles. The reason is that it is the most "usual" distance, baseline algorithm used. In addition, the different L^p norms did not provide any variation that affected the way the data could be interpreted. Therefore the analysis was based on the angle and euclidean distance.

5.1 Collecting data

The irregularities in the collection of process logs shows to have an impact on the results. The diversity of the users combined with the differences in the timespan of the logging works to enhance the irregularities. In addition, the analysis on the amount of data collected can only hint about the trends the test methods show. Several factors when collecting data have a negative effect on the results.

One is that the users that contributed with the process logs, with the exception of the predefined group, are individual users. The amount of logs, and behavior vary too much in order to classify them.

Another factor is the timespan of the logging. As mentioned earlier it's roughly from 2-4 weeks.

An interesting factor is how the unique process count affected the individual analysis for Linux users (section 4.2.1 on page 38).

In the future, it would be interesting to see how these tests would work with a larger dataset with a more balanced log timespan between the users.

5.2 Individual analysis

The individual analysis of the users and matching of profiles showed a clear distinction in the effectiveness of the statistical methods used. The Windows users had a much clearer hit rate for their respective profiles than in Linux (section 4.1.1 on page 32). The possible explanations for this is the problems with the data sets described in section 5.1 on the preceding page.

Another parameter that needs to be looked at, is the comparison between a user with few distinct processes and a profile with a large number of distinct processes. As seen in section 4.2.2 on page 43, this has an impact in the individual analysis, and is something to note if similar research is done.

The decision to remove the timestamp of the processes when creating the user profiles has most probably had an impact on the individualities of the user profiles. Using the time stamp of the processes would have provided a better picture of user behavior, and it is clearly something that should be looked at in the future.

5.3 Group analysis

The group analysis can be said to be more successful than individual analysis consideing the Linux users (section 4.2.2 on page 43). There seems to be a clear pattern on the users that belong to a group. In the future, it would be interesting to explore this with data from an organization with various departments. The Linux users 15-20 which can be said to be predefined as a group, matched each other well when using euclidean distance.

For the Windows users it was harder to see any pattern, the profile distance to each other is larger than with the Linux processes. This is probably due to the fact that the Windows users are individual users, where no groups are predefined and it was expected to not see any special pattern here.

5.4 Future research

Ontologies as discussed in the introduction and background, can provide a layer of abstraction. Perhaps it is not necessary to know what specific process/program is used, but what the program is used for. An ontology could be created to map this together, thus a user failing to comply to the group ontology profile, or his/her own profile can be flagged as suspicious. Ontologies [17] could help to an overview of the users behavior and create a "fingerprint" for it. This can be interesting for future. Research would have to be done to determine if an approach to process profiling and anomaly detection can be done with ontologies, the same way as ontologies for network intrusion/anomaly detections system [8].

Future work in this area is rather interesting, considering that it is a somewhat unresearched field. Trying other methods to classify groups, or having a larger data pool available to analyze, would be helpful to determine what methods are effective. There are good results achieved in [1, 2, 3, 4, 5], but even though the data is sanitized, the nature of the data collected is more intrusive than the data collected in this thesis.

An interesting aspect of user profiling is the possibility of adding it as an extra layer of authentication and security. Future research mentioned in [ref] includes mouse movement and keystroke dynamics in order to make a more accurate user profile. The number of parameters that can be added to create a very fine-grained profile have no limits. The question is if it is appropriate. Finding a balance between the simple and complicated is hard in any environment that requires profiling, this is even harder considering human behavior.

The goal of this thesis was to try simple solutions to complex problems, although based on the data and time available, it is hard to come to a definite conclusion. Nevertheless there are trends in the dataset that seem promising, and worth examining further.

Bibliography

[1] Tom Goldring. Scatter (and other) plots for visualizing user profiling data and network traffic. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 119–123. National Security Agency, ACM Press.

[2] Tom Goldring. User Profiling for Intrusion Detection in Windows NT. http://www.galaxy.gmu.edu/interface/I03/I2003Proceedings/GoldringTom/GoldringTom.paper.pdf, 2003.

[3] Tom Goldring. User Profiling for Intrusion Detection in Windows NT Presentation (ppt). http://www.cs.fit.edu/ pkc/dmsec03/slides/goldring03dmsec.ppt, 2003.

[4] Yihua Liao. Windows NT User Profiling With Support Vector Machines. In *Proceedings of the 2002 UC Davis Student Workshop on Computing*, page 64. Computer Science Department, University of California, Davis, 2002.

[5] C.N. Manikopoulos Ling Li, Sui Song. Windows nt user profiling for masquerader detection. In *Networking, Sensing and Control, 2006. ICNSC '06. Proceedings of the 2006 IEEE International Conference*, pages 386–391, April 2006.

[6] Claudia Picardi Francesco Begadano, Daniele Gunetti. User authentication through keystroke dynamics. *ACM Transactions on Information and System Security*, 5:367–397, 2002.

[7] Steve Wozniak Kevin D. Mitnick, William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. Wiley, 2002.

[8] John Pinkston Jeffrey Undercoffer, Anupam Joshi. Modeling computer attacks: An ontology for intrusion detection. In *Lecture Notes in Computer Science*, volume 2820/2003 of *Lecture Notes in Computer Science*, pages 113–135. Springer Berlin / Heidelberg, 2004.

[9] Terran Lane. *Machine Learning Techniques For The Computer Security Domain Of Anomaly Detection*. PhD thesis, Purdue University, August 2000.

[10] Vasant Honavar Dae-Ki Kang, Doug Fuller. Learning Classifiers for Misuse and Anomaly Detection Using a Bag of System Calls Representation. In *Proceedings of the 2005 UC Workshop on Information Assurance and Security*, pages 118–125, 2005.

[11] Dorothy E. Denning. An intrusion-detection model. *1986 IEEE Symposium on Security and Privacy*, page 118, 1986.

[12] Alexand Tuzhilin Gediminas Adomavicius. User profiling in personalization applications through rule discovery and validation. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–381. ACM Press, 1999.

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

[14] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[15] Nigel R. Shadbolt Stuart E. Middleton and David C. De Roure. Ontological user profiing in recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22:54–88, January 2004.

[16] Michael Gruninger Mike Uschold. Ontologies: Principles, Methods and Applications. In *Knowledge Engineering Review*, volume 11, pages 93–136, 1996.

[17] Katrina E Triezenberg Sergei Nirenburg Victor Raskin, Christian F. Hempelmann. Ontology in Information Security: A Useful Theoretical Foundation and Methodological Tool. In *Proceedings of the 2001 workshop on New security paradigms*, pages 53–59, 2001.

[18] David E. Penney C. Henry Edwards. *Calculus*, chapter Appendix B: A6. Prentice Hall, 6th edition, 2002.

[19] Julius O. Smith III. *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*. W3K Publishing, 2 edition, 2007. ISBN 970-0-9745607-4-8.

Appendix

Database Overview

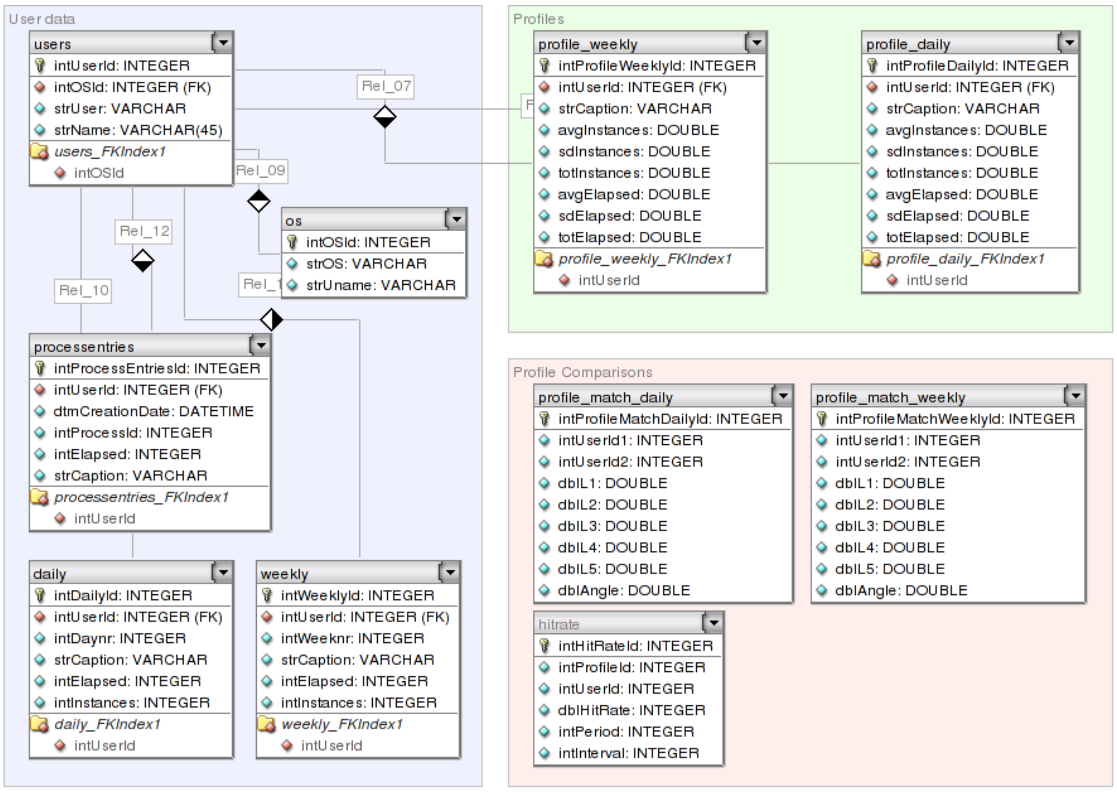

Figure 5.1: Database overview

processloggerd

```
#!/bin/bash
#
###########################
# Process logging script
# Edson Ochoa
# Oslo University College
###########################


# Setting some variables...

USER=`whoami`
START=`date +%d-%m-%Y_%H%M%S`
OS=`uname -a`
LOGDIR="/home/$USER/pslogs"
LOGFILE="$START-$USER-process.log"
PROCESS="processloggerd"


#ps -u $USER | grep -c $PROCESS
if [ `ps -u $USER | grep -c $PROCESS` -gt 2 ]
then
        echo `date` ": instance already running, new $PROCESS not starting!" >> $LOGDIR/error.log
        exit 0
fi

if [ ! -e $LOGDIR ]; then
        mkdir $LOGDIR
fi

# Doing some cleanup...

cd $LOGDIR

for LOGS in *.log
do
        if [ -e $LOGS ]; then
                tar -czf $LOGS.tar.gz $LOGS
        else
                break
        fi
done

rm *.log

# Starting the logging

echo "--- Logging started $START os: $OS filename: $LOGFILE---" >> $LOGDIR/$LOGFILE

while [ true ]

do

        echo `date +%d-%m-%Y_%H%M%S` >> $LOGDIR/$LOGFILE
        ps -u $USER -o lstart,time,c,etime,ppid,pid,stat,%cpu,%mem,comm >> $LOGDIR/$LOGFILE
        echo "#" >> $LOGDIR/$LOGFILE
        sleep 5

done
```

processloggerd.vbs

```
'---------------------------------------'
' Process logger for master thesis 2007 '
' Edson Ochoa                           '
' Oslo University College               '
'---------------------------------------'

' Checking if there are more wscript.exe processes running

        Dim objWMIService, colProcess, objProcess
        Dim intCounter, strProcess

        intCounter = 0
        strProcess="'wscript.exe'"

        Set objWMIService = GetObject("winmgmts:" _
        & "{impersonationLevel=impersonate}!\\" _
        & "." & "\root\cimv2")

        Set colProcess = objWMIService.ExecQuery _
        ("Select * from Win32_Process Where Name = " & strProcess )

        For Each objProcess in colProcess
                   intCounter = intCounter +1
        Next

        If intCounter > 1 Then
                WScript.Echo intCounter & " instance(s) of the processlogger seems to be running - aborting"
                WScript.Quit
        End If
' Setting system variables

        Dim WshShell, WshSysEnv, WshNetwork

        Set WshShell = CreateObject("WScript.Shell")
        Set WshSysEnv = WshShell.Environment("PROCESS")
        Set WshNetwork = WScript.CreateObject("WScript.Network")

        strHomePath = WshSysEnv("USERPROFILE")
        strUser = WshNetwork.UserName
        strComputer = WshNetwork.ComputerName


' Creating necessary directory and files

        Dim objFSO, objFolder, objShell, objTextFile, objFile, strText, dtmSessionFiles

        strSessionStart = Now()
        strSessionStart = Replace( strSessionStart, ":","" ) 'removing :
        strSessionStart = Replace( strSessionStart, " ", "_" )'replacing spaces with _
        strSessionStart = Replace( strSessionStart, ".", "-" )' replacing . with -
        strSessionStart = Replace( strSessionStart, "/", "-" )' replacing / with _

        ' Setting the directory to save logs
        strDirectory = strHomePath & "\pslogs\"

        ' Setting the file name
        strFileName = strSessionStart & "-" & strUser & "-process.log"

        Set objFSO = CreateObject( "Scripting.FileSystemObject" )

        If objFSO.FolderExists( strDirectory ) Then
                Set objFolder = objFSO.GetFolder( strDirectory )
        Else
                Set objFolder = objFSO.CreateFolder( strDirectory )
                strMessage = strMessage & "Log directory" & strDirectory _
                        & " created." & VBNewLine
        End If

        If objFSO.FileExists(strDirectory & strFileName) Then
                Set objFile = objFSO.GetFolder(strDirectory)
        Else
                Set objFile = objFSO.CreateTextFile(strDirectory & strFileName)
                strMessage = strMessage & "Log file: " & strDirectory & strFileName & " created."
        End If

        Set objFile = nothing
        Set objFolder = nothing

        WScript.Echo strMessage


' Getting the processes
```

```
        Dim strHost, dtmNow
        Dim strStartDate, intParentProcessId, intProcessId, strProcessCaption
        Dim strProcessDescription, strProcessPath, intElapsedTime, strProcessOwner

        strHost="."
        'strProcessList = NULL

        Set objWMIService = GetObject( "winmgmts:" _
        & "{impersonationLevel=impersonate}!\\" _
        & strHost & "\root/cimv2")


'Dim intCount
'intCount = 0

Do While True

        strProcessList = null
        strProcessList = VBNewLine & Now() & VBNewLine & "StartDate, Elapsed, ParentId, Id, Caption, Description, Path, Owner"
        Set colProcess = objWMIService.ExecQuery _
        ("Select * from Win32_Process ")

                For Each objProcess in colProcess

                On Error Resume Next
                        If objProcess.GetOwner( strUser ) = 0 Then

                                ' Counting seconds since creation of the process
                                If Len( strStartDate ) > 0 Then
                                        intElapsedTime = Mid(strStartDate, 1, 14)
                                        intElapsedTime = Mid(intElapsedTime, 1, 4) & "-" & Mid(intElapsedTime, 5, 2) _
                                         & "-" & Mid(intElapsedTime, 7, 2) & " " & Mid(intElapsedTime, 9, 2) & ":" _
                                          & Mid(intElapsedTime, 11, 2) & ":" & Mid(intElapsedTime, 13, 2)

                                        intElapsedTime = DateDiff("s", intElapsedTime, Now())
                                End If

                                strProcessOwner = strUser & "/" & strComputer
                                intProcessId =        objProcess.ProcessId
                                intParentProcessId = objProcess.ParentProcessId
                                strProcessCaption = objProcess.Caption
                                strProcessDescription = objProcess.Description
                                strProcessPath = objProcess.ExecutablePath
                                strProcessOwner = strUser & "/" & strComputer
                                strStartDate = objProcess.CreationDate

                                ' Testing ....
                        '        WScript.Echo "Startdate: " & strStartDate
                        '        WScript.Echo "Elapsed time: " & intElapsedTime
                        '        WScript.Echo "PID: " & intProcessId
                        '        WScript.Echo "PPID: " & intParentProcessId
                        '        WScript.Echo "Name: " & strProcessCaption
                        '        WScript.Echo "Caption: " & strProcessCaption
                        '        WScript.Echo "Owner: " & strProcessOwner
                        '        WScript.Echo "Path: " & strProcessPath


                                strProcessList = strProcessList & VBNewline &_
                                        strStartDate & ","& intElapsedTime & "," & intParentProcessId & "," & intProcessId & "," _
                                        & strProcessCaption & "," & strProcessDescription & "," & strProcessPath & "," _
                                        & "," & strProcessOwner

                        End If

                On Error GoTo 0

        Next

        Set objTextFile = objFSO.OpenTextFile(strDirectory & strFileName, 8, True)
        objTextFile.WriteLine( strProcessList )
        objTextFile.Close

'        intCount = intCount+1
'        WScript.Echo Now()
'        WScript.Echo intCount

        WScript.Sleep 5000 ' Sleeps for 5 seconds


Loop
```

winparser-0.1.py

```
#!/usr/bin/python

# Script for parsing windows logs
# Master thesis 2007 - Edson Ochoa
# Oslo University College

import os, sys, re, string
import MySQLdb

filename = sys.argv[1]

# ignoring reference text
ignore_datestamp="^\d{1,2}[/]\d{1,2}[/]\d{1,4}\s\d{1,2}[:]\d{1,2}[:]\d{1,2}(\sAM|PM)*"
ignore_tabtext="StartDate, Elapsed, ParentId, Id, Caption, Description, Path, Owner"
ignore_network="NETWORK SERVICE|LOCAL SERVICE|LOKAL TJENESTE|SYSTEM|NETTVERKSTJENESTE|ASPNET"

# this functions removes unecessary lines as non-user processes and
def removelines( file ):

        f = open( file, 'r')

        biglist=[]

        for line in f.xreadlines():
                if re.search(ignore_datestamp, line) or re.search(ignore_tabtext, line) or re.search(ignore_network, line):
                        continue

                values = line.split(',')
                if len(values) == 9:
                        # storing: id, creation timestamp, caption, elapsed time, owner
                        biglist.append( values[3] + "," + values[0] + "," + values[4] + "," + values[1] + "," + values[8] )

        f.close()
        return biglist

# function for creating dictionary from list.
def trim( untrimmed ):
        trimmed={}

        for line in untrimmed:
                value = line.split(',')
                pid = value[0]
                creation = value[1]
                caption = value[2]
                elapsed = value[3]
                owner = value[4]

                elapsed = int( elapsed )
                # Checking if the process already exists
                if trimmed.has_key( pid + "," + creation + "," + caption + "," + owner ):
                        # Checking if the elapsed time is greater, if yes -> replace entry
                        if trimmed[ pid + "," + creation + "," + caption + "," + owner ] < elapsed:
                                #print trimmed[ pid + "," + creation + "," + caption + "," + owner ]
                                trimmed[ pid + "," + creation + "," + caption + "," + owner ] = elapsed
                                continue
                        continue

                trimmed[ pid + "," + creation + "," + caption + "," + owner ] = elapsed

        return trimmed

def formatdate( windt ):
        # timeformat: 20070218120851.525200+060
        pattern = "(\d{4})(\d{2})(\d{2})(\d{2})(\d{2})(\d{2})"
        res = re.match( pattern, windt )

        datestring = res.group(1) + "-" + res.group(2) + "-" + res.group(3) + " " + res.group(4) + ":" + res.group(5) + ":" + res.group(5)
        return datestring


def writetofile( trimmed ):

        writelist = []
        for line in trimmed.keys():
                value = line.split(',')
                pid = value[0]
                timestamp = value[1]
                caption = value[2]
                owner = value[3].rstrip("\r\n")
                elapsed = trimmed[ value[0] + "," + value[1] + "," + value[2] + "," + value[3] ]

                timestamp = formatdate( timestamp )
                writelist.append( timestamp + "," + caption + "," + pid + "," + owner + "," + str(elapsed) )

        return writelist

def inserttodb( list ):
        conn = MySQLdb.connect( host="localhost", user="root", passwd="bl44s3f1sk", db="thesis")
        cursor = conn.cursor()

        icount = 0
        ucount = 0

        for line in list:
                value = line.split(',')
                timestamp = value[0]
                caption = value[1]
                pid = value[2]
                owner = value[3]
                elapsed = value[4]


                # Counting specific users
                cursor.execute( "SELECT * FROM users WHERE strUser = %s", ( owner ) )
                num_users = int( cursor.rowcount )

                if num_users == 0:
                        cursor.execute( """INSERT INTO users VALUES( 0, %s, "", 1 )""", ( owner ) )

                # Counting specific records
                cursor.execute( """
                        SELECT * FROM processentries
                                WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                AND dtmCreationDate = %s
                                AND intProcessId = %s
                                AND strCaption = %s""", ( owner, timestamp, int( pid ), caption ) )
                num_process = int( cursor.rowcount )

                if num_process == 0:
                        cursor.execute( """
                        INSERT INTO processentries
                        VALUES( 0, %s, %s, %s, (SELECT intUserId FROM users WHERE strUser = %s),%s )""", ( timestamp, int(pid), int(ela
                        icount = icount + cursor.rowcount
                # A record exists and we want to find out if the elapsed time is higher
                if num_process != 0:

                        cursor.execute( """
                                SELECT intElapsed, strCaption, intProcessId FROM processentries
                                WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                AND dtmCreationDate = %s
                                AND intProcessId = %s
                                AND strCaption = %s""", ( owner, timestamp, int( pid ), caption ) )

                        row = cursor.fetchone()
                        #print str( row[0] ) + " - " + str( row[1] ) + " - " + str( row[2] )

                        if elapsed > row[0]:

                                #print elapsed > row[0]
                                #print str(row[1]) +"-" + str(row[2]) + ": " + str(elapsed) + " > " + str(row[0])
                                cursor.execute( """
                                        UPDATE processentries SET intElapsed = %s
                                        WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                        AND dtmCreationDate = %s
                                        AND intProcessId = %s
                                        AND strCaption = %s""", ( elapsed, owner, timestamp, int( pid ), caption ) )

                                ucount = ucount + cursor.rowcount

        print "Inserted records: " + str( icount )
        print "Updated records: " + str( ucount )


cleanlist = removelines( filename )
trimmed = trim( cleanlist )
#trimmed.sort()

print "Old list: " + str( len( cleanlist ) )
print "New list: " + str( len ( trimmed ) )
#print "\n"
#print "CONTENT OF NEW LIST:"
#print "Datetimestamp, caption, processid, owner, elapsed time (s)"
towrite = writetofile( trimmed )
towrite.sort()

#for l in towrite:
#        print l

print "Adding to database... "
```

```
inserttodb( towrite )
print "Done!"

#for key, value in trimmed.items():
#        print key + " -> " + str(value)

print "----------------------------------------------------------------------------------------------------------"
trimmed.clear()
```

linparser-0.1.py

```
#!/usr/bin/python

# Script for parsing linux logs
# Master thesis 2007 - Edson Ochoa
# Oslo University College

import os, sys, re, string
import MySQLdb

filename = sys.argv[1]

# ignoring reference text
ignore_datestamp="^\d{1,2}[/]\d{1,2}[/]\d{1,4}\s\d{1,2}[:]\d{1,2}[:]\d{1,2}(\sAM|PM)*"
ignore_tabtext="StartDate, Elapsed, ParentId, Id, Caption, Description, Path, Owner"
ignore_network="NETWORK SERVICE|LOCAL SERVICE|LOKAL TJENESTE|SYSTEM|NETTVERKSTJENESTE|ASPNET"

# this functions removes unecessary lines as non-user processes and
def removelines( file ):

        f = open( file, 'r')

        biglist=[]

        for line in f.xreadlines():
                if re.search(ignore_datestamp, line) or re.search(ignore_tabtext, line) or re.search(ignore_network, line):
                        continue

                values = line.split(',')
                if len(values) == 9:
                        # storing: id, creation timestamp, caption, elapsed time, owner
                        biglist.append( values[3] + "," + values[0] + "," + values[4] + "," + values[1] + "," + values[8] )

        f.close()
        return biglist

# function for creating dictionary from list.
def trim( untrimmed ):
        trimmed={}

        for line in untrimmed:
                value = line.split(',')
                pid = value[0]
                creation = value[1]
                caption = value[2]
                elapsed = value[3]
                owner = value[4]

                elapsed = int( elapsed )
                # Checking if the process already exists
                if trimmed.has_key( pid + "," + creation + "," + caption + "," + owner ):
                        # Checking if the elapsed time is greater, if yes -> replace entry
                        if trimmed[ pid + "," + creation + "," + caption + "," + owner ] < elapsed:
                                #print trimmed[ pid + "," + creation + "," + caption + "," + owner ]
                                trimmed[ pid + "," + creation + "," + caption + "," + owner ] = elapsed
                                continue
                        continue

                trimmed[ pid + "," + creation + "," + caption + "," + owner ] = elapsed

        return trimmed

def formatdate( windt ):
        # timeformat: 20070218120851.525200+060
        pattern = "(\d{4})(\d{2})(\d{2})(\d{2})(\d{2})(\d{2})"
        res = re.match( pattern, windt )

        datestring = res.group(1) + "-" + res.group(2) + "-" + res.group(3) + " " + res.group(4) + ":" + res.group(5) + ":" + res.group
        return datestring


def writetofile( trimmed ):

        writelist = []
        for line in trimmed.keys():
                value = line.split(',')
                pid = value[0]
                timestamp = value[1]
                caption = value[2]
                owner = value[3].rstrip("\r\n")
                elapsed = trimmed[ value[0] + "," + value[1] + "," + value[2] + "," + value[3] ]

                timestamp = formatdate( timestamp )
                writelist.append( timestamp + "," + caption + "," + pid + "," + owner + "," + str(elapsed) )

        return writelist

def inserttodb( list ):
        conn = MySQLdb.connect( host="localhost", user="root", passwd="bl44s3f1sk", db="thesis")
        cursor = conn.cursor()

        icount = 0
        ucount = 0

        for line in list:
                value = line.split(',')
                timestamp = value[0]
                caption = value[1]
                pid = value[2]
                owner = value[3]
                elapsed = value[4]


                # Counting specific users
                cursor.execute( "SELECT * FROM users WHERE strUser = %s", ( owner ) )
                num_users = int( cursor.rowcount )

                if num_users == 0:
                        cursor.execute( """INSERT INTO users VALUES( 0, %s, "", 1 )""", ( owner ) )

                # Counting specific records
                cursor.execute( """
                        SELECT * FROM processentries
                                WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                AND dtmCreationDate = %s
                                AND intProcessId = %s
                                AND strCaption = %s""", ( owner, timestamp, int( pid ), caption ) )
                num_process = int( cursor.rowcount )

                if num_process == 0:
                        cursor.execute( """
                        INSERT INTO processentries
                        VALUES( 0, %s, %s, %s, (SELECT intUserId FROM users WHERE strUser = %s),%s )""", ( timestamp, int(pid), int(elapse
                        icount = icount + cursor.rowcount
                # A record exists and we want to find out if the elapsed time is higher
                if num_process != 0:

                        cursor.execute( """
                                SELECT intElapsed, strCaption, intProcessId FROM processentries
                                WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                AND dtmCreationDate = %s
                                AND intProcessId = %s
                                AND strCaption = %s""", ( owner, timestamp, int( pid ), caption ) )

                        row = cursor.fetchone()
                        #print str( row[0] ) + " - " + str( row[1] ) + " - " + str( row[2] )

                        if elapsed > row[0]:

                                #print elapsed > row[0]
                                #print str(row[1]) +"-" + str(row[2]) + ": " + str(elapsed) + " > " + str(row[0])
                                cursor.execute( """
                                        UPDATE processentries SET intElapsed = %s
                                        WHERE intUserId = (SELECT intUserId from users WHERE strUser = %s)
                                        AND dtmCreationDate = %s
                                        AND intProcessId = %s
                                        AND strCaption = %s""", ( elapsed, owner, timestamp, int( pid ), caption ) )

                                ucount = ucount + cursor.rowcount

        print "Inserted records: " + str( icount )
        print "Updated records: " + str( ucount )


cleanlist = removelines( filename )
trimmed = trim( cleanlist )
#trimmed.sort()

print "Old list: " + str( len( cleanlist ) )
print "New list: " + str( len ( trimmed ) )
#print "\n"
#print "CONTENT OF NEW LIST:"
#print "Datetimestamp, caption, processid, owner, elapsed time (s)"
towrite = writetofile( trimmed )
towrite.sort()

#for l in towrite:
#        print l

print "Adding to database... "
```

```
inserttodb( towrite )
print "Done!"

#for key, value in trimmed.items():
#        print key + " -> " + str(value)

print "------------------------------------------------------------------------------------------------------------"
trimmed.clear()
```